JMS

JMS является еще одной технологией создания распределенных приложений, основанных на модели обмена сообщениями.

Введение

JMS (Java Messaging System) представляет собой интерфейс к внешним системам, ориентированный на работу через сообщения. JMS является «старой» технологией – первая спецификация была опубликована в 1998г. В настоящее время пакет javax.jms входит в комплект jdk, а Sun Application Server реализует поддержку JMS в качестве одного из сервисов.

При разработке JMS в качестве основной задачи рассматривалось создание обобщенного Java API для приложений, ориентированных на работу с сообщениями (message-oriented application programming) и обеспечение независимости от конкретных реализаций соответствующих служб обработки сообщений. Таким образом, программа, написанная с использованием JMS, будет корректно работать с любой системой сообщений, поддерживающей эту спецификацию (или имеющую соответствующие интерфейсы).

Поскольку JMS является лишь оболочкой или интерфейсом, описывающим доступные для приложения методы, для работы приложения понадобится определенная реализация этих интерфейсов JMS, называемая провайдером JMS. Такие реализации создаются независимыми производителями, и в настоящее время таких реализаций существует достаточно много (в том числе, например, реализация, включенная в Sun Application Server и распространяемая вместе с J2EE, а также MQSeries от IBM, служба JMS WebLogic от BEA, SonicMQ от Progress и другие).

Модель обмена сообщениями (и JMS) удобно использовать в том случае, если распределенное приложение обладает следующими характеристиками:

- взаимодействие между компонентами является асинхронным; информация (сообщение) должна передаваться нескольким или даже всем компонентам системы (семантика передачи от одного ко многим);
- передаваемая информация используется многими внешними системами, часть из которых не известна на момент проектирования системы или интерфейсы которых подвержены частым изменениям (концепция ESB – Enterprise Service Bus);
- обменивающиеся информацией (сообщениями) компоненты выполняются в разное время, что требует наличия посредника для промежуточного хранения переданной информации.

Цели JMS

Для лучшего понимания JMS нужно знать цели, которые ставили перед собой создатели спецификации JMS.

В настоящее время на рынке существует множество корпоративных систем обмена сообщениями, и несколько компаний, производящих эти системы, были вовлечены в разработку JMS.

Эти имеющиеся системы различаются по возможностям и функциональности. Авторы знали, что JMS будет слишком сложной и громоздкой, если объединит все функциональные возможности всех существующих систем. Также они полагали, что не должны ограничивать себя только лишь функциями, общими для всех систем.

Авторы полагали, что важно было включить в JMS всю функциональность, необходимую для реализации "усовершенствованных корпоративных приложений".

Целями JMS, как утверждается в спецификации, являются:

- Определить общий набор концепций и возможностей системы обмена сообщениями.
- Минимизировать концепции, которые должен изучить программист для использования корпоративной системы обмена сообщениями.
- Максимизировать переносимость приложений, работающих с системой обмена сообщениями.
- Минимизировать работу, требуемую для реализации провайдера.

Предоставить клиентские интерфейсы для обоих доменов: "точка-точка" и pub/sub. "Домены" - это термин JMS для обозначения моделей обмена сообщениями, рассмотренных ранее. Примечание: Провайдер не обязан реализовывать оба домена.

Чего не обеспечивает JMS

Некоторые функциональные возможности, свойственные MOM-продуктам, не являются предметом рассмотрения в JMS-спецификации. Эти возможности хотя и признаны авторами JMS важными для разработки устойчивых приложений систем обмена сообщениями, но считаются зависящими от JMS-провайдера.

JMS-провайдеры могут реализовывать следующие функциональные возможности любым желаемым способом:

- Распределение нагрузки и отказоустойчивость.
- Система сообщений и уведомлений об ошибках и подсказки.
- Администрирование.
- Защита.
- Протокол связи.
- Репозиторий типов сообщений.

Архитектура JMS

Архитектура JMS выглядит следующим образом (Рис. 1.):

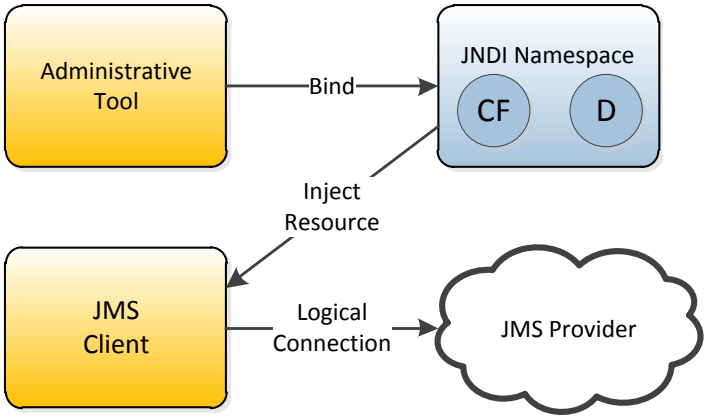

Рис. 1. Архитектура JMS

- Прикладные программы Java, использующие JMS, называются клиентами JMS (JMS client);
- Система обработки сообщений, управляющая маршрутизацией и доставкой сообщений, называется JMS-провайдером (JMS provider);
- Приложение JMS (JMS application) – это прикладная система, состоящая из нескольких JMS клиентов, и, как правило, одного JMS-провайдера. JMS-клиент, посылающий сообщение, называется поставщиком (producer). JMS-клиент, принимающий сообщение, называется потребителем (consumer). Один и тот же JMS клиент может быть одновременно и поставщиком и потребителем в разных актах взаимодействия;
- Сообщения (Messages) – это объекты, передающиеся и принимающиеся компонентами (клиентами JMS);
- Средства администрирования (Administrative tools) – средства управления ресурсами, использующимися клиентами.

JMS предоставляет два подхода к передаче сообщений. Первый называется «издание-подписка» (publish an subscribe) (Рис. 3.) и используется в том случае, если сообщение, отправленное одним клиентом должно быть получено несколькими.

Второй подход называется «точка-точка» (point to point) (Рис. 2.) и служит для реализации обмена сообщениями между двумя компонентами.

Спецификация JMS называет эти два подхода зонами сообщений (messaging domains).

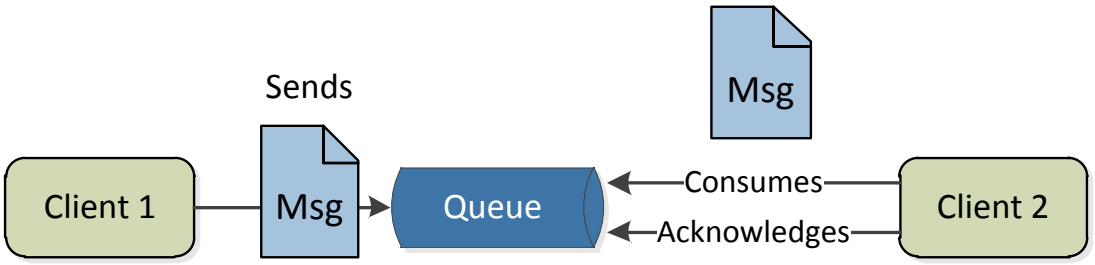

Рис. 2. Модель взаимодействия «точка-точка»

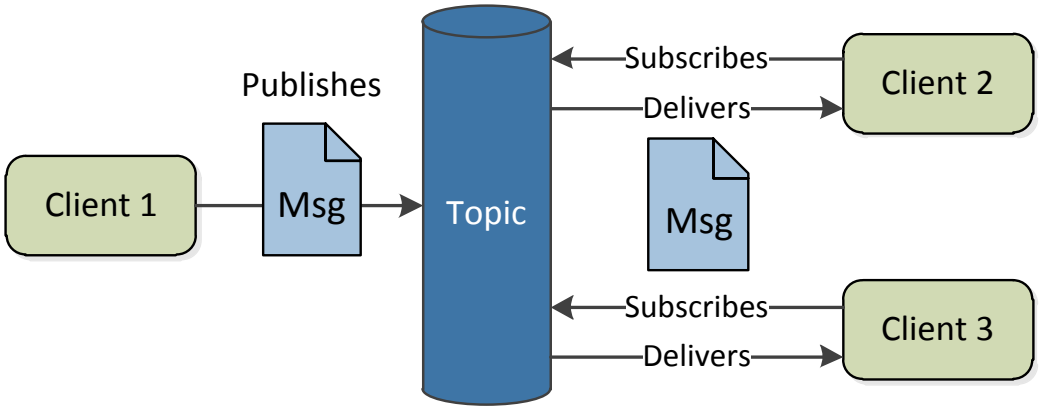

Рис. 3. Модель взаимодействия «издание-подписка»

Модель взаимодействия точка-точка

Модель передачи сообщений «точка-точка» предоставляет возможность клиентам JMS посылать и принимать сообщения (как синхронно, так и асинхронно) через виртуальные каналы, называемые очередями (queues). Модель передачи сообщений «точка-точка» основывается на методе опроса, при котором сообщения явно запрашиваются (считываются) клиентом из очереди. Несмотря на то, что чтение из очереди могут осуществлять несколько клиентов, каждое сообщение будет прочитано только единожды - провайдер JMS это гарантирует.

Модель взаимодействия издание-подписка

При использовании модели взаимодействия «издание-подписка» один клиент (поставщик) может посылать сообщения многим клиентам (потребителям) через виртуальный канал, называемый темой (topic). Потребители могут выбрать подписку (subscribe) на любую тему. Все сообщения, направляемые в тему, передаются всем потребителям данной темы. Каждый потребитель принимает копию каждого сообщения. Модель передачи сообщений издание-подписка, по существу, представляет собой модель, сервера, инициирующего соединение и «проталкивающего» информацию на клиента. В JMS эта концепция реализуется с помощью специальных «слушателей» (листенеров), регистрируемых в системе. При возникновении нового события листенер, закрепленный за данной темой, возбуждается.

Следует отметить, что при использовании модели «издание-подписка» клиенты JMS могут устанавливать долговременные подписки, позволяющие потребителям отсоединиться и позже снова подключиться и получать сообщения, поступившие во время отключения связи.

Использование JMS

Использование JMS предполагает выполнение разработчиком последовательности шагов. Ниже приведена общая схема использования JMS API.

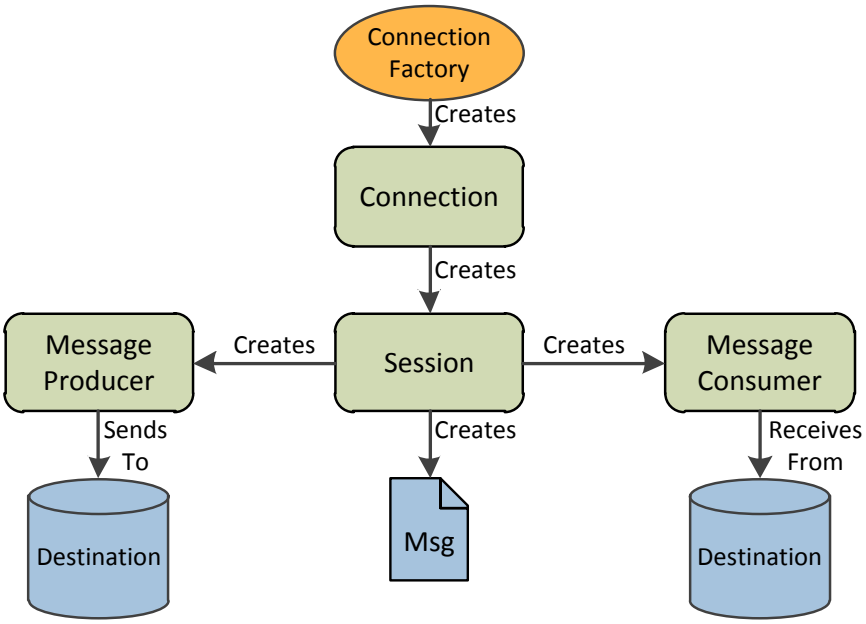

Рис. 4. Общая схема использования JMS API

Любой компонент, использующий JMS, прежде всего, должен создать соединение с JMS провайдером – собственно системой, обеспечивающей всю функциональности управления сообщениями.

Для этого используется специальная фабрика объектов - ConnectionFactory. ConnectionFactory на самом деле является интерфейсом, от которого наследуют QueueConnectionFactory, TopicConnectionFactory, XAQueueConnectionFactory и XATopicConnectionFactory. Таким образом, мы будем иметь дело с реализацией одного из этих интерфейсов. Для того, чтобы получить доступ к ConnectionFactory, программа должна извлечь соответствующую ссылку из JNDI. С использованием механизма аннотаций соответствующий код будет иметь следующий вид:

```
@Resource(mappedName="jms/ConnectionFactory")
private static ConnectionFactory connectionFactory;
```

Указанный фрагмент кода извлекает ресурс с JNDI-именем jms/ConnectionFactory и связывает его с переменной connectionFactory. Естественно, перед первым использованием ConnectionFactory должна быть создана:

```
asadmin.bat create-jms-resource --user admin --passwordfile admin-password.txt --host
localhost  --port 4848 --restype javax.jms.ConnectionFactory --enabled=true
jms/ConnectionFactory
```

Данная команда создает ресурс типа javax.jms.ConnectionFactory с именем jms/ConnectionFactory.

Следующим шагом является создание соединения с JMS провайдером. Соответствующий код будет выглядеть следующим образом:

```
Connection connection = connectionFactory.createConnection();
```

Следует отметить, что соединение должно быть закрыто после того, как оно более не используется (connection.close()).

После того, как соединение создано, могут быть созданы виртуальные каналы, в рамках которых будет осуществляться передача сообщений. Существуют два типа таких каналов – очередь (Queue) и тема (Topic). Следует отметить, что создание виртуального канала требует наличия соответствующего «пункта назначения» (destination), созданного в JNDI. Таким образом, прежде чем программа сможет использовать очередь (или тему) соответствующий объект должен быть создан в JMS провайдере. Для Sun Application Server соответствующая команда будет иметь вид:

```
asadmin.bat create-jms-resource --user admin --passwordfile admin-password.txt –host
localhost --port 4848 --restype javax.jms.Queue --enabled=true --property
Name=PhysicalQueue jms/Queue
```

где restype указывает тип объекта – в данном случае это javax.jms.Queue, а параметр property задает JNDI имя объекта - jms/Queue.

Для того, чтобы в программе получить ссылку на соответствующий объект, можно использовать код вида:

```
@Resource(mappedName="jms/Queue")
private static Queue queue;
```

для очереди, или, если нужно получить ссылку на объект типа тема:

```
@Resource(mappedName="jms/Topic")
private static Topic topic;
```

Следует обратить внимание на то, что объекты с именами jms/Queue или jms/Topic должны быть предварительно созданы. После того, как соединение с JMS провайдером установлено и получены ссылки на соответствующие пункты назначения (очередь или тема), может начаться собственно процесс обмена сообщениями.

Весь процесс обмена происходит в рамках сессии (Session), которая представляет собой однопотоковый контекст для обмена сообщениями. Сессия также предоставляет транзакционный контекст для обеспечения атомарности выполнения групп передач/приемов сообщений.

Сессия может быть создана следующим образом:

```
Session session = connection.createSession(false,  Session.AUTO_ACKNOWLEDGE);
```

где первый параметр указывает на то, что секция не транзакционна, а второй параметр указывает на то, что необходимо автоматически подтверждать успешную доставку сообщений.

В рамках сессии приложение может создать ряд объектов, обеспечивающих отправку и приемку сообщений. Это Message Producers и Message Consumers. Создаются эти объекты следующим образом:

```
MessageProducer producer = session.createProducer(queue);
MessageConsumer consumer = session.createConsumer(queue);
```

В качестве аргумента методы создания требуют указания пункта назначения. После создания посредством MessageProducer приложение может отправлять сообщения, а посредством MessageConsumer – принимать. Для отправки сообщения достаточно вызвать метод send, передав ему в качестве аргумента объект типа Message:

```
producer.send(message)
```

Для синхронного приема сообщения необходимо сначала запустить диспетчеризацию сообщения, затем вызвать метод receive соответствующего MessageConsumer. Метод receive является блокирующим, однако он может быть вызван с указанием количества миллисекунд, в течении которых он должен ожидать чтения сообщения.

```
connection.start();
Message m = consumer.receive();
```

при этом управление вернется в момент прихода сообщения, или connection.start();

```
Message m = consumer.receive(2000);
```

при этом управление вернется в момент прихода сообщения или по истечению 2 секунд.

Для асинхронного приема сообщения может быть создан специальный класс – листенер, который должен быть зарегистрирован в рамках данного MessageConsumer и метод которого onMessage будет вызван в момент прихода сообщения.

```
Listener myListener = new Listener();
consumer.setMessageListener(myListener);
```

Типы сообщений

Сообщение в JMS – это объект Java, состоящий из двух частей: заголовка (header) и тела (body) сообщения. В заголовке находится служебная информация, тело сообщения содержит в себе пользовательские данные, которые могут быть разной формы: текстовой, сериализуемые объекты, байтовые потоки и т. д.

JMS API определяет несколько типов сообщений:

- BytesMessage предназначен для передачи потока байт, который система никак не интерпретирует;

- MapMessage предназначен для передачи множества элементов типа «имя-значение», где имена являются объектами строкового типа, а значения – объектами примитивных типов данных Java;
- ObjectMessage предназначен для передачи сериализуемых объектов;
- StreamMessage предназначен для передачи множества элементов примитивных типов данных Java (они могут быть последовательно записаны, а затем прочитаны из тела сообщения этого типа);
- TextMessage предназначен для передачи текстовой информации.

Разработка JMS-программы

Типичная JMS-программа проходит следующие этапы перед началом производства и потребления сообщений:

1. Ищет ConnectionFactory через JNDI.
2. Ищет одно или более Destination через JNDI.
3. Использует ConnectionFactory для создания Connection.
4. Использует Connection для создания одной или более Session.
5. Использует Session и Destination для создания необходимых MessageProducer и MessageConsumer.
6. Запускает Connection.

С этого момента сообщения могут начинать перемещение, и приложение может получать, обрабатывать и передавать сообщения при необходимости. В следующих разделах мы разработаем JMS-программы и рассмотрим эти этапы более подробно.

Сообщения

Сердцем системы обмена сообщениями, естественно, являются сообщения. JMS предоставляет несколько типов сообщений для различных типов содержимого, но все сообщения наследуются из интерфейса `Message`.

`Message` разделяется на три составные части:

- **Заголовок (header)** - это стандартный набор полей, используемых клиентами и провайдерами для идентификации и маршрутизации сообщений.

- **Свойства** предоставляют возможность добавления необязательных полей заголовка сообщения. Если вашему приложению нужно категоризировать или классифицировать сообщения способом, не предоставляемым стандартными полями заголовка, вы можете добавить свойство к сообщению для выполнения этой категоризации или классификации. Для получения и установки свойств различных Java-типов, включая `Object`, применяются методы `set<Type>Property(...)` и `get<Type>Property(...)`. JMS определяет стандартный набор свойств, необязательных для предоставления провайдерами.

- **Тело (body)** сообщения содержит информацию, передаваемую в принимающее приложение. Каждый интерфейс сообщения специализирован для поддерживаемого им типа содержимого.

Поля заголовка

В следующем списке перечислены все поля заголовка `Message`, их соответствующий Java-тип и описание поля:

- `JMSMessageID` - тип `string`
  Однозначно идентифицирует каждое сообщение, передаваемое провайдером. Это поле устанавливается провайдером во время передачи; клиенты не могут определить `JMSMessageID` сообщения до его передачи.
- `JMSDestination` - тип `Destination`
  Адресат, которому передается сообщение; устанавливается провайдером во время процесса передачи.
- `JMSDeliveryMode` - тип `int`
  Содержит значение `DeliveryMode.PERSISTENT` или `DeliveryMode.NON_PERSISTENT`. Персистентное сообщение доставляется "один раз и только один раз"; не персистентное сообщение доставляется "не более одного раза". Знайте, что "не более одного раза" включает и отсутствие доставки. Не персистентные сообщения могут быть потеряны провайдером во время сбоя приложения или системы. Для гарантирования отсутствия влияния системных сбоев на доставку персистентных сообщений должны выполняться дополнительные действия. Для передачи персистентных сообщений часто необходимы значительные дополнительные накладные расходы, и нужно тщательно рассматривать баланс между надежностью и производительностью при выборе режима доставки сообщения.
- `JMSTimestamp` - тип `long`
  Время, когда сообщение было доставлено провайдеру для передачи; устанавливается провайдером во время процесса передачи.
- `JMSExpiration` - тип `long`
  Время, за которое сообщение должно стать недействительным. Это значение вычисляется во время процесса передачи как сумма значения time-to-live (время жизни) передающего метода и текущего времени.

Недействительные сообщения не должны доставляться провайдером. Значение 0 указывает на то, что сообщение никогда не становится недействительным.

- `JMSPriority` - тип `int`
  Приоритет сообщения; устанавливается провайдером во время процесса передачи. Приоритет 0 является самым низким приоритетом; приоритет 9 является самым высоким приоритетом.
- `JMSCorrelationID` - тип `string`
  Обычно используется для связи ответного сообщения с сообщением запроса; устанавливается JMS-программой, передающей сообщение. JMS-программа, отвечающая на сообщение от другой JMS-программы, должна скопировать `JMSMessageID` сообщения, на которое она отвечает, в это поле, для того чтобы запрашивающая программа могла *связать* ответ с конкретным запросом.
- `JMSReplyTo` - тип `Destination`
  Используется запрашивающей программой для указания места, куда должно быть передано ответное сообщение; устанавливается JMS-программой, передающей сообщение.
- `JMSType` - тип `string`
  Может использоваться JMS-программой для указания типа сообщения. Некоторые провайдеры поддерживают репозиторий типов сообщений и используют это поле для ссылки на определение типа в этом репозитории; в этом случае JMS-программа не должна использовать это поле.
- `JMSRedelivered` - тип `boolean`
  Указывает, что сообщение было ранее доставлено в JMS-программу, но эта программа не подтвердила его прием; устанавливается провайдером во время процесса приема.

Стандартные свойства

В следующем списке перечислены названия каждого стандартного свойства `Message`, его соответствующий Java-тип и описание свойства. Поддержка стандартных свойств провайдером необязательна. JMS резервирует название свойства "JMSX" для этих и будущих свойств, определенных в JMS.

- `JMSXUserID` - тип `string`
  Идентифицирует пользователя, передающего сообщение.
- `JMSXApplID` - тип `string`
  Идентифицирует приложение, передающее сообщение.
- `JMSXDeliveryCount` - тип `int`
  Количество сделанных попыток доставки сообщения.
- `JMSXGroupID` - тип `string`
  Идентифицирует группу сообщений, к которой принадлежит данное сообщение.
- `JMSXGroupSeq` - тип `int`
  Последовательный номер этого сообщения в группе сообщений.
- `JMSXProducerTXID` - тип `string`
  Идентифицирует транзакцию, в которой это сообщение производится.
- `JMSXConsumerTXID` - тип `string`
  Идентифицирует транзакцию, в которой это сообщение потребляется.
- `JMSXRcvTimestamp` - тип `long`
  Время доставки сообщения потребителю.
- `JMSXState` - тип `int`
  Используется провайдерами, поддерживающими хранилище сообщений; обычно не используется JMS-производителями или JMS-потребителями.
- `JMSX_<vendor_name>`
  Зарезервировано для специфичных для поставщика свойств.

Тело сообщения

Существует пять форматов тела сообщений, и каждый формат определяется интерфейсом, расширяющим `Message`. Этими интерфейсами являются:

- `StreamMessage`: Содержит поток значений Java-примитивов, который заполняется и читается последовательно при помощи стандартных потоковых операций.
- `MapMessage`: Содержит набор пар имя-значение; имена имеют тип `string`, а значения - это Java-примитивы.
- `TextMessage`: Содержит `String`.
- `ObjectMessage`: Содержит Java-объект `Serializable`; могут использоваться классы коллекций JDK 1.2.
- `BytesMessage`: Содержит поток не интерпретированных байтов; разрешает кодирование тела сообщения для соответствия существующему формату сообщений.

Каждый провайдер предоставляет классы, специфичные для его системы и реализующие эти интерфейсы. Важно отметить, что спецификация JMS требует, чтобы провайдеры были готовы к принятию и обработке объекта `Message`, не являющегося экземпляром одного из их собственных классов `Message`.

Хотя эти "чужие" объекты могут не обрабатываться провайдером так же эффективно, как одна из собственных реализаций, они должны обрабатываться, чтобы гарантировать возможность взаимодействия всех JMS-провайдеров.

Транзакции

JMS-транзакция группирует набор произведенных сообщений и набор потребленных сообщений в атомарную единицу работы. Если во время транзакции происходит ошибка, производство и потребление сообщений, выполненное до ошибки, может быть "отменено".

Объекты Session управляют транзакциями. Объект `Session` может быть отмечен при его создании как *поддерживающий транзакции*. В таких объектах `Session` всегда имеется текущая транзакция, то есть, нет `begin()`, `commit()` и`rollback()`, заканчивающих одну транзакцию и автоматически начинающих другую.
Могут поддерживаться распределенные транзакции посредством Java Transaction API (JTA) `XAResource` API, хотя это не обязательно для провайдеров.

Подтверждение

Подтверждение (acknowledgment) - это механизм, посредством которого провайдер информируется о том, что сообщение было успешно получено.
Если объект `Session`, получающий сообщение, поддерживает транзакции, подтверждение обрабатывается автоматически. В противном случае тип подтверждения определяется при создании `Session`.
Существует три типа подтверждения:

- `Session.DUPS_OK_ACKNOWLEDGE`: Отложенное ("ленивое") подтверждение доставки сообщения; уменьшает накладные расходы, минимизируя работу по предотвращению дублирования; должно использоваться только тогда, когда ожидаются дублированные сообщения, и они могут быть обработаны.

- `Session.AUTO_ACKNOWLEDGE`: Доставка сообщений автоматически подтверждается по завершении метода, принимающего сообщение.

- `Session.CLIENT_ACKNOWLEDGE`: Доставка сообщений подтверждается явно путем вызова метода `acknowledge()` объекта `Message`.

Селектор сообщений

JMS обеспечивает для JMS-программ механизм, называемый *селектором сообщений*, для фильтрации и категоризации принимаемых ими сообщений.

Селектор сообщений является объектом `String`, содержащим выражение, синтаксис которого основан на подмножестве SQL92. Выражение селектора сообщений вычисляется при попытках получения сообщения, и только сообщение, удовлетворяющее критерию, селектор делает доступным для программы.

Выбор основывается на сопоставлении полей заголовка и свойств; содержимое тела сообщения не может участвовать в критериях выбора. Синтаксис для селекторов сообщений определяется в спецификации JMS.

Общие интерфейсы

`ConnectionFactory`

`ConnectionFactory` - это администрируемый объект, извлекаемый из JNDI для создания соединения с провайдером. Он содержит метод `createConnection()`, который возвращает объект `Connection`.

`Connection`

Объект `Connection` инкапсулирует активное соединение с провайдером. Некоторыми из его методов являются:

- `createSession(boolean, int)`: Возвращает объект `Session`. Параметр `boolean` указывает, поддерживает объект `Session` транзакции или нет; параметр `int` указывает режим подтверждения (см. раздел "Подтверждение").

- `start()`: Активирует доставку сообщений от провайдера.

- `stop()`: Временно останавливает доставку сообщений; доставка может быть активирована повторно при помощи метода `start()`.

- `close()`: Закрывает соединение с провайдером и освобождает все ресурсы, занятые для него.

`Session`

Объект `Session` является однопоточным контекстом для передачи и приема сообщений. Некоторыми из его методов являются:

- `createProducer(Destination)`: Возвращает объект `MessageProducer` для передачи сообщений в указанный объект `Destination`.

- `createConsumer(Destination)`: Возвращает объект `MessageConsumer` для приема сообщений от указанного объекта `Destination`.

- `commit()`: Фиксирует все потребленные или произведенные сообщения для текущей транзакции.
- `rollback()`: Делает откат для всех потребленных или произведенных сообщений текущей транзакции.
- `create<MessageType>Message(...)`: Набор методов, возвращающих `<MessageType>Message` - например,`MapMessage`, `TextMessage` и т.д.

Destination

Объект `Destination` инкапсулирует адресат сообщений. Это администрируемый объект, извлекаемый из JNDI.

MessageProducer

Объект `MessageProducer` используется для передачи сообщений. Некоторыми из его методов являются:

- `send(Message)`: Передает указанный объект `Message`.
- `setDeliveryMode(int)`: Устанавливает режим доставки для последующих передаваемых сообщений; корректными значениями являются `DeliveryMode.PERSISTENT` и `DeliveryMode.NON_PERSISTENT`.
- `setPriority(int)`: Устанавливает приоритет для последующих передаваемых сообщений; корректными значениями являются числа от 0 до 9.
- `setTimeToLive(long)`: Устанавливает продолжительность в миллисекундах последующих передаваемых сообщений перед тем, как они станут не действительными.

MessageConsumer

Объект `MessageConsumer` используется для приема сообщений. Некоторыми из его методов являются:

- `receive()`: Возвращает следующее получаемое сообщение; этот метод блокирует работу до тех пор, пока сообщение не станет доступным.
- `receive(long)`: Принимает следующее сообщение, прибывающее в течение указанных в параметре `long`миллисекунд; этот метод возвращает `null`, если за это время не было принято ни одного сообщения.
- `receiveNoWait`: Принимает следующее сообщение, если оно доступно в этот же момент времени; этот метод возвращает `null`, если нет доступных сообщений.
- `setMessageListener(MessageListener)`: Устанавливает `MessageListener`; объект `MessageListener`принимает сообщения по прибытии, то есть, асинхронно.

MessageListener

`MessageListener` - это интерфейс с одним методом `onMessage(Message)`, обеспечивающим асинхронный прием и обработку сообщений.

Этот интерфейс должен быть реализован клиентским классом. Экземпляр этого класса передается объекту`MessageConsumer` при помощи метода `setMessageListener(MessageListener)`. По прибытии сообщения адресату оно передается объекту при помощи метода `onMessage(Message)`.

Программирование клиентских приложений с использованием общих интерфейсов

Sender: Запрос JNDI-имен

Все примеры программ являются программами командной строки, использующими `System.in` для вода и `System.out`для вывода.

Класс `Sender` имеет два метода: `main(String[])` и `send()`. Метод `main(String[])` просто создает экземпляр `Sender` и вызывает его метод `send()`.

Первый фрагмент метода `send()` запрашивает JNDI-имена администрируемых объектов, которые будут использоваться для передачи сообщений.

```
import java.io.*;
import javax.jms.*;
import javax.naming.*;
```

```
public class Sender {

    public static void main(String[] args) {

        new Sender().send();
    }

    public void send() {

        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

        try {
            //Запрос JNDI-имен
            System.out.println("Enter ConnectionFactory name:");
            String factoryName = reader.readLine();
            System.out.println("Enter Destination name:");
            String destinationName = reader.readLine();
    . . .
```

Sender: Поиск администрируемых объектов

Следующий фрагмент метода `send()` ищет администрируемые объекты в JNDI, используя имена, введенные ранее. Доступ к JNDI осуществляется путем создания экземпляра объекта `InitialContext`; администрируемые объекты извлекаются при помощи вызова метода `lookup(String)` и передачи в него имени объекта, который нужно извлечь. Обратите внимание на то, что метод `lookup(String)` возвращает `Object`, поэтому необходимо выполнить операцию приведения типа с возвращаемым объектом.

```
    . . .
            //Поиск администрируемых объектов
            InitialContext initContext = new InitialContext();
            ConnectionFactory factory =
                (ConnectionFactory) initContext.lookup(factoryName);
            Destination destination = (Destination) initContext.lookup(destinationName);
            initContext.close();
            . . .
```

Sender: Создание JMS-объектов

Теперь мы создаем необходимые для передачи JMS-объекты. Обратите внимание на то, что мы не создаем экземпляры этих объектов напрямую с использованием `new`. Все объекты создаются путем вызова метода другого объекта.

Прежде всего, мы используем `ConnectionFactory` для создания `Connection`. Затем мы используем этот объект`Connection` для создания объекта `Session`.

Объект `Session` не поддерживает транзакции (`false`) и будет использовать автоматическое подтверждение (`Session.AUTO_ACKNOWLEDGE`).

Наконец, мы создаем `Sender` для передачи сообщений в объект `Destination`, извлеченный нами из JNDI.

```
    . . .
            //Создать JMS-объекты
            Connection connection = factory.createConnection();
            Session session =
                connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
            MessageProducer sender = session.createProducer(destination);
            . . .
```

Sender: Передача сообщений

Теперь мы готовы передавать сообщения. В данном фрагменте мы вводим цикл, где запрашиваем текст сообщения для передачи. Если пользователь введет *quit*, цикл закончится.

В цикле мы создаем `TextMessage` на основе введенного текста и используем `MessageProducer` для передачи сообщения, затем возвращаемся в начало цикла.

```
    . . .
            //Передать сообщения
            String messageText = null;
            while (true) {
                System.out.println("Enter message to send or 'quit':");
                messageText = reader.readLine();
                if ("quit".equals(messageText))
                    break;
                TextMessage message = session.createTextMessage(messageText);
                sender.send(message);
            }
            . . .
```

Sender: Выход

После завершения цикла мы закрываем `Connection`. Закрытие `Connection` автоматически закрывает `Session` и`MessageProducer`.

```
    . . .
            //Выход
            System.out.println("Exiting...");
            reader.close();
            connection.close();
            System.out.println("Goodbye!");

        } catch (Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

Receiver: Запрос JNDI-имен и поиск администрируемых объектов

Класс `Receiver`, аналогично классу `Sender`, имеет метод `main(String[])` который просто создает экземпляр `Receiver`и вызывает его главный метод `receive()`.

Код запроса JNDI-имен и выполнения поиска администрируемых объектов идентичен коду, использованному в `Sender`.

Однако в этом классе есть два отличия:

- Для указания того, что программа должна завершиться, используется булева переменная экземпляра `stop`.
- `Receiver` реализует интерфейс `MessageListener`, для того чтобы принимать сообщения в асинхронном режиме.

```
import java.io.*;
import javax.jms.*;
import javax.naming.*;

public class Receiver implements MessageListener {

    private boolean stop = false;

    public static void main(String[] args) {

        new Receiver().receive();
    }

    public void receive() {

        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

        try {
            //Запрос JNDI-имен
            System.out.println("Enter ConnectionFactory name:");
            String factoryName = reader.readLine();
            System.out.println("Enter Destination name:");
            String destinationName = reader.readLine();
            reader.close();

            //Поиск администрируемых объектов
            InitialContext initContext = new InitialContext();
            ConnectionFactory factory =
                (ConnectionFactory) initContext.lookup(factoryName);
            Destination destination = (Destination) initContext.lookup(destinationName);
            initContext.close();
            . . .
```

Receiver: Создание JMS-объектов

Объекты `Connection` и `Session` создаются так же, как и в `Sender`, а затем создается `MessageConsumer`.

Далее вызывается метод `setMessageListener()` с аргументом `this` - локальным экземпляром `Receiver`, который реализует интерфейс `MessageListener`.

Наконец, запускается `Connection` для разрешения приема сообщений.

```
    . . .
            //Создать JMS-объекты
            Connection connection = factory.createConnection();
            Session session =
                connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
            MessageConsumer receiver = session.createConsumer(queue);
            receiver.setMessageListener(this);
            connection.start();
            . . .
```

Receiver: Подождать остановки и выйти

Затем программа выполняет цикл, выход из которого производится при равенстве переменной `stop` значению `true`. В цикле поток "спит" в течение одной секунды. После выхода из цикла `Connection` закрывается, и программа завершает работу.

```
    . . .
            //Ожидать останова
            while (!stop) {
                Thread.sleep(1000);
            }

            //Выход
            System.out.println("Exiting...");
            connection.close();
            System.out.println("Goodbye!");

        } catch (Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
    . . .
```

Receiver: Метод onMessage(Message)

Наличие метода `onMessage(Message)` в классе `Receiver` необходимо, поскольку `Receiver` реализует интерфейс`MessageListener`.

При приеме сообщения вызывается этот метод с параметром `Message`.

В нашей реализации мы получаем текстовое содержимое сообщения и выводим его в `System.out`. Затем мы проверяем, записано ли в сообщении *stop*, и, если это так, устанавливаем переменную `stop` в значение `true`; это позволит завершить цикл в методе `receive()`.

```
    . . .
    public void onMessage(Message message) {

        try {
            String msgText = ((TextMessage) message).getText();
            System.out.println(msgText);
            if ("stop".equals(msgText))
                stop = true;
        } catch (JMSException e) {
            e.printStackTrace();
            stop = true;
        }
    }
}
```

Выполнение программ

Для компилирования программ `Sender` и `Receiver` вам необходимы пакеты`javax.naming` и `javax.jms`.

Перед запуском программ вы должны использовать средство администрирования, предоставленное вашим JMS-провайдером, для создания администрируемых объектов `ConnectionFactory` и `Destination` и размещения их в пространстве имен JNDI.

Вам также необходимо записать эти JMS-классы реализации в вашу переменную classpath.

После этого вы можете запустить обе программы одновременно, предоставляя одинаковые JNDI-имена для `ConnectionFactory` и `Destination`, и передавать сообщения из `Sender` в `Receiver`.

Интерфейсы "точка-точка"

`QueueConnectionFactory`

`QueueConnectionFactory` - это администрируемый объект, извлекаемый из JNDI для создания соединения с провайдером. Он содержит метод `createQueueConnection()`, который возвращает объект `QueueConnection`.

`QueueConnection`

`QueueConnection` инкапсулирует активное соединение с провайдером. Некоторыми из его методов являются:

- `createQueueSession(boolean, int)`: Возвращает объект `QueueSession`. Параметр `boolean` указывает на то, поддерживает ли `QueueSession` транзакции или нет; параметр `int` указывает режим подтверждения (см. раздел "Подтверждение").

- `start()` (наследуемый из `Connection`): Активирует доставку сообщений от провайдера.

- `stop()` (наследуемый из `Connection`): Временно останавливает доставку сообщений; доставка может быть активизирована повторно при помощи метода `start()`.

- `close()` (наследуемый из `Connection`): Закрывает соединение с провайдером и освобождает все ресурсы, выделенные для него.

QueueSession

Объект `QueueSession` является однопоточным контекстом для передачи и приема PTP сообщений. Некоторыми из его методов являются:

- `createSender(Queue)`: Возвращает объект `QueueSender` для передачи сообщений в указанный объект `Queue`.
- `createReceiver(Queue)`: Возвращает объект `QueueReceiver` для приема сообщений от указанного объекта`Queue`.
- `createBrowser(Queue)` (наследуемый из `Session`): Возвращает объект `QueueBrowser` для просмотра сообщений в указанном объекте `Queue`.
- `commit()` (наследуемый из `Session`): Фиксирует все потребленные или произведенные сообщения для текущей транзакции.
- `rollback()` (наследуемый из `Session`): Осуществляет откат для всех потребленных или произведенных сообщений текущей транзакции.
- `create<MessageType>Message(...)` (наследуемый из `Session`): Набор методов, возвращающих`<MessageType>Message`, например, `MapMessage`, `TextMessage` и т.д.

Queue

`Queue` инкапсулирует адресат "точка-точка". Это администрируемый объект, извлекаемый из JNDI.

QueueSender

Объект `QueueSender` используется для передачи сообщений "точка-точка". Некоторыми из его методов являются:

- `send(Message)`: Передает указанный объект `Message`.
- `setDeliveryMode(int)` (наследуемый из `MessageProducer`): Устанавливает режим доставки для последующих передаваемых сообщений; корректными значениями являются `DeliveryMode.PERSISTENT` и`DeliveryMode.NON_PERSISTENT`.
- `setPriority(int)` (наследуемый из `MessageProducer`): Устанавливает приоритет для последующих передаваемых сообщений; корректными значениями являются значения от 0 до 9.
- `setTimeToLive(long)` (наследуемый из `MessageProducer`): Устанавливает продолжительность в миллисекундах последующих передаваемых сообщений перед тем, как они станут не действительными.

QueueReceiver

Объект `QueueReceiver` используется для приема сообщений "точка-точка". Некоторыми из его методов являются:

- `receive()` (наследуемый из `MessageConsumer`): Возвращает следующее получаемое сообщение; этот метод блокирует работу до тех пор, пока сообщение не станет доступным.
- `receive(long)` (наследуемый из `MessageConsumer`): Принимает следующее сообщение, прибывающее в течение указанных в параметре `long` миллисекунд; этот метод возвращает `null`, если за это время не было принято ни одного сообщения.
- `receiveNoWait` (наследуемый из `MessageConsumer`): Принимает следующее сообщение, если оно доступно в этот же момент времени; этот метод возвращает `null`, если нет доступных сообщений..
- `setMessageListener(MessageListener)` (наследуемый из `MessageConsumer`): Устанавливает`MessageListener`; объект `MessageListener` принимает сообщения по прибытии, то есть, асинхронно.

QueueBrowser

Когда объект QueueReceiver используется для приема сообщений, сообщения удаляются из очереди после их приема.QueueBrowser используется для поиска сообщений в очереди без их удаления. Для этого применяется методgetEnumeration(), возвращающий java.util.Enumeration, который может быть использован для сканирования сообщений в очереди; изменения в очереди (прибытие и истечение срока жизни сообщений) могут быть видимы, а могут и не быть.

Программирование систем "точка-точка"

QSender: Запрос JNDI-имен

Все примеры программ являются программами командной строки и используют System.in для ввода и System.out для вывода.

Класс QSender имеет два метода: main(String[]) и send(). Метод main(String[]) просто создает экземпляр QSenderи вызывает его метод send().

В первом фрагменте метода send() запрашиваются JNDI-имена администрируемых объектов, используемых для передачи сообщений.

```
import java.io.*;
import javax.jms.*;
import javax.naming.*;

public class QSender {

    public static void main(String[] args) {

        new QSender().send();
    }

    public void send() {

        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

        try {
            //Запрос JNDI-имен
            System.out.println("Enter QueueConnectionFactory name:");
            String factoryName = reader.readLine();
            System.out.println("Enter Queue name:");
            String queueName = reader.readLine();
    . . .
```

QSender: Поиск администрируемых объектов

Следующий фрагмент метода send() ищет администрируемые объекты в JNDI, используя введенные ранее имена.

Доступ к JNDI осуществляется путем создания экземпляра объекта InitialContext; администрируемые объекты извлекаются путем вызова метода lookup(String) и передачи в него в качестве аргумента имени извлекаемого объекта. Обратите внимание на то, что метод lookup(String) возвращает Object, поэтому необходимо выполнить операцию приведения типа с возвращаемым объектом.

```
    . . .
            //Поиск администрируемых объектов
            InitialContext initContext = new InitialContext();
            QueueConnectionFactory factory =
                (QueueConnectionFactory) initContext.lookup(factoryName);
            Queue queue = (Queue) initContext.lookup(queueName);
            initContext.close();
            . . .
```

QSender: Создание JMS-объектов

Теперь мы создаем необходимые для передачи сообщений JMS-объекты. Обратите внимание на то, что мы не создаем экземпляры этих объектов напрямую с использованием new. Все объекты создаются путем вызова метода другого объекта.

Прежде всего, мы используем QueueConnectionFactory для создания QueueConnection. Затем мы используем этот объект QueueConnection для создания объекта QueueSession.

Объект QueueSession не поддерживает транзакции (false) и будет использовать автоматическое подтверждение (Session.AUTO_ACKNOWLEDGE).

Наконец, мы создаем QueueSender для передачи сообщений в объект Queue, извлеченный нами из JNDI.

```
    . . .
            //Создать JMS-объекты
            QueueConnection connection = factory.createQueueConnection();
            QueueSession session =
                connection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
            QueueSender sender = session.createSender(queue);
            . . .
```

QSender: Передача сообщений

Теперь мы готовы передавать сообщения. В данном фрагменте мы вводим цикл, где запрашиваем текст сообщения для передачи. Если пользователь введет *quit*, цикл закончится.

В цикле мы создаем `TextMessage` на основе введенного текста и используем `QueueSender` для передачи сообщения, а затем возвращаемся в начало цикла.

```
    . . .
            //Передать сообщения
            String messageText = null;
            while (true) {
                System.out.println("Enter message to send or 'quit':");
                messageText = reader.readLine();
                if ("quit".equals(messageText))
                    break;
                TextMessage message = session.createTextMessage(messageText);
                sender.send(message);
            }
            . . .
```

QSender: Выход

После завершения цикла мы закрываем `QueueConnection`. Закрытие `QueueConnection` автоматически закрывает`QueueSession` и `QueueSender`.

```
    . . .
            //Выход
            System.out.println("Exiting...");
            reader.close();
            connection.close();
            System.out.println("Goodbye!");

        } catch (Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

QReceiver: Запрос JNDI-имен и поиск администрируемых объектов

Класс `QReceiver`, аналогично классу `QSender`, имеет метод `main(String[])`, который просто создает экземпляр`QReceiver` и вызывает его главный метод `receive()`.

Код запроса JNDI-имен и выполнения поиска администрируемых объектов идентичен коду, использованному в `QSender`.

Однако в этом классе есть два отличия:

- Для указания того, что программа должна завершиться, используется булева переменная экземпляра `stop`.
- `QReceiver` реализует интерфейс `MessageListener`, для того чтобы принимать сообщения в асинхронном режиме.

```
import java.io.*;
import javax.jms.*;
import javax.naming.*;

public class QReceiver implements MessageListener {

    private boolean stop = false;

    public static void main(String[] args) {

        new QReceiver().receive();
    }

    public void receive() {

        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

        try {
            //Запрос JNDI-имен
            System.out.println("Enter QueueConnectionFactory name:");
            String factoryName = reader.readLine();
            System.out.println("Enter Queue name:");
            String queueName = reader.readLine();
            reader.close();

            //Поиск администрируемых объектов
            InitialContext initContext = new InitialContext();
            QueueConnectionFactory factory =
                (QueueConnectionFactory) initContext.lookup(factoryName);
            Queue queue = (Queue) initContext.lookup(queueName);
            initContext.close();
            . . .
```

QReceiver: Создание JMS-объектов

Объекты `QueueConnection` и `QueueSession` создаются так же, как и в `QSender`, а затем создается `QueueReceiver`.

Затем вызывается метод `setMessageListener()` с аргументом `this` - локальным экземпляром `QReceiver`, который реализует интерфейс `MessageListener`.
Наконец, запускается `QueueConnection` для разрешения приема сообщений.

```
    . . .
            //Создать JMS-объекты
            QueueConnection connection = factory.createQueueConnection();
            QueueSession session =
                connection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
            QueueReceiver receiver = session.createReceiver(queue);
            receiver.setMessageListener(this);
            connection.start();
            . . .
```

QReceiver: Подождать остановки и выйти

Затем программа выполняет цикл, выход из которого производится при равенстве переменной `stop` значению `true`. В цикле поток спит в течение одной секунды. После выхода из цикла, QueueConnection закрывается, и программа завершает работу.

```
    . . .
            //Ожидать останова
            while (!stop) {
                Thread.sleep(1000);
            }

            //Выход
            System.out.println("Exiting...");
            connection.close();
            System.out.println("Goodbye!");

        } catch (Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
    . . .
```

QReceiver: Метод onMessage(Message)

Наличие метода `onMessage(Message)` в классе `QReceiver` необходимо, поскольку `QReceiver` реализует интерфейс`MessageListener`.

При приеме сообщения вызывается этот метод с параметром `Message`.
В нашей реализации мы получаем текстовое содержимое сообщения и выводим его в `System.out`. Затем мы проверяем, записано ли в сообщении *stop*, и, если это так, устанавливаем переменную `stop` в значение `true`; это позволит завершить цикл в методе `receive()`.

```
    . . .
    public void onMessage(Message message) {

        try {
            String msgText = ((TextMessage) message).getText();
            System.out.println(msgText);
            if ("stop".equals(msgText))
                stop = true;
        } catch (JMSException e) {
            e.printStackTrace();
            stop = true;
        }
    }
}
```

Интерфейсы "публикация/подписка"

`TopicConnectionFactory`
`TopicConnectionFactory` - это администрируемый объект, извлекаемый из JNDI для создания соединения с провайдером. Он содержит метод `createTopicConnection ()`, который возвращает объект `TopicConnection`.

`TopicConnection`
Объект `TopicConnection` инкапсулирует активное соединение с провайдером. Некоторыми из его методов являются:

- `createTopicSession(boolean, int)`: Возвращает объект `TopicSession`. Параметр `boolean` указывает на то, поддерживает `TopicSession` транзакции или нет; параметр `int` указывает режим подтверждения (см. раздел "Подтверждение").

- `start()` (наследуемый из `Connection`): Активирует доставку сообщений от провайдера.

- `stop()` (наследуемый из `Connection`): Временно останавливает доставку сообщений; доставка может быть активизирована повторно при помощи метода `start()`.

- `close()` (наследуемый из `Connection`): Закрывает соединение с провайдером и освобождает все ресурсы, выделенные для него.

TopicSession
Объект `TopicSession` является однопоточным контекстом для передачи и приема pub/sub-сообщений. Некоторыми из его методов являются:

- `createPublisher(Topic)`: Возвращает объект `TopicPublisher` для передачи сообщений в указанный объект`Topic`.

- `createSubscriber(Topic)`: Возвращает объект `TopicSubscriber` для приема сообщений от указанного объекта`Topic`. Подписчик *не существует постоянно*; то есть, подписка будет продолжаться только на время жизни объекта, и сообщения будут приниматься только при его активности.

- `createDurableSubscriber(Topic, String)`: Возвращает объект `TopicSubscriber` для получения сообщений от указанного объекта `Topic`, передавая `String`-имя подписчику. Сообщения для *долговременного* подписчика будут сохраняться JMS, если объект не активен, и будут доставлены последующим объектам-подписчикам, созданным с этим же именем.

- `unsubscribe(String)`: Заканчивает подписку на `String`-имя.

- `commit()` (наследуемый из Session `Session`): Фиксирует все потребленные или произведенные сообщения для текущей транзакции.

- `rollback()` (наследуемый из Session `Session`): Осуществляет откат для всех потребленных или произведенных сообщений текущей транзакции.

- `create<MessageType>Message(...)` (наследуемый из Session `Session`): Набор методов, возвращающих`<MessageType>Message` - например, `MapMessage`, `TextMessage` и т.д.

Topic
Объект `Topic` инкапсулирует pub/sub-адресат. Он является администрируемым объектом, извлекаемым из JNDI.

TopicPublisher
Объект `TopicPublisher` используется для передачи сообщений "публикация/подписка". Некоторыми из его методов являются:

- `publish(Message)`: Публикует указанный объект `Message`.

- `setDeliveryMode(int)` (наследуемый из `MessageProducer`): Устанавливает режим доставки для последующих передаваемых сообщений; корректными значениями являются `DeliveryMode.PERSISTENT` и`DeliveryMode.NON_PERSISTENT`.

- `setPriority(int)` (наследуемый из `MessageProducer`): Устанавливает приоритет для последующих передаваемых сообщений; корректными значениями являются значения от 0 до 9.

- `setTimeToLive(long)` (наследуемый из `MessageProducer`): Устанавливает продолжительность в миллисекундах последующих передаваемых сообщений, перед тем как они станут не действительными.

TopicSubscriber
Объект `TopicSubscriber` используется для приема сообщений "публикация/подписка". Некоторыми из его методов являются:

- `receive()` (наследуемый из `MessageConsumer`): Возвращает следующее получаемое сообщение; этот метод блокирует работу до тех пор, пока сообщение не станет доступным.

- `receive(long)` (наследуемый из `MessageConsumer`): Принимает следующее сообщение, прибывающее в течение указанных в параметре `long` миллисекунд; этот метод возвращает `null`, если за это время не было принято ни одного сообщения.

- `receiveNoWait` (наследуемый из `MessageConsumer`): Принимает следующее сообщение, если оно доступно в этот же момент времени; этот метод возвращает `null`, если нет доступных сообщений.

- `setMessageListener(MessageListener)` (наследуемый из `MessageConsumer`): Устанавливает`MessageListener`; объект `MessageListener` принимает сообщения по прибытии, то есть, асинхронно.

Программирование систем "публикация/подписка"

То же самое, но по-другому

Мы не будем их рассматривать в пошаговом режиме, как это делали для PTP-программ, потому что кроме типов используемых JMS-интерфейсов они идентичны программам QSender.java и QReceiver.java.
Перед запуском этих программ необходимо установить администрируемые объекты `TopicConnectionFactory` и `Topic`.
Вы увидите отличие между данными программами и PTP-программами после их запуска. Если вы запустите несколько экземпляров `QReceiver`, используя одни и те же `QueueConnectionFactory` и `Queue`, то увидите, что после передачи сообщений из `QSender` только один из экземпляров `QReceiver` принимает каждое переданное сообщение.
Если вы запустите несколько экземпляров `TSubscriber`, вы увидите, что все сообщения, переданные из `TPublisher`, принимаются всеми экземплярами `TSubscriber`.

Ресурсы

Научиться

- Оригинал руководства "Introducing the Java Message Service".
- Спецификация Java Message Service specification, версия 1.1 - наилучший источник информации для подробного изучения JMS.
- Документация по JMS API необходима для JMS-программирования.
- Для корпоративной разработки необходим Java 2 Enterprise Edition. J2EE версии 1.2 требует от совместимых серверов приложений наличия JMS API, но не требует присутствия JMS-провайдера; J2EE версии 1.3 требует от серверов приложений поддержки JMS-провайдера. J2EE версии 1.4 требует от серверов приложений предоставления JMS-провайдера, поддерживающего оба домена.
- Управляемые сообщениями компоненты, часть спецификации EJB 2.0, добавляют в контейнеры Enterprise JavaBeans возможности асинхронного уведомления.
- Статья "Написание JMS-программ с использованием WebSphere MQ v5.3 и WebSphere Studio Application Developer v5", тоже Вилли Фаррела, описывает, как получить, установить и использовать инструментальные средства от IBM для написания JMS-программ. (*developerWorks*, октябрь 2003).
- Бобби Вульф (Bobby Woolf) демонстрирует, как JMS 1.1 упрощает систему обмена сообщениями с использованием унифицированных доменов: (*developerWorks*, август 2002).
- Николас Вайтхед (Nicholas Whitehead) демонстрирует, как сделать ваше JMS-решение работающим с системами любого поставщика, в своей статье "Реализация независимых от поставщика JMS-решений" (*developerWorks*, февраль 2002).
- Суть изменений в JMS 1.1 объясняется в статье Дэвида Карри (David Currie) "Получите сообщение" (*developerWorks*, апрель 2004).
- В зоне developerWorks по Java-технологии вы найдете статьи по любому аспекту Java-программирования.
- На странице руководств в зоне Java-технологии на developerWorks приведен полный список свободно распространяемых руководств по теме Java.

Получить продукты и технологии

- Для работы с данным руководством вам необходимо загрузить пакеты javax.jms и javax.naming.

Приложение

Листинг Sender.java

```
import java.io.*;
import javax.jms.*;
import javax.naming.*;

public class Sender {

    public static void main(String[] args) {

        new Sender().send();
    }

    public void send() {

        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

        try {
            //Запрос JNDI-имен
            System.out.println("Enter ConnectionFactory name:");
            String factoryName = reader.readLine();
            System.out.println("Enter Destination name:");
            String destinationName = reader.readLine();

            //Поиск администрируемых объектов
            InitialContext initContext = new InitialContext();
            ConnectionFactory factory =
                (ConnectionFactory) initContext.lookup(factoryName);
            Destination destination = (Destination) initContext.lookup(destinationName);
            initContext.close();

            //Создание JMS-объектов
            Connection connection = factory.createConnection();
            Session session =
                connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
            MessageProducer sender = session.createProducer(queue);

            //Передача сообщений
            String messageText = null;
            while (true) {
                System.out.println("Enter message to send or 'quit':");
                messageText = reader.readLine();
                if ("quit".equals(messageText))
                    break;
                TextMessage message = session.createTextMessage(messageText);
                sender.send(message);
            }

            //Выход
            System.out.println("Exiting...");
            reader.close();
            connection.close();
            System.out.println("Goodbye!");

        } catch (Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

Листинг Receiver.java

```
import java.io.*;
import javax.jms.*;
import javax.naming.*;

public class Receiver implements MessageListener {

    private boolean stop = false;

    public static void main(String[] args) {

        new Receiver().receive();
    }

    public void receive() {

        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

        try {
            //Запрос JNDI-имен
            System.out.println("Enter ConnectionFactory name:");
            String factoryName = reader.readLine();
            System.out.println("Enter Destination name:");
            String destinationName = reader.readLine();
            reader.close();
```

```
            //Поиск администрируемых объектов
            InitialContext initContext = new InitialContext();
            ConnectionFactory factory =
                (ConnectionFactory) initContext.lookup(factoryName);
            Destination destination = (Destination) initContext.lookup(destinationName);
            initContext.close();

            //Создание JMS-объектов
            Connection connection = factory.createConnection();
            Session session =
                connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
            MessageConsumer receiver = session.createConsumer(queue);
            receiver.setMessageListener(this);
            connection.start();

            //Ожидание останова
            while (!stop) {
                Thread.sleep(1000);
            }

            //Выход
            System.out.println("Exiting...");
            connection.close();
            System.out.println("Goodbye!");

        } catch (Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    public void onMessage(Message message) {

        try {
            String msgText = ((TextMessage) message).getText();
            System.out.println(msgText);
            if ("stop".equals(msgText))
                stop = true;
        } catch (JMSException e) {
            e.printStackTrace();
            stop = true;
        }
    }
}
```

Листинг QSender.java

```
import java.io.*;
import javax.jms.*;
import javax.naming.*;

public class QSender {

    public static void main(String[] args) {

        new QSender().send();
    }

    public void send() {

        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

        try {
            //Запрос JNDI-имен
            System.out.println("Enter QueueConnectionFactory name:");
            String factoryName = reader.readLine();
            System.out.println("Enter Queue name:");
            String queueName = reader.readLine();

            //Поиск администрируемых объектов
            InitialContext initContext = new InitialContext();
            QueueConnectionFactory factory =
                (QueueConnectionFactory) initContext.lookup(factoryName);
            Queue queue = (Queue) initContext.lookup(queueName);
            initContext.close();

            //Создание JMS-объектов
            QueueConnection connection = factory.createQueueConnection();
            QueueSession session =
                connection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
            QueueSender sender = session.createSender(queue);

            //Передача сообщений
            String messageText = null;
            while (true) {
                System.out.println("Enter message to send or 'quit':");
                messageText = reader.readLine();
                if ("quit".equals(messageText))
```

```
                    break;
                TextMessage message = session.createTextMessage(messageText);
                sender.send(message);
            }

            //Выход
            System.out.println("Exiting...");
            reader.close();
            connection.close();
            System.out.println("Goodbye!");

        } catch (Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

Листинг QReceiver.java

```
import java.io.*;
import javax.jms.*;
import javax.naming.*;

public class QReceiver implements MessageListener {

    private boolean stop = false;

    public static void main(String[] args) {

        new QReceiver().receive();
    }

    public void receive() {

        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

        try {
            //Запрос JNDI-имен
            System.out.println("Enter QueueConnectionFactory name:");
            String factoryName = reader.readLine();
            System.out.println("Enter Queue name:");
            String queueName = reader.readLine();
            reader.close();

            //Поиск администрируемых объектов
            InitialContext initContext = new InitialContext();
            QueueConnectionFactory factory =
                (QueueConnectionFactory) initContext.lookup(factoryName);
            Queue queue = (Queue) initContext.lookup(queueName);
            initContext.close();

            //Создание JMS-объектов
            QueueConnection connection = factory.createQueueConnection();
            QueueSession session =
                connection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
            QueueReceiver receiver = session.createReceiver(queue);
            receiver.setMessageListener(this);
            connection.start();

            //Ожидание останова
            while (!stop) {
                Thread.sleep(1000);
            }

            //Выход
            System.out.println("Exiting...");
            connection.close();
            System.out.println("Goodbye!");

        } catch (Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    public void onMessage(Message message) {

        try {
            String msgText = ((TextMessage) message).getText();
            System.out.println(msgText);
            if ("stop".equals(msgText))
                stop = true;
        } catch (JMSException e) {
            e.printStackTrace();
            stop = true;
        }
    }
}
```

Листинг TPublisher.java

```
import java.io.*;
import javax.jms.*;
import javax.naming.*;

public class TPublisher {

    public static void main(String[] args) {

        new TPublisher().publish();
    }

    public void publish() {

        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

        try {
            //Запрос JNDI-имен
            System.out.println("Enter TopicConnectionFactory name:");
            String factoryName = reader.readLine();
            System.out.println("Enter Topic name:");
            String topicName = reader.readLine();

            //Поиск администрируемых объектов
            InitialContext initContext = new InitialContext();
            TopicConnectionFactory factory =
                (TopicConnectionFactory) initContext.lookup(factoryName);
            Topic topic = (Topic) initContext.lookup(topicName);
            initContext.close();

            //Создание JMS-объектов
            TopicConnection connection = factory.createTopicConnection();
            TopicSession session =
                connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
            TopicPublisher publisher = session.createPublisher(topic);

            //Передача сообщений
            String messageText = null;
            while (true) {
                System.out.println("Enter message to send or 'quit':");
                messageText = reader.readLine();
                if ("quit".equals(messageText))
                    break;
                TextMessage message = session.createTextMessage(messageText);
                publisher.publish(message);
            }

            //Выход
            System.out.println("Exiting...");
            reader.close();
            connection.close();
            System.out.println("Goodbye!");

        } catch (Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

Листинг TSubscriber.java

```
import java.io.*;
import javax.jms.*;
import javax.naming.*;

public class TSubscriber implements MessageListener {

    private boolean stop = false;

    public static void main(String[] args) {

        new TSubscriber().subscribe();
    }

    public void subscribe() {

        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

        try {
            //Запрос JNDI-имен
            System.out.println("Enter TopicConnectionFactory name:");
            String factoryName = reader.readLine();
            System.out.println("Enter Topic name:");
            String topicName = reader.readLine();
            reader.close();

            //Поиск администрируемых объектов
```

```
            InitialContext initContext = new InitialContext();
            TopicConnectionFactory factory =
                (TopicConnectionFactory) initContext.lookup(factoryName);
            Topic topic = (Topic) initContext.lookup(topicName);
            initContext.close();

            //Создание JMS-объектов
            TopicConnection connection = factory.createTopicConnection();
            TopicSession session =
                connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
            TopicSubscriber subscriber = session.createSubscriber(topic);
            subscriber.setMessageListener(this);
            connection.start();

            //Ожидание останова
            while (!stop) {
                Thread.sleep(1000);
            }

            //Выход
            System.out.println("Exiting...");
            connection.close();
            System.out.println("Goodbye!");

        } catch (Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    public void onMessage(Message message) {

        try {
            String msgText = ((TextMessage) message).getText();
            System.out.println(msgText);
            if ("stop".equals(msgText))
                stop = true;
        } catch (JMSException e) {
            e.printStackTrace();
            stop = true;
        }
    }
}
```

Темы заданий для разработки систем на основе JMS:

1. Чат.
2. Система организации взаимодействия склад – магазины
3. Система организации взаимодействия магазин - клиенты

Для примера смотреть http://oreilly.com/catalog/javmesser/chapter/ch02.html