13

Компоненты, управляемые сообщениями

Эта глава разбита на два подраздела: «JMS как средство» и «Компоненты, управляемые сообщениями». Первый раздел описывает службу сообщений Java (Java Message Service – JMS) и ее роль в качестве ресурса, доступного всем типам компонентов (сеансовым, объектным и управляемым сообщениями). Читателям, не знакомым с JMS, следует прочитать первый раздел и только потом переходить ко второму.

Те же, для кого JMS не является тайной за семью печатями, могут перейти прямо ко второму разделу, содержащему краткий обзор нового типа компонента – компонента, управляемого сообщениями. Компонент, управляемый сообщениями, – это асинхронный компонент, активируемый при получении сообщения. В EJB 2.0 производители обязаны поддерживать компоненты, управляемые сообщениями на базе JMS, принимающие JMS-сообщения из отдельных тем (topics) или очередей (queues) и обрабатывающие эти сообщения по мере их поступления.

Все производители EJB 2.0 должны по умолчанию поддерживать провайдер JMS. Большая часть производителей EJB 2.0 имеют встроенный провайдер JMS, а некоторые кроме него могут поддерживать и других провайдеров JMS. Независимо от того, каким образом производитель EJB 2.0 предоставляет службу JMS, она является обязательной, если производитель предполагает поддерживать компоненты, управляемые сообщениями. Преимущество этого принудительного включения JMS состоит в том, что разработчики EJB могут рассчитывать на присутствие работоспособного провайдера JMS, с помощью которого можно как отправлять, так и передавать сообщения.

JMS как средство

JMS – это стандартный, независимый от производителя программный интерфейс (API), который является частью платформы J2EE и может применяться для доступа к корпоративным системам передачи сообщений. Корпоративные системы передачи сообщений (ориентируемые на сообщения программные продукты) дают возможность обмениваться сообщениями между приложениями через сеть. JMS во многом схож с JDBC: так же, как JDBC является программным интерфейсом, который может служить для доступа ко многим различным реляционным базам данных, JMS предоставляет такой же независимый от производителей доступ к корпоративным системам передачи сообщений. Многие программы передачи сообщений на уровне предприятия в настоящее время поддерживают JMS, включая такие, как MQSeries от IBM, службу JMS WebLogic от BEA, iPlanet Message Queue от Sun Microsystems, SonicMQ от Progress и другие. Приложения, в которых посылка или прием сообщений основываются на JMS API, переносимы между разными марками JMS-продуктов.

Прикладные программы Java, использующие JMS, называются *клиентами JMS* (*JMS client*), а система передачи сообщений, управляющая маршрутизацией и доставкой сообщений, называется *провайдером JMS* (*JMS provider*). *Приложение JMS* (*JMS application*) – это прикладная система, состоящая из нескольких JMS-клиентов и обычно одного JMS-провайдера.

JMS-клиент, посылающий сообщение, называется *поставщиком* (*producer*), а JMS-клиент, принимающий сообщение, называется *потребителем* (*consumer*). Один JMS-клиент может быть одновременно и поставщиком и потребителем. Когда мы употребляем термины «потребитель» и «поставщик», мы подразумеваем JMS-клиент, который соответственно принимает или передает сообщения.

В EJB компоненты всех типов могут использовать JMS для передачи сообщений различным адресатам. Эти сообщения принимаются другими прикладными программами Java или компонентами, управляемыми сообщениями. JMS обеспечивает передачу сообщений от компонентов с помощью службы передачи сообщений, иногда называемой посредником сообщения или маршрутизатором. Посредники сообщений существуют уже пару десятилетий (самым старым и наиболее устоявшимся является MQSeries от IBM), но сама JMS достаточно нова и специально предназначена для передачи различных типов сообщений от одного приложения Java другому.

Новая реализация компонента TravelAgent с использованием JMS

Мы можем изменить компонент TravelAgent, разработанный в главе 12, так, чтобы он использовал JMS для уведомления некоторого другого

приложения Java о том, что был сделан заказ билетов. Приведенный ниже код показывает, как следует изменить метод bookPassage(), чтобы компонент TravelAgent посылал простое текстовое сообщение, основанное на описании объекта TicketDO:

```
public TicketDO bookPassage(CreditCardDO card, double price)
    throws IncompleteConversationalState {

    if (customer == null || cruise == null || cabin == null) {
        throw new IncompleteConversationalState();
    }
    try {
        ReservationHomeLocal resHome = (ReservationHomeLocal)
            jndiContext.lookup("java:comp/env/ejb/ReservationHomeLocal");

        ReservationLocal reservation =
            resHome.create(customer, cruise, cabin, price, new Date());

        Object ref = jndiContext.lookup
            ("java:comp/env/ejb/ProcessPaymentHomeRemote");

        ProcessPaymentHomeRemote ppHome = (ProcessPaymentHomeRemote)
           PortableRemoteObject.narrow(ref, ProcessPaymentHomeRemote.class);

        ProcessPaymentRemote process = ppHome.create();
        process.byCredit(customer, card, price);

        TicketDO ticket = new TicketDO(customer,cruise,cabin,price);

        String ticketDescription = ticket.toString();

        TopicConnectionFactory factory = (TopicConnectionFactory)
            jndiContext.lookup("java:comp/env/jms/TopicFactory");

        Topic topic = (Topic)
            jndiContext.lookup("java:comp/env/jms/TicketTopic");

        TopicConnection connect = factory.createTopicConnection();

        TopicSession session = connect.createTopicSession(true,0);

        TopicPublisher publisher = session.createPublisher(topic);

        TextMessage textMsg = session.createTextMessage();
        textMsg.setText(ticketDescription);
        publisher.publish(textMsg);
        connect.close();

        return ticket;
    } catch(Exception e) {
        throw new EJBException(e);
    }
}
```

Посылка сообщения потребовала добавления довольно большого объема нового кода. Несмотря на этот поначалу пугающий факт, основы JMS не так уж и сложны.

TopicConnectionFactory и Topic

Для того чтобы посылать сообщение JMS, нам необходимо соединение с провайдером JMS и адрес назначения сообщения. Подключение к провайдеру JMS может быть выполнено через фабрику подключений JMS, а адрес места назначения сообщения определяется объектом `Topic` (тема). И фабрика подключений и объект `Topic` извлекаются из JNDI ENC компонента TravelAgent:

```
TopicConnectionFactory factory = (TopicConnectionFactory)
    jndiContext.lookup("java:comp/env/jms/TopicFactory");

Topic topic = (Topic)
    jndiContext.lookup("java:comp/env/jms/TicketTopic");
```

Фабрика `TopicConnectionFactory` в JMS по своим функциям похожа на `DataSource` в JDBC. Так же как `DataSource` предоставляет JDBC-подключение к базе данных, `TopicConnectionFactory` предоставляет JMS-подключение к маршрутизатору сообщений.[1]

Сам объект `Topic` представляет независимый от сетевой реализации пункт назначения, в который будет отправлено сообщение. В JMS сообщения посылаются в пункт назначения (тему или очередь), а не другим прикладным программам непосредственно. Тема представляет собой аналог списка рассылки или телеконференции: любое приложение, обладающее достаточными правами, может принимать сообщения из темы и посылать сообщения в тему. Когда клиент JMS принимает сообщения из темы, говорят, что клиент *подписан (subscribed)* на данную тему. JMS разделяет прикладные программы, позволяя им посылать сообщения друг другу через пункт назначения, который работает в качестве виртуального канала.

JMS также поддерживает другой тип пункта назначения, называемый *очередью (queue)*. Различия между темами и очередями более подробно объяснены ниже.

TopicConnection и TopicSession

`TopicConnectionFactory` предназначена для создания `TopicConnection`, который является текущим подключением к провайдеру JMS:

```
TopicConnection connect = factory.createTopicConnection();

TopicSession session = connect.createTopicSession(true,0);
```

После того как `TopicConnection` получен, он может использоваться для создания `TopicSession`. `TopicSession` позволяет Java-программисту сгруппировать операции посылки и приема сообщений. В нашем случае нам

[1] Аналогия не совсем точная. Можно было бы сказать, что `TopicSession` являются аналогом `DataSource`, т. к. они оба представляют соединения с ресурсами транзакций.

понадобится лишь один TopicSession. Однако часто полезно иметь несколько объектов TopicSession: если требуется создавать и получать сообщения с применением многопоточности, каждым потоком должен быть создан отдельный объект Session, относящийся к данному потоку. Это необходимо ввиду того, что объекты JMS Session основываются на однопоточной модели, запрещающей параллельный доступ нескольких потоков к одному объекту Session. Поток, создавший TopicSession, – это обычно поток, который использует производителей и потребителей данного объекта Session (т. е. объекты TopicPublisher и TopicSubscriber). Если необходимо создавать и получать сообщения на основе многопоточности, каждым потоком должен быть создан и использован отдельный объект Session.

Метод createTopicSession() имеет два параметра:

```
createTopicSession(boolean transacted, int acknowledgeMode)
```

В соответствии со спецификацией EJB 2.0 эти параметры игнорируются во время выполнения, т. к. контейнер EJB управляет режимами транзакций и подтверждений каждого ресурса JMS, полученного через JNDI ENC. Спецификация рекомендует, чтобы разработчики использовали параметры true для transacted и 0 для acknowlegeMode, но поскольку они, по определению, игнорируются, не имеет значения, что вы будете использовать. К сожалению, не все производители твердо придерживаются этой части спецификации. Одни производители игнорируют данные параметры, а другие – нет. Ознакомьтесь с документацией от производителя, чтобы определить правильные значения для этих параметров и в управляемых контейнером, и в управляемых компонентом транзакциях.

Хорошим стилем программирования является закрытие TopicConnection по окончании работы с ним, т. к. это позволяет сберечь ресурсы:

```
TopicConnection connect = factory.createTopicConnection();
...
connect.close();
```

TopicPublisher

TopicSession применяется для создания TopicPublisher, посылающего сообщения из компонента TravelAgent в пункт назначения, указанный объектом Topic. Каждый клиент JMS, подписавшийся на данную тему, получит копию сообщения:

```
TopicPublisher publisher = session.createPublisher(topic);

TextMessage textMsg = session.createTextMessage();
textMsg.setText(ticketDescription);
publisher.publish(textMsg);
```

Типы сообщений

Сообщение в JMS – это объект Java, состоящий из двух частей: заголовка и тела сообщения. В заголовке находится информация о доставке и метаданные, а тело сообщения заключает в себе данные прикладной программы, которые могут принимать разные формы: текстовые, сериализуемые объекты, байтовые потоки и т. д. JMS API определяет несколько типов сообщений (TextMessage, MapMessage, ObjectMessage и другие) и предоставляет методы для доставки и приема сообщений от других прикладных программ.

Например, мы можем модифицировать компонент TravelAgent так, чтобы он вместо TextMessage посылал MapMessage:

```
TicketDO ticket = new TicketDO(customer,cruise,cabin,price);
...
TopicPublisher publisher = session.createPublisher(topic);

MapMessage mapMsg = session.createTextMessage();
mapMsg.setInt("CustomerID", ticket.customerID.intValue());
mapMsg.setInt("CruiseID", ticket.cruiseID.intValue());
mapMsg.setInt("CabinID", ticket.cabinID.intValue());
mapMsg.setDouble("Price", ticket.price);

publisher.publish(mapMsg);
```

JMS-клиенты, которые примут MapMessage, могут обращаться к его атрибутам (CustomerID, CruiseID, CabinID и Price) по имени.

В качестве альтернативы компонент TravelAgent мог бы быть изменен так, чтобы использовать тип ObjectMessage, который позволит нам посылать в виде сообщения весь объект TicketDO, с применением сериализации Java:

```
TicketDO ticket = new TicketDO(customer,cruise,cabin,price);
...
TopicPublisher publisher = session.createPublisher(topic);

ObjectMessage objectMsg = session.createObjectMessage();
ObjectMsg.setObject(ticket);

publisher.publish(objectMsg);
```

Кроме TextMessage, MapMessage и ObjectMessage JMS предоставляет еще два типа сообщений: StreamMessage и BytesMessage. Первый из них – StreamMessage – может нести в качестве полезной нагрузки содержимое потока ввода-вывода. А BytesMessage может принимать любой массив байт, которые он воспринимает как необрабатываемые данные.

XML-дескриптор развертывания

Применяя JMS, это средство необходимо объявить в XML-дескрипторе развертывания компонента, таким же образом, как и ресурс JDBC, используемый компонентом Ship в главе 10:

```
<enterprise-beans>
    <session>
        <ejb-name>TravelAgentBean</ejb-name>
        ...
        <resource-ref>
            <res-ref-name>jms/TopicFactory</res-ref-name>
            <res-type>javax.jms.TopicConnectionFactory</res-type>
            <res-auth>Container</res-auth>
        </resource-ref>
        <resource-ref>
            <res-ref-name>jdbc/titanDB</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Container</res-auth>
        </resource-ref>
        <resource-env-ref>
            <resource-env-ref-name>jms/TicketTopic</resource-env-ref-name>
            <resource-env-ref-type>javax.jms.Topic</resource-env-ref-type>
        </resource-env-ref>
        ...
    </session>
```

Элемент <resource-ref> для JMS TopicConnectionFactory похож на объявление <resource-ref> для JDBC DataSource: он объявляет имя JNDI ENC, тип интерфейса и протокол авторизации. Кроме <resource-ref> компонент TravelAgent также должен объявить элемент <resource-env-ref>, в котором перечислены все «управляемые объекты», связанные с элементом <resource-ref>. В данном случае мы объявляем Topic, используемый для отправки сообщения о билете. Во время развертывания JMS TopicConnectionFactory и Topic, объявленные в элементах <resource-ref> и <resource-env-ref>, необходимо связать с JMS-фабрикой и темой.

Прикладной клиент JMS

Для того чтобы лучше понять механизм применения JMS, мы создадим приложение Java, единственным назначением которого будет прием и обработка сообщений о заказах билетов. Разработаем очень простой клиент JMS, просто выводящий описание по каждому билету по мере поступления сообщений. Допустим, что компонент TravelAgent для посылки описания билета клиентам JMS использует TextMessage. Следующий код показывает, как могло бы выглядеть клиентское приложение JMS:

```
import javax.jms.Message;
import javax.jms.TextMessage;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicConnection;
import javax.jms.TopicSession;
import javax.jms.Topic;
import javax.jms.Session;
import javax.jms.TopicSubscriber;
```

```
import javax.jms.JMSException;
import javax.naming.InitialContext;
public class JmsClient_1 implements javax.jms.MessageListener {

    public static void main(String [] args) throws Exception {

        if(args.length != 2)
            throw new Exception("Недопустимое количество аргументов");

        new JmsClient_1(args[0], args[1]);

        while(true){Thread.sleep(10000);}

    }

   public JmsClient_1(String factoryName, String topicName) throws Exception
{

        InitialContext jndiContext = getInitialContext();

        TopicConnectionFactory factory = (TopicConnectionFactory)
            jndiContext.lookup("ИмяФабрикиТемы");

        Topic topic = (Topic)jndiContext.lookup("ИмяТемы");

        TopicConnection connect = factory.createTopicConnection();

        TopicSession session =
            connect.createTopicSession(false,Session.AUTO_ACKNOWLEDGE);

        TopicSubscriber subscriber = session.createSubscriber(topic);

        subscriber.setMessageListener(this);

        connect.start();
    }

    public void onMessage(Message message) {
        try {

            TextMessage textMsg = (TextMessage)message;
            String text = textMsg.getText();
            System.out.println("\n RESERVATION RECIEVED:\n"+text);

        } catch(JMSException jmsE) {
            jmsE.printStackTrace();
        }
    }

    public static InitialContext getInitialContext() {
        // создаем контекст JNDI, специфичный для производителя
    }
}
```

Конструктор JmsClient_1 получает TopicConnectionFactory и Topic из InitialContext JNDI. Этот контекст создается со специфичными для производителя свойствами так, чтобы клиент мог соединиться с тем же

провайдером JMS, который используется компонентом TravelAgent. Например, метод getInitialContext() для сервера приложений WebLogic будет выглядеть следующим образом:[1]

```
public static InitialContext getInitialContext() {
    Properties env = new Properties();
    env.put(Context.SECURITY_PRINCIPAL, "guest");
    env.put(Context.SECURITY_CREDENTIALS, "guest");
    env.put(Context.INITIAL_CONTEXT_FACTORY,
       "weblogic.jndi.WLInitialContextFactory");
    env.put(Context.PROVIDER_URL, "t3://localhost:7001");
    return new InitialContext(env);
}
```

Клиент, получив TopicConnectionFactory и Topic, создает TopicConnection и TopicSession таким же образом, как и компонент TravelAgent. Главное отличие состоит в том, что объект TopicSession применяется для создания TopicSubscriber вместо TopicPublisher. TopicSubscriber специально предназначен для обработки входящих сообщений, созданных для указанной в нем темы:

```
TopicSession session =
    connect.createTopicSession(false,Session.AUTO_ACKNOWLEDGE);

TopicSubscriber subscriber = session.createSubscriber(topic);

subscriber.setMessageListener(this);

connect.start();
```

TopicSubscriber может принимать сообщения непосредственно или перенаправлять обработку сообщений объекту javax.jms.MessageListener. Мы решили, что JmsClient_1 будет реализовывать интерфейс MessageListener так, чтобы он мог обрабатывать сообщения самостоятельно. Объекты MessageListener реализуют единственный метод – onMessage(), вызываемый каждый раз, когда в тему подписчика посылается новое сообщение. В нашем случае каждый раз, когда компонент TravelAgent посылает сообщение о резервировании билетов в тему, у клиента JMS будет вызываться его метод onMessage() так, чтобы он мог принять копию сообщения и обработать его:

```
public void onMessage(Message message) {
    try {
        TextMessage textMsg = (TextMessage)message;
        String text = textMsg.getText();
        System.out.println("\n Получена информация о резервации:\n"+text);
```

[1] Кроме этого JNDI позволяет устанавливать свойства в файле *jndi.properties*, содержащем значения свойств для InitialContext, и может быть обработан динамически во время выполнения. В этой книге свойство устанавливается явно.

```
        } catch(JMSException jmsE) {
            jmsE.printStackTrace();
        }
    }
```

JMS – асинхронная система

Одно из основных преимуществ системы сообщений JMS состоит в том, что она является *асинхронной* (*asynchronous*). Другими словами, клиент JMS может посылать сообщение без необходимости ждать ответ. Сравните эту гибкость с системой синхронных сообщений Java RMI. RMI – это превосходный выбор для сборки транзакционных компонентов, но для некоторых применений ее возможности слишком ограничены. Клиент, вызывая метод компонента, каждый раз блокирует текущий поток, пока метод не завершит выполнение. Эта обработка с блокировкой делает клиент зависимым от готовности сервера EJB и приводит к тесной связи между клиентом и компонентом.

В JMS клиент асинхронно посылает сообщения в пункт назначения (тему или очередь), из которого другие клиенты JMS одновременно могут принимать сообщения. Когда клиент JMS посылает сообщение, то не ждет ответа. Он посылает сообщение маршрутизатору, отвечающему за доставку его другим клиентам. Клиенты, посылающие сообщения, отделены от клиентов, принимающих их. Отправители не зависят от готовности получателей.

Ограничения RMI делают JMS привлекательной альтернативой для связи с другими приложениями. Используя стандартный контекст имен окружения JNDI, компонент может получать JMS-соединение с провайдером JMS и с его помощью организовывать доставку асинхронных сообщений другим приложениям Java. Например, сеансовый компонент TravelAgent может средствами JMS уведомлять другие приложения об окончании обработки заказа, как показано на рис. 13.1.

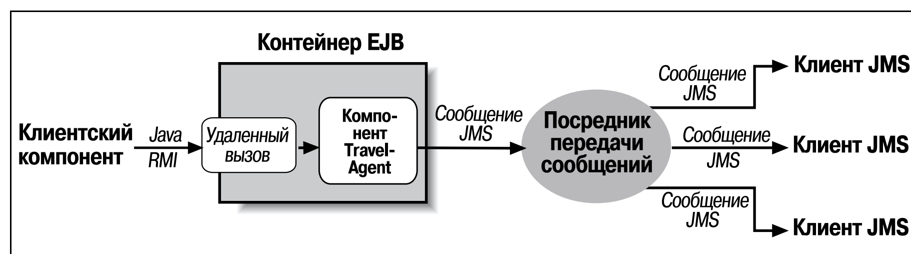

Рис. 13.1. Использование JMS в компоненте TravelAgent

В данном случае приложения, принимающие сообщения JMS от компонента TravelAgent, могут быть компонентами, управляемыми со-

общениями, другими корпоративными приложениями Java или приложениями других организаций, которым нужна информация о том, что заказ был обработан. Примеры могли бы включать и деловых партнеров, одновременно использующих информацию о клиенте, и внутреннее маркетинговое приложение, заносящее клиентов в список рассылки.

JMS дает возможность компоненту послать сообщения без блокировки. Компонент не знает, кто примет сообщение, из-за того, что он передает сообщение виртуальному каналу (пункту назначения), а не другому приложению непосредственно. Прикладные программы могут выбрать получение сообщения из этого виртуального канала и принимать уведомления о новых заказах билетов.

Один интересный аспект корпоративной системы сообщений состоит в том, что развязанный асинхронный характер этой технологии означает, что контексты транзакций и безопасности отправителя не наследуются получателем сообщения. Например, если компонент TravelAgent посылает сообщение о билете, оно может быть авторизовано провайдером JMS, но его контекст безопасности не будет передан клиенту JMS, принимающему сообщение. Когда клиент JMS будет принимать сообщение от компонента TravelAgent, он не будет иметь никакого представления о контексте безопасности, под которым данное сообщение было послано. Так и должно быть, потому что отправитель и получатель часто работают в разных окружениях с разными зонами безопасности.

Точно так же транзакции никогда не передаются от отправителя получателю. С одной стороны, отправитель понятия не имеет о том, кто получает его сообщения. Если сообщение посылается в тему, у него может быть один или тысячи получателей. Управление распределенной транзакцией в таких неопределенных ситуациях недостаточно надежно. Кроме того, клиенты, принимающие сообщение, могут не получить его в течение долгого времени после того, как оно было послано, в связи с тем, что они временно были отключены или по другой причине неспособны были принимать сообщения. Одно из основных преимуществ JMS состоит в том, что он допускает временное рассоединение отправителей и получателей. Транзакции предназначены для быстрого выполнения, поскольку они блокируют ресурсы. Длинная транзакция с непредсказуемым завершением также является ненадежной.

Клиент JMS, однако, может иметь распределенную транзакцию с провайдером JMS так, чтобы управлять операциями посылки и получения в контексте транзакции. Например, если транзакция компонента TravelAgent прервется по какой-либо причине, провайдер JMS исключит сообщение о билете, посланное компонентом TravelAgent. Транзакции и JMS более подробно рассматриваются в главе 14.

Модели передачи сообщений JMS: «издание-подписка» и «точка-точка»

JMS предоставляет два типа моделей передачи сообщений: *«издание-подписка»* (*publish-and-subscribe*) и *«точка-точка»* (*point-to-point*). Спецификация JMS называет их *зонами сообщений* (*messaging domains*). В терминологии JMS «издание-подписка» и «точка-точка» часто сокращаются до pub/sub и p2p (или PTP) соответственно. В данной главе используются и длинные и короткие формы записи.

В двух словах, «издание-подписка» предназначена для передачи сообщений типа «один-ко-многим», тогда как модель «точка-точка» предназначена для рассылки сообщений «один-к-одному» (рис. 13.2).

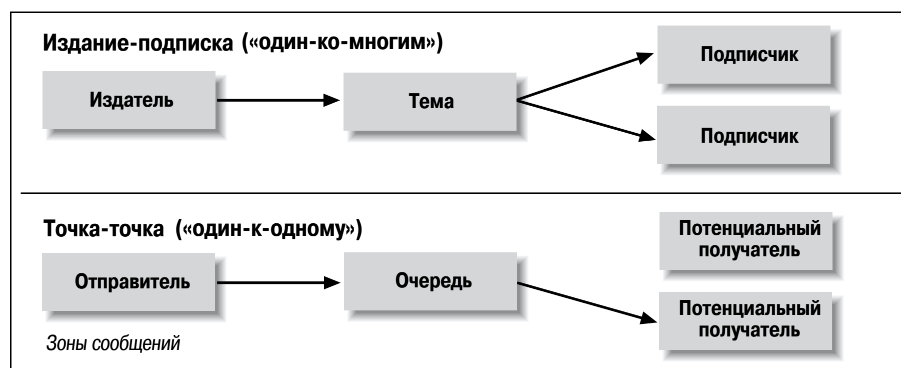

Рис. 13.2. Зоны сообщений JMS

«Издание-подписка»

При передаче сообщений «издание-подписка» один поставщик может посылать сообщение многим 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 pub/sub, по существу, представляет собой модель, основанную на проталкивании (push-based), где сообщения автоматически распространяются по потребителям без необходимости запрашивать, или вытаскивать, новые сообщения из темы.

В модели передачи сообщений pub/sub производитель, посылающий сообщение, не зависит от потребителей, принимающих сообщение. Дополнительно клиенты JMS, использующие pub/sub, могут устанавливать долговременные подписки, позволяющие потребителям отсоединиться и позже снова подключиться и забрать сообщения, поступившие во время отключения связи.

Компонент TravelAgent в данной главе основывается на программной модели pub/sub с объектом `Topic` в качестве пункта назначения.

«Точка-точка»

Модель передачи сообщений «точка-точка» позволяет клиентам JMS посылать и принимать сообщения как синхронно, так и асинхронно, через виртуальные каналы, известные как *очереди* (*queues*). Модель передачи сообщений p2p традиционно основывается на модели вытягивания (pull), или опроса (poll), в которой сообщения запрашиваются из очереди вместо автоматического выталкивания клиенту.[1]

У очереди может быть несколько получателей, но только один получатель может принимать каждое отдельное сообщение. Как было показано выше на рис. 13.2, провайдер JMS заботится о раздаче сообщений клиентам JMS, гарантируя, что каждое сообщение будет использовано только одним клиентом. Спецификация JMS не задает правила распределения сообщений среди нескольких получателей.

Программный интерфейс службы сообщений для p2p похож на используемый для pub/sub. Следующий фрагмент кода содержит листинг, иллюстрирующий, как компонент TravelAgent мог бы быть изменен для использования основанного на очереди API для p2p вместо основанной на теме модели pub/sub, фигурирующей в приведенном ранее примере:

```
public TicketDO bookPassage(CreditCardDO card, double price)
    throws IncompleteConversationalState {
        ...
        TicketDO ticket = new TicketDO(customer,cruise,cabin,price);


        String ticketDescription = ticket.toString();

        QueueConnectionFactory factory = (QueueConnectionFactory)
            jndiContext.lookup("java:comp/env/jms/QueueFactory");

        Queue queue = (Queue)
            jndiContext.lookup("java:comp/env/jms/TicketQueue");

        QueueConnection connect = factory.createQueueConneciton();

        QueueSession session = connect.createQueueSession(true,0);

        QueueSender sender = session.createSender(queue);

        TextMessage textMsg = session.createTextMessage();
        textMsg.setText(ticketDescription);
        sender.send(textMsg);
        connect.close();
```

[1] В спецификации JMS не указывается определенно, как должны быть реализованы модели p2p и pub/sub. Каждая из них может использовать и проталкивание и опрос, но, по крайней мере концептуально, модель pub/sub основана на проталкивании, а модель p2p – на опросе.

```
            return ticket;
        } catch(Exception e) {
            throw new EJBException(e);
        }
    }
```

Какую модель сообщений предпочесть?

Причины существования двух моделей объясняются происхождением спецификации JMS. JMS начиналась как способ обеспечения общего API для доступа к существующим системам передачи сообщений. Во время его планирования одни производители придерживались модели p2p передачи сообщений, а другие – модели pub/sub. Следовательно, для получения широкой промышленной поддержки в JMS требовалось предоставить API для обеих моделей. Спецификация JMS 1.0.2 не требовала, чтобы провайдер JMS поддерживал обе модели. Однако от производителя EJB 2.0 требуется поддержка обеих моделей передачи сообщений.

Почти все, что может быть сделано при помощи модели pub/sub, можно сделать и с помощью «точка-точка», и наоборот. Аналогия может быть проведена с предпочтением разработчиками различных языков программирования. Теоретически любое приложение, которое может быть написано на Паскале, может также быть написано и на C. Все что может быть написано на C++, может также быть написано и на Java. В некоторых случаях все сводится к вопросу о предпочтениях или к тому, с какой моделью вы уже знакомы.

В большинстве случаев решение о том, какую модель предпочесть, зависит от различных достоинств каждой модели. В случае с pub/sub тему может прослушивать любое число подписчиков, и все они будут принимать копии одних и тех же сообщений. Поставщик не заботится о том, слушает ли кто-нибудь его. Например, рассмотрим издателя, который рассылает биржевые котировки. Если какой-либо отдельный подписчик в данное время не подсоединен и пропустил выгодную цену, издателя это не касается. И наоборот, сессия «точка-точка», вероятно, будет предназначена для диалога «один-к-одному» с отдельным приложением, находящемся на другом конце. В этом сценарии каждое сообщение действительно важно.

Диапазон и разнообразие данных, представляемых сообщениями, также могут сыграть свою роль. В случае применения pub/sub сообщения распределяются по потребителям, основываясь на фильтрации, которая задается использованием определенных тем. Даже если сообщения применяются для установки связи «один-к-одному» с другим известным приложением, может быть выгодно использовать pub/sub с несколькими темами для выделения разных видов сообщений. Каждый вид сообщения может обрабатываться отдельно своим собственным уникальным потребителем и слушателем `onMessage()`.

Модель «точка-точка» более удобна, если требуется, чтобы отдельный получатель обработал данное сообщение только один раз. Возможно, это наиболее существенное различие между двумя моделями: p2p гарантирует, что только один потребитель обработает каждое сообщение. Это чрезвычайно важно, когда сообщения должны быть обработаны отдельно и последовательно друг за другом.

Объектные и сеансовые компоненты не должны принимать сообщения

JmsClient_1 был предназначен для получения сообщений, производимых компонентом TravelAgent. Может ли другой объектный или сеансовый компонент также принимать эти сообщения? Ответ – да, но это – очень плохая мысль.

Компоненты с данными и сеансовые компоненты отвечают на вызовы Java RMI от компонентных клиентов и не могут быть запрограммированы, чтобы отвечать на сообщения JMS так же, как это делают компоненты, управляемые сообщениями. Это означает, что невозможно создать сеансовый или объектный компонент, который будет управляться входящими сообщениями. Неспособность делать компонент, отвечающий на сообщения JMS, послужило причиной того, что в EJB 2.0 были введены компоненты, управляемые сообщениями. Компоненты, управляемые сообщениями, предназначены для получения сообщений из тем и очередей. Они заполнили важную нишу, и в следующем разделе мы подробно изучим, как их программировать.

Можно разработать компонент с данными или сеансовый компонент, который может получать сообщение JMS из прикладного метода, но сначала этот метод должен быть вызван клиентом EJB. Например, вызываемый прикладной метод компонента Hypothetical устанавливает сессию JMS и затем пытается считать сообщение из очереди:

```
public class HypotheticalBean implements javax.ejb.SessionBean {
    InitialContext jndiContext;

    public String businessMethod() {

        try{

            QueueConnectionFactory factory = (QueueConnectionFactory)
                jndiContext.lookup("java:comp/env/jms/QueueFactory");

            Queue topic = (Queue)
                jndiContext.lookup("java:comp/env/jms/Queue");

            QueueConnection connect = factory.createQueueConneciton();
            QueueSession session = connect.createQueueSession(true,0);
            QueueReceiver receiver = session.createReceiver(queue);

            TextMessage textMsg = (TextMessage)receiver.receive();
```

```
    connect.close();

            return textMsg.getText();

        } catch(Exception e) {
            throws new EJBException(e);
        }

    }
    ...
}
```

Объект QueueReceiver, являющийся потребителем сообщений, использует упреждающее чтение сообщения из очереди. Хотя все было запрограммировано правильно, это – достаточно опасная операция, т. к. вызов метода QueueReceiver.receive() блокирует поток, пока сообщение не станет доступным. Если сообщение никогда не поступит в очередь получателя, поток будет блокирован на неопределенное время! Другими словами, если никто никогда не пошлет сообщение в очередь, QueueReceiver будет просто сидеть там в вечном ожидании.

По правде говоря, существуют также и другие методы receive(), которые являются менее опасными. Например, receive(long timeout) позволяет указать время, после которого QueueReceiver должен снять блокировку потока и прекратить ожидание сообщения. Также существует метод receiveNoWait(), проверяющий наличие сообщений и возвращающий null, если нет ни одного ожидающего сообщения, таким образом избегая длительной блокировки потока.

Хотя альтернативные методы receive() гораздо безопаснее, применение их все же рискованно. Нет никакой гарантии, что они будут выполняться, как мы ожидаем, и риск ошибки программиста (например, использование неверного метода receive()) слишком велик. Кроме этого, компонент, управляемый сообщениями, предоставляет мощный и простой компонент, специально предназначенный для получения сообщений JMS. В этой книге не рекомендуется пытаться получать сообщения в объектных или сеансовых компонентах.

Дополнительная информация о JMS

JMS (и корпоративные системы сообщений вообще) представляет собой мощную парадигму распределенных вычислений. По моему мнению, служба сообщений Java так же важна, как и Enterprise JavaBeans, и должна быть хорошо изучена перед применением ее в процессе разработки.

В этой главе приведен краткий обзор JMS, и она содержит лишь самый необходимый материал, призванный подготовить читателя к обсуждению компонентов, управляемых сообщениями, в следующем разделе. JMS имеет множество особенностей и деталей, которые просто слишком обширны, чтобы рассмотреть их все в этой книге. Чтобы понять

JMS и особенности ее применения, необходимо изучать ее самостоятельно.[1] Время, потраченное на изучение JMS, окупится с лихвой.

Компоненты, управляемые сообщениями

Компоненты, управляемые сообщениями (Message-Driven Beans, MDB), – это не имеющие состояния, серверные компоненты для обработки асинхронных сообщений JMS, поддерживающие транзакции. Впервые представленные в EJB 2.0, компоненты, управляемые сообщениями, обрабатывают сообщения, распространяемые через службу сообщений Java (Java Message Service).

Компоненты, управляемые сообщениями, могут принимать сообщения JMS и обрабатывать их. Компонент, управляемый сообщениями, отвечает за обработку сообщений, а его контейнер заботится об автоматическом управлении всем окружением компонента, включающим в себя транзакции, безопасность, ресурсы, совместный доступ и подтверждения получения сообщений.

Один из наиболее важных аспектов компонентов, управляемых сообщениями, состоит в том, что они могут получать и обрабатывать сообщения параллельно. Эта возможность дает им значительное преимущество перед обычными клиентами JMS, в которых управление ресурсами, транзакциями и безопасностью в многопоточном окружении должно выполняться самим клиентом. Контейнеры компонентов, управляемых сообщениями, поставляемые вместе с EJB, управляют совместным доступом автоматически, поэтому разработчик компонента может направить свои усилия только на создание прикладной логики обработки сообщений. MDB может принимать сотни сообщений JMS от разных прикладных программ и обрабатывать их все в одно и то же время благодаря тому, что в контейнере могут выполняться несколько экземпляров MDB одновременно.

Компонент, управляемый сообщениями, – это полноценный компонент, точно такой же, как и сеансовый или объектный компонент, но имеющий несколько важных отличий. Хотя у компонента, управляемого сообщениями, есть и класс компонента и XML-дескриптор развертывания, у него нет компонентных интерфейсов. Компонентные интерфейсы отсутствуют из-за того, что компонент, управляемый сообщениями, не доступен через программный интерфейс Java RMI, он отвечает только на асинхронные сообщения.

---

[1] Детали работы JMS описываются в «Java Message Service» (Служба сообщений Java) Ричарда Монзона-Хейфела (Richard Monson-Haefel) и Дэвида Чапела (David Chappell), издательство O’Reilly.

Компонент ReservationProcessor

Компонент ReservationProcessor – это компонент, управляемый сообщениями, который принимает сообщения JMS, информирующие его о новых заказах билетов. Компонент ReservationProcessor представляет собой автоматизированную версию компонента TravelAgent, обрабатывающую заказы билетов, переданные через JMS другими туристическими организациями. Он не требует никакого вмешательства человека, он полностью автоматизирован.

Сообщения JMS, уведомляющие компонент ReservationProcessor о новых заказах билетов, могли бы исходить из другого приложения предприятия или приложения, находящегося в другой организации. Когда компонент ReservationProcessor принимает сообщение, он создает новый компонент Reservation (добавляя его к базе данных), с помощью компонента ProcessPayment обрабатывает оплату и подготавливает билет. Этот процесс показан на рис. 13.3.

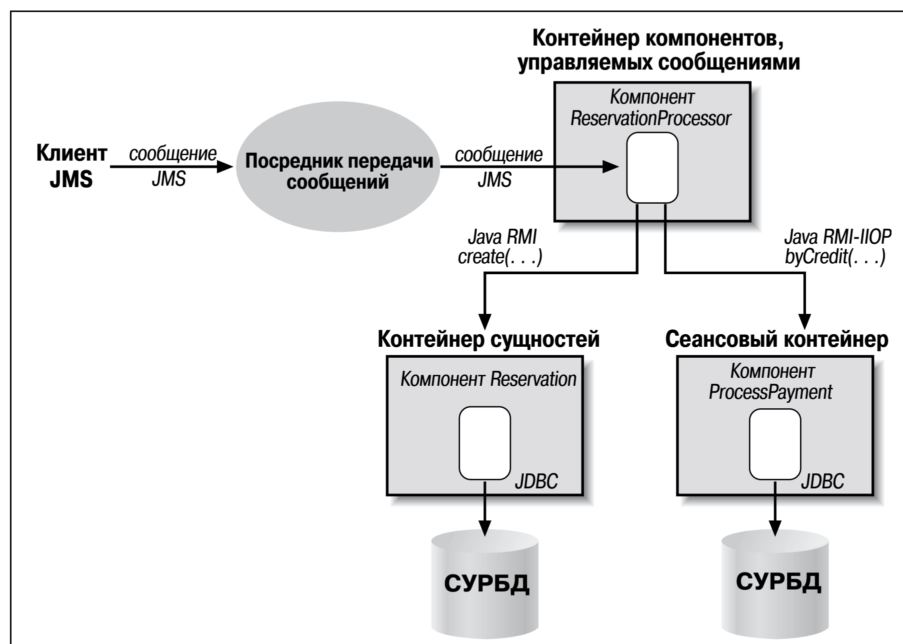

Рис. 13.3. Компонент ReservationProcessor, обрабатывающий заказы билетов

Класс ReservationProcessorBean

Здесь приведен фрагмент определения класса `ReservationProcessorBean`. Некоторые методы оставлены пустыми, они будут заполнены позже. Обратите внимание, что метод `onMessage()` содержит прикладную логику класса компонента, похожую на прикладную логику, содержащуюся в методе `bookPassage()` компонента TravelAgent в главе 12:

```
package com.titan.reservationprocessor;

import javax.jms.Message;
import javax.jms.MapMessage;
import com.titan.customer.*;
import com.titan.cruise.*;
import com.titan.cabin.*;
import com.titan.reservation.*;
import com.titan.processpayment.*;
import com.titan.travelagent.*;
import java.util.Date;

public class ReservationProcessorBean implements
javax.ejb.MessageDrivenBean,
    javax.jms.MessageListener {

    MessageDrivenContext ejbContext;
    Context jndiContext;

    public void setMessageDrivenContext(MessageDrivenContext mdc) {
        ejbContext = mdc;
        try {
            jndiContext = new InitialContext();
        } catch(NamingException ne) {
            throw new EJBException(ne);
        }
    }

    public void ejbCreate() {}

    public void onMessage(Message message) {
        try {
            MapMessage reservationMsg = (MapMessage)message;

            Integer customerPk =
(Integer)reservationMsg.getObject("CustomerID");
            Integer cruisePk = (Integer)reservationMsg.getObject("CruiseID");
            Integer cabinPk = (Integer)reservationMsg.getObject("CabinID");

            double price = reservationMsg.getDouble("Price");

            // получаем кредитную карточку
            Date expirationDate =
                new Date(reservationMsg.getLong("CreditCardExpDate"));
            String cardNumber = reservationMsg.getString("CreditCardNum");
            String cardType = reservationMsg.getString("CreditCardType");
            CreditCardDO card = new CreditCardDO(cardNumber,
                expirationDate, cardType);

            CustomerRemote customer = getCustomer(customerPk);
            CruiseLocal cruise = getCruise(cruisePk);
            CabinLocal cabin = getCabin(cabinPk);

            ReservationHomeLocal resHome = (ReservationHomeLocal)
              jndiContext.lookup("java:comp/env/ejb/ReservationHomeLocal");
```

```
            ReservationLocal reservation =
                resHome.create(customer, cruise, cabin, price, new Date());

            Object ref = jndiContext.lookup
                ("java:comp/env/ejb/ProcessPaymentHomeRemote");

            ProcessPaymentHomeRemote ppHome = (ProcessPaymentHomeRemote)
           PortableRemoteObject.narrow(ref, ProcessPaymentHomeRemote.class);

            ProcessPaymentRemote process = ppHome.create();
            process.byCredit(customer, card, price);

            TicketDO ticket = new TicketDO(customer,cruise,cabin,price);

            deliverTicket(reservationMsg, ticket);

        } catch(Exception e) {
            throw new EJBException(e);
        }
    }

    public void deliverTicket(MapMessage reservationMsg, TicketDO ticket) {

        // посылаем его в нужный пункт назначения
    }
    public CustomerRemote getCustomer(Integer key)
        throws NamingException, RemoteException, FinderException {
        // получаем удаленную ссылку на компонент Customer
    }
    public CruiseLocal getCruise(Integer key)
        throws NamingException, FinderException {
        // получаем локальную ссылку на компонент Cruise
    }
    public CabinLocal getCabin(Integer key)
        throws NamingException, FinderException {
        // получаем локальную ссылку на компонент Cabin
    }

    public void ejbRemove() {
        try {
            jndiContext.close();
            ejbContext = null;
        } catch(NamingException ne) { /* ничего не делаем */ }
    }
}
```

Интерфейс MessageDrivenBean

Классу компонента, управляемого сообщениями, необходимо реализовать интерфейс `javax.ejb.MessageDrivenBean`, в котором определены методы обратного вызова, похожие на те, которые были определены для объектных и сеансовых компонентов. Приведем определение интерфейса `MessageDrivenBean`:

```
package javax.ejb;

public interface MessageDrivenBean extends javax.ejb.EnterpriseBean {
    public void setMessageDrivenContext(MessageDrivenContext context)
        throws EJBException;
    public void ejbRemove() throws EJBException;
}
```

Метод `setMessageDrivenContext()` вызывается в начале жизненного цикла MDB и предоставляет экземпляру MDB ссылку на его `MessageDrivenContext`:

```
MessageDrivenContext ejbContext;
Context jndiContext;

public void setMessageDrivenContext(MessageDrivenContext mdc) {
    ejbContext = mdc;
    try {
        jndiContext = new InitialContext();
    } catch(NamingException ne) {
        throw new EJBException(ne);
    }
}
```

Метод `setMessageDrivenContext()` класса `ReservationProcessorBean` устанавливает поле экземпляра `ejbContext` в значение `MessageDrivenContext`, которое было передано в метод. Кроме этого он получает ссылку на JNDI ENC, которую сохраняет в `jndiContext`. MDB может иметь поля экземпляра, похожие на поля экземпляра сеансового компонента без состояния. Значения этих полей сохраняются в экземпляре MDB в течение всей его жизни и могут многократно использоваться каждый раз, когда он обрабатывает новое сообщение. В отличие от сеансовых компонентов с состоянием, у MDB нет состояния диалога, и они не предназначаются для одного клиента JMS. Экземпляры MDB применяются для обработки сообщений от нескольких разных клиентов JMS и связаны не с клиентом, а с определенной темой или очередью, от которых они принимают сообщения. Как и у сеансовых компонентов без состояния, у них нет состояния.

Метод `ejbRemove()` предоставляет экземпляру MDB возможность освобождения всех ресурсов, которые он содержит в своих полях экземпляра. В нашем случае он применяется для закрытия контекста JNDI и установки поля `ejbContext` в `null`. Эти действия не являются абсолютно необходимыми, но они показывают, что может происходить в методе `ejbRemove()`. Обратите внимание, что `ejbRemove()` вызывается в конце жизненного цикла MDB перед тем, как он будет удален сборщиком мусора. Он не может быть вызван в случаях, когда сервер EJB, выполняющий данный MDB, терпит крах или когда в одном из методов экземпляра MDB генерируется исключение `EJBException`. Когда каким-либо методом экземпляра MDB возбуждается исключение `EJBException` (или

любой тип `RuntimeException`), экземпляр немедленно удаляется из памяти, а транзакция откатывается назад.

MessageDrivenContext

`MessageDrivenContext` просто расширяет `EJBContext`, он не содержит никаких новых методов. `EJBContext` определен как:

```
package javax.ejb;
public interface EJBContext {

    // транзакционные методы
    public javax.transaction.UserTransaction getUserTransaction()
        throws java.lang.IllegalStateException;
    public boolean getRollbackOnly() throws java.lang.IllegalStateException;
    public void setRollbackOnly() throws java.lang.IllegalStateException;

    // внутренние методы компонента
    public EJBHome getEJBHome();
    public EJBLocalHome getEJBLocalHome();

    // методы безопасности
    public java.security.Principal getCallerPrincipal();
    public boolean isCallerInRole(java.lang.String roleName);

    // отмененные методы
    public java.security.Identity getCallerIdentity();
    public boolean isCallerInRole(java.security.Identity role);
    public java.util.Properties getEnvironment();

}
```

Компонентам, управляемым сообщениями, доступны только те транзакционные методы, которые `MessageDrivenContext` наследует от `EJBContext`. Внутренние методы – `getEJBHome()` и `getEJBLocalHome()` – при вызове генерируют исключение `RuntimeException`, т. к. у MDB нет ни внутренних интерфейсов, ни внутренних объектов. Методы безопасности интерфейса `MessageDrivenContext` – `getCallerPrincipal()` и `isCallerInRole()` – также при вызове генерируют RuntimeException. Когда MDB обслуживает сообщение JMS, не существует никакого «вызывающего объекта», поэтому отсутствует какой-либо контекст безопасности, который можно было бы получить из него. Помните, что служба JMS является асинхронной и не передает получателю контекст безопасности отправителя. Это не имело бы смысла, поскольку отправители и получатели обычно работают в разных окружениях.

MDB обычно выполняются в инициированной контейнером или компонентом транзакции, поэтому транзакционные методы позволяют MDB управлять своим контекстом. Контекст транзакции не передается от отправителя JMS, а инициируется либо контейнером, либо компонентом явным использованием `javax.jta.UserTransaction`. Транзакционные методы `EJBContext` обсуждаются более подробно в главе 14.

Кроме этого компоненты, управляемые сообщениями, имеют доступ к своему собственному контексту имен окружения (Environment Naming Context, ENC) JNDI, который предоставляет экземплярам MDB доступ к элементам окружения, к другим компонентам и ресурсам. Например, компонент ReservationProcessor пользуется возможностями JNDI ENC для получения ссылок на компоненты Customer, Cruise, Cabin, Reservation и ProcessPayment, а также QueueConnectionFactory JMS и Queue для рассылки билетов.

Интерфейс MessageListener

Кроме интерфейса MessageDrivenBean MDB реализуют интерфейс javax.jms.MessageListener, в котором определен метод onMessage(). Разработчики компонентов реализуют данный метод для обработки сообщений JMS, получаемых компонентом. Именно в методе onMessage() компонент обрабатывает сообщение JMS:

```
package javax.jms;
public interface MessageListener {
    public void onMessage(Message message);
}
```

Интересно рассмотреть, почему MDB реализует интерфейс MessageListener отдельно от интерфейса MessageDrivenBean. Почему бы просто не поместить метод onMessage(), единственный метод интерфейса MessageListener, в интерфейс MessageDrivenBean так, чтобы в классе MDB надо было реализовывать только один интерфейс? Именно это решение было принято в ранней рабочей версии EJB 2.0. Однако разработчики быстро осознали, что в будущем компоненты, управляемые сообщениями, могли бы обрабатывать сообщения из других типов систем, а не только JMS. Для того чтобы сделать MDB открытым для других систем сообщений, было решено, что он должен реализовать интерфейс javax.jms.MessageListener отдельно, таким образом отделяя понятие компонента, управляемого сообщениями, от типа сообщений, которые он может обрабатывать. В будущих версиях спецификации могут стать доступными другие типы MDB для таких технологий, как SMTP (электронная почта) или JAXM (Java API для XML-сообщений) для ebXML. Эти технологии будут использовать методы, отличные от onMessage(), являющегося специфичным для JMS.

Метод OnMessage (): Рабочий поток и интеграция с B2B

Именно в методе onMessage() выполняется вся прикладная логика. Сообщения по мере их поступления передаются в MDB контейнером через метод onMessage(). После возвращения из метода MDB готов обработать новое сообщение.

В компоненте ReservationProcessor метод onMessage() извлекает информацию, относящуюся к заказу билетов, из MapMessage и использует эту информацию для выполнения резервирования в системе:

```
public void onMessage(Message message) {
    try {
        MapMessage reservationMsg = (MapMessage)message;

        Integer customerPk = (Integer)reservationMsg.getObject("CustomerID");
        Integer cruisePk = (Integer)reservationMsg.getObject("CruiseID");
        Integer cabinPk = (Integer)reservationMsg.getObject("CabinID");

        double price = reservationMsg.getDouble("Price");

        // получаем кредитную карточку

        Date expirationDate =
            new Date(reservationMsg.getLong("CreditCardExpDate"));
        String cardNumber = reservationMsg.getString("CreditCardNum");
        String cardType = reservationMsg.setString("CreditCardType");
        CreditCardDO card = new CreditCardDO(cardNumber,
            expirationDate, cardType);
```

JMS нередко используется в качестве точки интеграции для прикладных программ типа «бизнес-бизнес», поэтому легко представить себе сообщение о заказе билетов, исходящее от одного из деловых партнеров системы «Титан» (возможно, от посредника или отраслевого транспортного агентства).

Для того чтобы обработать заказ билетов, компоненту ReservationProcessor необходимо обратиться к компонентам Customer, Cruise и Cabin. MapMessage содержит первичные ключи для этих объектов. Компонент ReservationProcessor применяет вспомогательные методы (`getCustomer()`, `getCruise()` и `getCabin()`) для поиска компонентов с данными и получения на них ссылок компонентных объектов:

```
public void onMessage(Message message) {
    ...
    CustomerRemote customer = getCustomer(customerPk);
    CruiseLocal cruise = getCruise(cruisePk);
    CabinLocal cabin = getCabin(cabinPk);
    ...
}

public CustomerRemote getCustomer(Integer key)
    throws NamingException, RemoteException, FinderException {

    Object ref = jndiContext.lookup("java:comp/env/ejb/CustomerHomeRemote");
    CustomerHomeRemote home = (CustomerHomeRemote)
        PortableRemoteObject.narrow(ref, CustomerHomeRemote.class);
    CustomerRemote customer = home.findByPrimaryKey(key);
    return customer;
}
public CruiseLocal getCruise(Integer key)
    throws NamingException, FinderException {

    CruiseHomeLocal home = (CruiseHomeLocal)
```

```
        jndiContext.lookup("java:comp/env/ejb/CruiseHomeLocal");
    CruiseLocal cruise = home.findByPrimaryKey(key);
    return cruise;
}
public CabinLocal getCabin(Integer key)
    throws NamingException, FinderException{

    CabinHomeLocal home = (CabinHomeLocal)
        jndiContext.lookup("java:comp/env/ejb/CabinHomeLocal");
    CabinLocal cabin = home.findByPrimaryKey(key);
    return cabin;
}
```

Информация, извлеченная из `MapMessage`, применяется для создания резервирования и обработки оплаты. Это в основном тот же рабочий поток, который использовался компонентом TravelAgent в главе 12. Компонент Reservation создается, чтобы представлять сам заказ билетов, а компонент ProcessPayment создается для обработки оплаты по кредитной карточке:

```
ReservationHomeLocal resHome = (ReservationHomeLocal)
    jndiContext.lookup("java:comp/env/ejb/ReservationHomeLocal");

ReservationLocal reservation =
    resHome.create(customer, cruise, cabin, price, new Date());

Object ref = jndiContext.lookup("java:comp/env/ejb/
ProcessPaymentHomeRemote");

ProcessPaymentHomeRemote ppHome = (ProcessPaymentHomeRemote)
    PortableRemoteObject.narrow (ref, ProcessPaymentHomeRemote.class);

ProcessPaymentRemote process = ppHome.create();
process.byCredit(customer, card, price);

TicketDO ticket = new TicketDO(customer,cruise,cabin,price);

deliverTicket(reservationMsg, ticket);
```

Из этого видно, что, как и сеансовый компонент, MDB может обращаться к любому другому объектному или сеансовому компоненту и использовать его для выполнения своей задачи. Таким образом, MDB играет роль точки интеграции в сценариях с приложениями B2B. MDB может управлять процессом и взаимодействовать с другими компонентами и ресурсами. Например, для MDB обычной практикой является применение JDBC для обращения к базе данных на основе содержания сообщения, которое он обрабатывает.

Посылка сообщений из компонента, управляемого сообщениями

MDB также может посылать сообщения, используя JMS. Метод `deliverTicket()` посылает информацию о билете в пункт назначения, полученный из посылки клиента JMS:

```
public void deliverTicket(MapMessage reservationMsg, TicketDO ticket)
    throws NamingException, JMSException{

    Queue queue = (Queue)reservationMsg.getJMSReplyTo();

    QueueConnectionFactory factory = (QueueConnectionFactory)
        jndiContext.lookup("java:comp/env/jms/QueueFactory");

    QueueConnection connect = factory.createQueueConneciton();

    QueueSession session = connect.createQueueSession(true,0);

    QueueSender sender = session.createSender(queue);

    ObjectMessage message = session.createObjectMessage();
    message.setObject(ticket);

    sender.send(message);

    connect.close();

}
```

Как было сказано выше, каждый тип сообщения состоит из двух частей: заголовка сообщения и тела сообщения (т. е. полезной информации). Заголовок сообщения содержит информацию о маршрутизации и может также содержать свойства для фильтрации сообщений и другие атрибуты, в том числе атрибут `JMSReplyTo`. Клиент JMS, посылая сообщение, может установить атрибут `JMSReplyTo` в значение какого-либо пункта назначения, доступного для его провайдера JMS.[1] В случае с сообщением о заказе билетов отправитель устанавливает атрибут `JMSReplyTo` в значение очереди, в которую должен быть послан итоговый билет. Другое приложение может обратиться к этой очереди, чтобы считать билеты и распределять их по клиентам или сохранить эту информацию в базе данных отправителя.

Можно также с помощью адреса `JMSReplyTo` указать на прикладную ошибку, произошедшую во время обработки сообщения. Например, если каюта уже занята, компонент ReservationProcessor мог бы послать в очередь `JMSReplyTo` сообщение об ошибке, объясняющее, что заказ не может быть обработан. Реализация такой обработки ошибок оставлена в качестве упражнения для читателя.

XML-дескриптор развертывания

У MDB есть XML-дескрипторы развертывания, точно так же, как у объектных и сеансовых компонентов. Они могут быть развернуты отдельно или, чаще всего, вместе с другими компонентами. Например,

[1] Очевидно, что если пункт назначения, указанный атрибутом JMSReplyTo, имеет тип Queue, должна быть использована модель сообщений «точка-точка» (на основе очередей). Если же пункт назначения имеет тип Topic, следует выбрать модель сообщений типа «издание-подписка» (на основе тем).

компонент ReservationProcessor необходимо было бы развертывать в том же архиве JAR, используя тот же XML-дескриптор развертывания, что и компоненты Customer, Cruise и Cabin, если он планирует использовать их локальные интерфейсы.

Здесь приведен XML-дескриптор развертывания, в котором определяется компонент ReservationProcessor. Этот дескриптор развертывания также определяет компоненты Customer, Cruise, Cabin и другие, но для краткости они опущены:

```
<enterprise-beans>
    ...
    <message-driven>
        <ejb-name>ReservationProcessorEJB</ejb-name>
        <ejb-class>
            com.titan.reservationprocessor.ReservationProcessorBean
        </ejb-class>
        <transaction-type>Container</transaction-type>
        <message-selector>MessageFormat = 'Version 3.4'</message-selector>
        <acknowledge-mode>Auto-acknowledge</acknowledge-mode>
        <message-driven-destination>
            <destination-type>javax.jms.Queue</destination-type>
        </message-driven-destination>
        <ejb-ref>
            <ejb-ref-name>ejb/ProcessPaymentHomeRemote</ejb-ref-name>
            <ejb-ref-type>Session</ejb-ref-type>
            <home>com.titan.processpayment.ProcessPaymentHomeRemote</home>
            <remote>com.titan.processpayment.ProcessPaymentRemote</remote>
        </ejb-ref>
        <ejb-ref>
            <ejb-ref-name>ejb/CustomerHomeRemote</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <home>com.titan.customer.CustomerHomeRemote</home>
            <remote>com.titan.customer.CustomerRemote</remote>
        </ejb-ref>
        <ejb-local-ref>
            <ejb-ref-name>ejb/CruiseHomeLocal</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <local-home>com.titan.cruise.CruiseHomeLocal</local-home>
            <local>com.titan.cruise.CruiseLocal</local>
        </ejb-local-ref>
        <ejb-local-ref>
            <ejb-ref-name>ejb/CabinHomeLocal</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
            <local-home>com.titan.cabin.CabinHomeLocal</local-home>
            <local>com.titan.cabin.CabinLocal</local>
        </ejb-local-ref>
        <ejb-local-ref>
            <ejb-ref-name>ejb/ReservationHomeLocal</ejb-ref-name>
            <ejb-ref-type>Entity</ejb-ref-type>
          <local-home>com.titan.reservation.ReservationHomeLocal</local-home>
```

```
            <local>com.titan.reservation.ReservationLocal</local>
        </ejb-local-ref>
        <security-identity>
            <run-as>
                <role-name>everyone</role-name>
            </run-as>
        </security-identity>
        <resource-ref>
            <res-ref-name>jms/QueueFactory</res-ref-name>
            <res-type>javax.jms.QueueConnectionFactory</res-type>
            <res-auth>Container</res-auth>
        </resource-ref>
    </message-driven>
    ...
</enterprise-beans>
```

MDB объявлен в элементе <message-driven> внутри элемента <enterprise-beans> рядом с компонентами <session> и <entity>. Как и компоненты типа <session>, он определяет <ejb-name>, <ejb-class> и <transaction-type>, но он не определяет компонентные интерфейсы (ни локальный, ни удаленный). У MDB нет компонентных интерфейсов, – вот почему эти определения не нужны.

<message-selector>

В MDB также может быть указан элемент <message-selector> (селектор сообщений), являющийся уникальным для компонентов, управляемых сообщениями:

```
<message-selector>MessageFormat = 'Version 3.4'</message-selector>
```

Селекторы сообщений позволяют MDB выбирать сообщения, которые он принимает из определенной темы или очереди. Свойства Message выступают как критерии в условных выражениях селекторов сообщений.[1] В этих условных выражениях для описания сообщений, которые должны быть доставлены клиенту, применяется булева логика.

Свойства сообщений, которые используются селектором сообщений, представляют собой дополнительные заголовки, которые могут быть связаны с сообщением. Они дают разработчику приложений или производителю JMS возможность присоединить к сообщению дополнительную информацию. Интерфейс Message предоставляет несколько методов для получения и изменения свойств. Свойства могут иметь значение String или одно из примитивных значений (boolean, byte, short, int, long, float, double). Система именования свойств, вместе с их значениями и правилами преобразования, строго определяется в JMS.

[1] Селекторы сообщений также могут использовать информацию из заголовков сообщений, но это выходит за рамки данной книги.

Компонент ReservationProcessor использует фильтр селектора сообщений для выбора сообщений определенного формата. В нашем случае форматом является строка «Version 3.4». По этой строке в системе «Титан» осуществляется идентификация сообщений типа MapMessage, содержащих значения имен CruiseID, CabinID, CreditCard и Price. Другими словами, указывая MessageFormat в каждом сообщении о резервировании, мы можем написать MDB, предназначенные для обработки разных видов сообщений о заказах билетов. Если новому деловому партнеру потребуется работать с отдельным типом объекта Message, «Титан» мог бы использовать новую версию сообщения и MDB для его обработки.

Ниже показано, как производитель JMS должен устанавливать свойство MessageFormat объекта Message:

```
Message message = session.createMapMessage();
message.setStringPropery("MessageFormat","Version 3.4");

// устанавливаем именованные значения для резервирования

sender.send(message);
```

Селекторы сообщений основаны на подмножестве синтаксиса условных выражений SQL-92, который используется в секции WHERE операторов SQL. Они могут быть достаточно сложными, включая применение литеральных значений, выражений типа Boolean, унарных операторов и т. д.

Примеры селекторов сообщений

Здесь приведены три сложных селектора, применяемые в гипотетических окружениях. И хотя вам придется немного напрячь воображение, цель этих примеров состоит в том, чтобы показать мощность селекторов сообщений. Во время объявления селектора идентификатор всегда ссылается на имя свойства или имя заголовка JMS. Например, селектор UserName !='William' предполагает, что в сообщении существует свойство с именем UserName, которое можно сравнивать со значением 'William'.

Управление заявками в HMO. Из-за нескольких ложных заявок реализован автоматический процесс, использующий MDB, который будет отслеживать все заявки, направляемые пациентами – служащими производственной компании ACME – на посещение хиромантов, физиологов и дерматологов:

```
<message-selector>
<![CDATA[
    PhysicianType IN ('Chiropractic','Psychologists','Dermatologist')
    AND PatientGroupID LIKE 'ACME%'
]]>
</message-selector>
```

Операторы MDB <message-selector> объявляются в XML-дескрипторах развертывания. XML присваивает специальное значение ряду символов, таким как символы больше (>) и меньше (<), поэтому применение этих символов в операторах <message-selector> вызовет ошибки синтаксического анализа, если не использовать разделы CDATA. Разделы CDATA необходимы по той же причине, что и в операторах EJB QL <ejb-ql>, как было показано в главе 8.

Уведомление об определенных предложениях на товары. Продавец желает получать уведомления о запросах по предложению цены на определенный товар с заданным количеством:

```
<message-selector>
<![CDATA[
    InventoryID ='S93740283-02' AND Quantity BETWEEN 1000 AND 13000
]]>
</message-selector>
```

Выбор получателей для рассылки по каталогу. Интернет-продавец хочет разослать специальный каталог всем клиентам, сделавшим заказы на сумму, превышающую $500, если средняя стоимость заказанного товара превышает $75 и покупатель постоянно находится в одном из нескольких штатов. Продавец создает MBD, который подписывается на тему обработки заказов и обрабатывает доставку каталога только для тех клиентов, которые удовлетворяют заданным критериям:

```
<message-selector>
<![CDATA[
    TotalCharge >500.00 AND ((TotalCharge /ItemCount)>=75.00)
    AND State IN ('MN','WI','MI','OH')
]]>
</message-selector>
```

<acknowledge-mode>

В JMS есть понятие подтверждения (acknowledge[ment]), которое означает, что клиент JMS уведомляет провайдер JMS (маршрутизатор сообщений) о получении сообщения. В EJB посылка подтверждения провайдеру JMS является обязанностью контейнера MDB, когда он принимает сообщение. Подтверждение сообщения информирует провайдер JMS, что контейнер MDB принял сообщение и обработал его с помощью экземпляра MDB. Без подтверждения провайдер JMS не будет знать, принял ли контейнер MDB сообщение, поэтому он попытается послать его повторно. Это может вызывать проблемы. Например, после того как мы обработали сообщение о заказе билетов, используя компонент ReservationProcessor, мы не должны снова получать это же сообщение.

Когда мы имеем дело с транзакциями, режим подтверждения, установленный провайдером компонента, игнорируется. В этом случае подтверждение выполняется внутри контекста транзакции. Если транзакция завершается успешно, сообщение подтверждается. Если происходит сбой транзакции, сообщение не подтверждается. Если MDB основывается на управляемых контейнером транзакциях, как это будет происходить в большинстве случаев, то контейнер MDB игнорирует режим подтверждения. При использовании управляемых контейнером транзакций с атрибутом транзакции Required элемент <acknowledge-mode> обычно не указывается. Однако мы включили его в дескриптор развертывания в учебных целях:

```
<acknowledge-mode>Auto-acknowledge</acknowledge-mode>
```

Если MDB выполняется внутри управляемой компонентом транзакции или внутри управляемой контейнером транзакции с атрибутом NotSupported (глава 14), значение <acknowledge-mode> становится значимым.

Для <acknowledge-mode> могут быть заданы два значения: Auto-acknowledge и Dups-ok-acknowledge. Auto-acknowledge указывает контейнеру, что он должен посылать подтверждение провайдеру JMS сразу после того, как сообщение будет передано для обработки экземпляру MDB. Dups-ok-acknowledge указывает контейнеру, что он не должен посылать подтверждение немедленно, это может быть сделано в любое время после того, как сообщение будет передано экземпляру MDB. Если задано значение Dups-ok-acknowledge, становится возможной ситуация, когда контейнер MDB задержит подтверждение настолько, что провайдер JMS решит, что сообщение не было получено, и пошлет «продублированное» сообщение. Очевидно, что в случае с Dups-ok-acknowledge ваши MDB должны уметь правильно обрабатывать повторные сообщения.

Auto-acknowledge предупреждает появление повторных сообщений из-за того, что подтверждение посылается немедленно. Поэтому провайдер JMS не будет посылать дубликат. В большинстве MDB Auto-acknowledge применяются для того, чтобы избежать повторной обработки одного сообщения. Dups-ok-acknowledge существует потому, что он дает возможность провайдеру JMS оптимизировать использование сети. Тем не менее, на практике накладные расходы на подтверждения настолько малы, а частота передачи между контейнером MDB и провайдером JMS настолько высока, что Dups-ok-acknowledge не оказывает на эффективность большого влияния.

<message-driven-destination>

Элемент <message-driven-destination> определяет тип пункта назначения, из которого MDB принимает сообщения. Разрешенные значения для этого элемента – javax.jms.Queue и javax.jms.Topic. В компоненте ReservationProcessor это значение установлено в javax.jms.Queue, показывая, что MDB получает свои сообщения через модель передачи сообщений p2p из очереди:

```
<message-driven-destination>
    <destination-type>javax.jms.Queue</destination-type>
</message-driven-destination>
```

Во время развертывания MDB необходимо связать его так, чтобы он прослушивал в сети существующую очередь.

Когда `<destination-type>` содержит `javax.jms.Topic`, элемент `<subscription-durability>` должен содержать в качестве своего значения или `Durable`, или `NonDurable`:

```
<message-driven-destination>
    <destination-type>javax.jms.Topic</destination-type>
    <subscription-durability>Durable</subscription-durability>
</message-driven-destination>
```

Элемент `<subscription-durability>` определяет, действительно ли подписка MDB на тему будет длительной (`Durable`). Длительная подписка «переживает» подключение контейнера MDB к провайдеру JMS. Так, если сервер EJB терпит частичный крах, выключается или как-то иначе отсоединяется от провайдера JMS, сообщения, которые он должен принять, не будут потеряны. Пока «длительный» контейнер MDB отсоединен от провайдера JMS, в обязанности провайдера входит сохранение всех сообщений, подписчик которых отсутствует. Когда длительный контейнер MDB воссоединяется с провайдером JMS, провайдер посылает ему все накопившиеся за время отсутствия связи не устаревшие сообщения. Такое поведение обычно называется *передачей сообщений с промежуточным накоплением* (*store-and-forward messaging*). Длительный MDB допускает потери соединения, являются ли они преднамеренными или происходят в результате частичного отказа.

Если `<subscription-durability>` установлен в `NonDurable`, все сообщения, которые компонент мог бы принять, пока он был отсоединен, будут потеряны. Разработчики используют подписки типа `NonDurable` в случаях, когда обработка всех сообщений не обязательна. Подписка `NonDurable` улучшает производительность провайдера JMS, но, вместе с тем, значительно уменьшает надежность MDB.

Если `<destination-type>` установлен в `javax.jms.Queue`, как в случае с компонентом ReservationProcessor, длительность не является критичной из-за самой природы p2p или систем сообщений, основанных на очереди. В случае применения очереди сообщения могут быть получены только один раз, и они могут оставаться в очереди, пока не будут переданы одному из слушателей очереди.

С остальными элементами дескриптора развертывания вы должны быть знакомы. Элемент `<ejb-ref>` предоставляет связь JNDI ENC для удаленного внутреннего объекта, тогда как элементы `<ejb-local-ref>` предоставляют связь JNDI ENC для локальных внутренних объектов. Обратите внимание, что элемент `<resource-ref>`, определяющий JMS `QueueConnectionFactory`, используемую компонентом ReservationProcessor

для отправки сообщения о билете, не соответствует никакому элементу <resource-env-ref>. Очередь, в которую посылаются билеты, извлекается из заголовка JMSReplyTo самой MapMessage, а не из JNDI ENC.

Клиенты ReservationProcessor

Для того чтобы протестировать компонент ReservationProcessor, нам необходимо разработать два новых клиентских приложения: одно – для отправки сообщения о заказе билетов, а второе – для приема сообщения о билете, сгенерированного компонентом ReservationProcessor.

Генератор сообщений о заказе билетов. Компонент JmsClient_ReservationProducer предназначен для очень быстрой отправки 100 запросов по заказам билетов. Быстродействие, с которым он посылает эти сообщения, вынудит большинство контейнеров MDB использовать несколько экземпляров компонента для обработки сообщений о резервировании. Код для JmsClient_ReservationProducer выглядит следующим образом:

```
import javax.jms.Message;
import javax.jms.MapMessage;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueConnection;
import javax.jms.QueueSession;
import javax.jms.Session;
import javax.jms.Queue;
import javax.jms.QueueSender;
import javax.jms.JMSException;
import javax.naming.InitalContext;
import java.util.Date;

import com.titan.processpayment.CreditCardDO;

public class JmsClient_ReservationProducer {

    public static void main(String [] args) throws Exception {

        InitialContext jndiContext = getInitialContext();

        QueueConnectionFactory factory = (QueueConnectionFactory)
            jndiContext.lookup("ИмяФабрикиОчереди");

        Queue reservationQueue = (Queue)
            jndiContext.lookup("ИмяОчереди");

        QueueConnection connect = factory.createQueueConneciton();

        QueueSession session =
            connect.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);

        QueueSender sender = session.createSender(reservationQueue);

        Integer cruiseID = new Integer(1);

        for(int i = 0; i < 100; i++){
```

```
            MapMessage message = session.createMapMessage();
            message.setStringProperty("MessageFormat","Version 3.4");

            message.setInt("CruiseID",1);
            message.setInt("CustomerID",i%10);
            message.setInt("CabinID",i);
            message.setDouble("Price", (double)1000+i);

            // действие карточки истекает примерно через 30 дней
            Date expirationDate = new
            Date(System.currentTimeMillis()+43200000);
            message.setString("CreditCardNum", "923830283029");
            message.setLong("CreditCardExpDate", expirationDate.getTime());
            message.setString("CreditCardType", CreditCardDO.MASTER_CARD);

            sender.send(message);

        }

        connect.close();
    }

    public static InitialContext getInitialContext()
        throws JMSException {
        // создаем контекст JNDI, специфичный для данного производителя
    }
}
```

Вы, возможно, заметили, что JmsClient_ReservationProducer устанавливает CustomerID, CruiseID и CabinID как значения примитивного типа int, но ReservationProcessorBean считывает эти значения в виде типов java.lang.Integer. Это не ошибка. MapMessage автоматически преобразует все примитивные типы к их соответствующим оберткам, если эти примитивы считываются с помощью MapMessage.getObject(). Так, именованное значение, загружаемое в MapMessage с помощью setInt(), может быть с помощью getObject() считано в виде Integer. Например, следующий код устанавливает значение в виде примитива int, а затем обращается к нему как к объекту java.lang.Integer:

```
MapMessage mapMsg = session.createMapMessage();

mapMsg.setInt("TheValue",3);

Integer myInteger = (Integer)mapMsg.getObject("TheValue");

if(myInteger.intValue() == 3 )
    // всегда true
```

Потребитель сообщения о билете. JmsClient_TicketConsumer предназначен для получения всех сообщений о билетах, направляемых экземплярами компонента ReservationProcessor в очередь. Он получает сообщения и распечатывает их описания:

```
import javax.jms.Message;
import javax.jms.ObjectMessage;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueConnection;
import javax.jms.QueueSession;
import javax.jms.Session;
import javax.jms.Queue;
import javax.jms.QueueReceiver;
import javax.jms.JMSException;
import javax.naming.InitalContext;

import com.titan.travelagent.TicketDO;

public class JmsClient_TicketConsumer
    implements javax.jms.MessageListener {

    public static void main(String [] args) throws Exception {

        new JmsClient_TicketConsumer();

        while(true){Thread.sleep(10000);}

    }

    public JmsClient_TicketConsumer() throws Exception {

        InitialContext jndiContext = getInitialContext();

        QueueConnectionFactory factory = (QueueConnectionFactory)
            jndiContext.lookup("ИмяФабрикиОчереди");

        Queue ticketQueue = (Queue)jndiContext.lookup("ИмяОчереди");

        QueueConnection connect = factory.createQueueConneciton();

        QueueSession session =
            connect.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);

        QueueReceiver receiver = session.createReceiver(ticketQueue);

        receiver.setMessageListener(this);

        connect.start();
    }

    public void onMessage(Message message) {
        try {

            ObjectMessage objMsg = (ObjectMessage)message;
            TicketDO ticket = (TicketDO)objMsg.getObject();
            System.out.println("********************************");
            System.out.println(ticket);
            System.out.println("********************************");

        } catch(JMSException jmsE) {
            jmsE.printStackTrace();
        }
    }
    public static InitialContext getInitialContext() throws JMSException {
```

```
            // создаем контекст JNDI, специфичный для производителя
        }
    }
```

Для того чтобы компонент ReservationProcessor мог работать с двумя клиентскими приложениями – `JmsClient_ReservationProducer` и `JmsClient_TicketConsumer`, вы должны настроить провайдер JMS вашего контейнера EJB так, чтобы у него было две очереди: одна для сообщений о резервировании, а другая – для сообщений о билетах.

Рабочее упражнение 13.2. Компонент, управляемый сообщениями

Жизненный цикл компонента, управляемого сообщениями

Так же как объектные и сеансовые компоненты, компоненты MDB имеют четко определенный жизненный цикл. Жизненный цикл экземпляра MDB имеет два состояния: *«не существует»* и *«пул готовых методов»*. Пул готовых методов похож на пул экземпляров, используемый для сеансовых компонентов без состояния. Как и для компонентов без состояния, для жизненных циклов MDB определен пул экземпляров.[1]

На рис. 13.4 показаны состояния и переходы, через которые проходит экземпляр MDB на протяжении своего времени жизни.

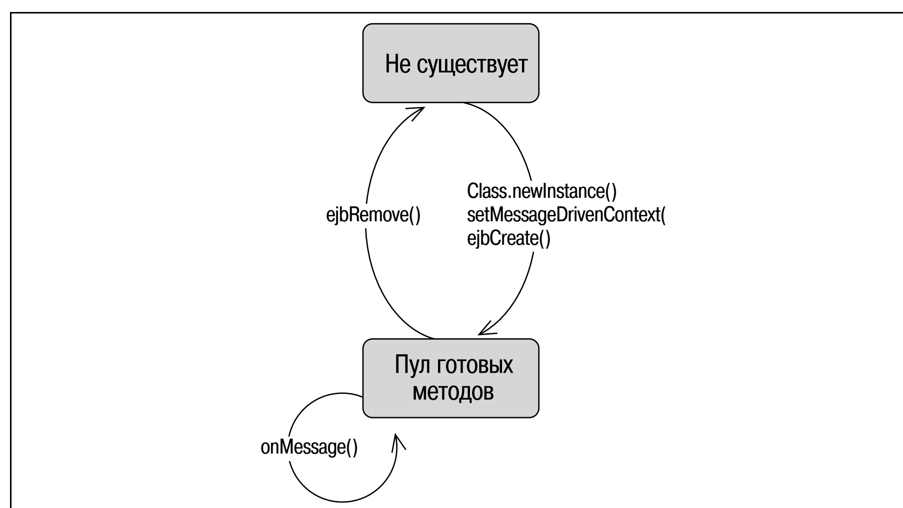

Рис. 13.4. Жизненный цикл MDB

[1] Некоторые производители могут не использовать пул для MDB-экземпляров, а вместо этого создавать и уничтожать экземпляры для каждого нового сообщения. Это – нестандартное решение, и оно не должно влиять на установленный жизненный цикл MDB-экземпляра компонента.

Не существует

Когда экземпляр MDB находится в состоянии «не существует», он не является экземпляром в памяти системы. Другими словами, он еще не создан.

Пул готовых методов

Экземпляры MDB переходят в пул готовых методов по мере того, как они становятся нужны контейнеру. При первом запуске сервер EJB может создавать несколько экземпляров MDB, помещая их в пул готовых методов. (Действительное поведение сервера зависит от реализации.) Когда количество экземпляров MDB, обрабатывающих входящие сообщения, становится недостаточным, могут быть созданы и добавлены к пулу дополнительные.

Переход в пул готовых методов

Когда экземпляр перемещается из состояния «не существует» в пул готовых методов, над ним выполняются три действия. Сначала посредством метода `Class.newInstance()` класса MDB создается экземпляр компонента. Далее контейнер вызывает метод `setMessageDrivenContext()`, предоставляя экземпляру MDB ссылку на его `EJBContext`. Ссылка на `MessageDrivenContext` может быть сохранена в поле экземпляра MDB.

Наконец, контейнером вызывается безаргументный метод `ejbCreate()` экземпляра компонента. У MDB есть только один метод `ejbCreate()`, не принимающий никаких параметров. Метод `ejbCreate()` вызывается только один раз в течение жизненного цикла MDB.

MDB не являются объектами, подлежащими активации, поэтому они могут поддерживать открытые соединения с ресурсами на всем протяжении их жизненного цикла.[1] Метод `ejbRemove()` должен закрывать все открытые ресурсы прежде, чем MDB будет удален из памяти в конце своего жизненного цикла.

Жизнь в пуле готовых методов

Если экземпляр находится в пуле готовых методов, он готов к обработке входящих сообщений. Сообщение, поступающее к MDB, перенаправляется любому доступному экземпляру, находящемуся в пуле готовых методов. Пока экземпляр выполняет запрос, он не может обрабатывать другие сообщения. MDB может одновременно обрабатывать

---

[1] Предполагается, что продолжительность жизни MDB-экземпляра будет очень большой. Однако в действительности некоторые серверы EJB могут уничтожать и создавать экземпляры для каждого нового сообщения, что делает такую тактику менее привлекательной. Подробности обработки MDB-экземпляров можно найти в документации по серверу.

несколько сообщений, делегируя обязанность по обработке каждого сообщения другому экземпляру MDB. Когда сообщение направлено экземпляру контейнером, `MessageDrivenContext` экземпляра MDB изменяется, чтобы отразить новый контекст транзакции. Экземпляр, закончивший обработку, сразу же становится доступным для обработки нового сообщения.

Переход из пула готовых методов: смерть экземпляра MDB

Экземпляры компонентов переводятся из пула готовых методов в состояние «не существует», когда они становятся ненужными серверу. Это происходит, когда сервер принимает решение уменьшить общий размер пула готовых методов, удаляя из памяти один или несколько экземпляров. Этот процесс начинается с вызова метода `ejbRemove()` экземпляра. В это время экземпляр компонента должен выполнить все необходимые действия по очистке, такие как закрытие открытых ресурсов. Метод `ejbRemove()` вызывается только один раз за жизненный цикл экземпляра MDB, когда контейнер собирается переводить его в состояние «не существует». Во время выполнения метода `ejbRemove()` для экземпляра компонента все еще доступен контекст `MessageDrivenContext` и открыт доступ к JNDI ENC. По завершении метода `ejbRemove()` удаляются все указывающие на компонент ссылки, после чего он удаляется сборщиком мусора.