Оглавление

Транзакции (ACID Transactions)

Транзакция — процесс выполнения единицы работы (Unit Of Work, UOW) некоторой системы, подчиняющийся определенным правилам, называемым транзакционным

механизмом.

ACID

ACID (атомарность согласованность изолированность надежность) – характеристика транзакции или требование к транзакционному механизму, обеспечивающие устойчивую работу системы.

Атомарность (atomicity) — либо все, либо ничего. Транзакция неделима и выполняется как единое целое. Либо ее результат фиксируется (commit), либо состояние системы откатывается к исходному (rollback).

Согласованность (consistency) – адекватность измененного состояния после завершения транзакции. Она обеспечивается как другими транзакционными свойствами (изоляцией и надежностью), так и бизнес-логикой приложения (нормализованное хранение в РБД, адекватность бизнес-логики окружающему миру).

Изолированность (isolation) — отсутствие влияния на транзакционные данные со стороны остальной (вне границ транзакции, например из других параллельных транзакций) части системы до фиксации транзакции.

Надежность (durable) — изменения, сделанные успешно-завершенной транзакцией (после сигнала о фиксации) должны сохраняться в системе даже при последующем сбое (например, сбое во второй фазе двухфазной фиксации). Для этого может вестись резервирование изменений состояния вплоть до фиксации. Резервная запись хода транзакции позволяет восстановить данные после сбоя системы.

Rem: обычно атомарность и надежность транзакции поддерживается при помощи непрерывного ее журналирования, т.е. записи операций в WAL (журнал упреждающей записи), хранящийся в ROM. После сбоя и перезапуска система сверяет текущее состояние РБД с WAL, дописывает в РБД из WAL все зафиксированное и дооткатывает все помеченное на откат.

Расширение транзакции. Транзакционные границы

В процедурном программировании под UOW логично понимать работу процедуры или части ее операций. Эти операции могут включать в себя вызовы других процедур, их выполнение также может включаться в UOW.

При выполнении операций UOW может затрагиваться состояние системы. Под транзакционными границами понимается та часть состояния системы, которая управляется транзакцией, т.е. подчинена ACID транзакционного механизма (фиксация, откат, изоляция).

В ООП выполнение методов (процедур) связано с объектами, поэтому можно говорить об объектах, вовлеченных в транзакцию. Объекты, чье состояние управляется транзакцией, называют транзакционными, а их совокупность составляют границы транзакции.

Вместе с расширением UOW расширяются и транзакционные границы. Поэтому эти явления называют расширением транзакции (Transaction Propagation).

Транзакционный механизм ACID налагает на UOW и состояние системы в транзакционных границах ряд ограничений: блокировку, ограничения видимости внешнего состояния, возможность фиксации и отката. Поэтому для расширения транзакций и изоляции придуманы различные политики.

Локальные и распределенные транзакции. 2PC

Локальная транзакция — транзакция, которая полностью управляет своим UOW в рамках единственного транзакционного механизма.

Но при расширении транзакции возможен и вариант попадания в UOW операций, вовлеченных в другую транзакцию, порой даже из другой системы. В этом случае возможно взаимодействие между транзакциями: определяется главная транзакция, называемая распределенной, она синхронизирует свою работу с остальными. Обычно это происходит по протоколу двухфазной фиксации (2PC).

2PC (Two Phase Commit) - механизм фиксации/отката распределенных транзакций в 2 этапа: 1 — опрос локальных транзакций о готовности/неготовности фиксации; 2 — принятие глобального решения commit/rollback и финальная рассылка локальным транзакциям указания применить его.

Пример транзакций в JDBC

Современные СУБД обладают встроенными транзакционными механизмами, управлять которыми в Java-приложениях можно программно при помощи технологии JDBC. В интерфейсе java.sql.Connection по-умолчанию каждая инструкция оборачивается в свою транзакцию, которая фиксируется после успешного завершения этой инструкции. Но можно объединить в UOW несколько инструкций, отменив автофиксацию и разметив транзакционные границы:

```
… Connection con = getConnection();
con.setAutoCommit(false);
try{
  Statement st = con.createStatement();
  …
  st.executeUpdate(sqlStr1);
  st.executeUpdate(sqlStr2);
  con.commit();
```

```
}
catch(Throwable t){
  con.rollback();
}
```

В примере отменяется автофиксация con.setAutoCommit(false), транзакция начинается с создания выражения st и заканчивается вызовом фиксации con.commit(). При возникновении сбоя внутри транзакции происходит ее откат con.rollback() в операторе catch(...).

Rem: транзакция относится только к РБД, она открывается в менеджере РБД (JDBC), куда и отправляются все вызовы от Connection и Statement. Другие операции между con.createStatement() и con.commit() в транзакцию не попадут, равно как и затронутые ими данные не попадут в транзакционные границы.

Явное управление транзакционными механизмами ресурсов становится сложным с распространением транзакции на множество объектов приложения, а в случае наличия нескольких ресурсов транзакция становится распределенной и требует механизма согласования работы локальных транзакций этих ресурсов. Поэтому в EJB-контейнере есть свой транзакционный механизм, автоматизирующий управление транзакциями приложения.

Пример ACID-транзакции в BlackJack

Игра в «очко». Моделируется ! состояние в бизнес-методе:

```
public interface BlackJackGameLocalBusiness {
  boolean bet(long userID, BigDecimal amount);
}
```

При выигрыше на счет userID идет amount $ и возвращается true, а со счета игры списывается amount. В противном случае — перевод делается наоборот.

Для обработки денежных транзакций воодится интерфейс простейшего банка:

```
public interface BankLocalBusiness {
  BigDecimal withdraw(long accountID, BigDecimal amount);
  BigDecimal deposit(long accountID, BigDecimal amount);
  BigDecimal getBalance(long accountID);
  void transfer(long accountIDFrom, long accountIDTo, BigDecimal amount);
}
```

withdraw – снятие со счета и deposit – перевод на счет выдают остаток на счете после операции, getBalance выдает остаток на счете, transfer осуществляет перевод.

Для игры с транзакциями вводится интерфейс-обертка, организующая выполнение транзакции в потоке. Это позволяет вручную задавать границы UOW :

```
public interface TxWrappingLocalBusiness {
  <T> wrapInTx(Callable<T> task);
  <T> wrapperInTx(Callable<T> … tasks);
}
```

Данный интерфейс будет реализован в SLSB, callable – задачи передаются в один из его методов, где и запускаются в потоках в рамках временной транзакции этого метода. Это нужно для заворачивания бизнес-логики в транзакции для тестирования доступа к JPA-Entity перед отсоединением (detach) и проверки отката (rollback) при возникновении сбоев.

Атомарность — перечисление средств после одной игры в рамках одной транзакции.

Согласованность — требование адекватности состояния игроков после транзакции. Это требует от транзакции CID. В игре типовым нарушением согласованности будет сбой в начислении средств на один счет после снятия с другого — деньги пропадут. Помимо приложения, согласованность должна быть обеспечена и на уровне РБД. Банковский аккаунт не должен быть связан с null-клиентами и все его FK – userID не должны быть пустыми.

Изоляция — заключается в блокировке доступа к данным, используемым в транзакции. Иначе, например, несколько транзакций могут снять средства с одного счета, вызвав превышение баланса. Поэтому в игре должны быть заблокированы счета участников, между которыми происходит обмен в рамках одной транзакции. Это достигается как потоковой синхронизацией, так и блокировкой строк в РБД.

Надежность — означает гарантированную сохранность результатов успешной транзакции. Например, если клиент получил в ходе 2PC извещение от некоторой подсистемы приложения об успешном завершении своей надежной транзакции, то он может быть уверен, что ее результаты не пропадут и будут зафиксированы в системе даже в случае последующего сбоя. Надежность обычно реализуется сохранением информации о транзакции бизнес-системы (журналированием) на долговременных носителях (ROM). Физическая память (RAM) – ненадежное хранилище, т.к. хранит данные только пока работает система. Сброс данных на долговременный носитель после каждой транзакции и разрешение доступа к ним только после её фиксации обеспечивают надежность и изолированность. Ведение журнала транзакции позволяет восстановить ее ход при сбоях.

О транзакционной архитектуре в JEE7

Транзакции приложения в JEE7 можно поделить на локальные и глобальные (распределенные).

Локальные транзакции относятся к некоторому внешнему ресурсу, приложение использует его локальный транзакционный механизм. Такие транзакции характерны

для JSE-приложений, но могут применяться и в JEE.

Глобальные транзакции охватывают множество транзакционных ресурсов. Ими управляет специальный менеджер транзакций. Глобальные транзакции используются в JEE-приложениях и на высоком уровне описываются JTA (Java Transaction API), поэтому также они называются JTA-транзакциями. Менеджер транзакций есть во всех JEE-контейнерах: в EJB-контейнере глобальные транзакции можно использовать через API (JTA) и транзакционную разметку методов, остальные контейнеры используют транзакционную разметку с перехватчиками CDI.

Локальные транзакции (Resource Local)

Локальная транзакция — транзакция, которая работает отдельно с некоторым ресурсом при помощи его внутреннего транзакционного механизма. Этой локальной транзакцией можно управлять через клиентский API, называемый менеджером ресурса. Локальные транзакции обычно применяются в JSE-приложениях.

Ex: фрагмент локальной транзакции в JSE, управляемой JDBC (менеджером РБД)

```
Connection con = xxx;
con.setAutoCommit(false);
try{
  Statement st = con.createStatement();
  st.executeUpdate(sqlStr1);
  st.executeUpdate(sqlStr2);
  con.commit();
} catch(Throwable t){
  con.rollback();
}
```

Ex: фрагмент с локальной транзакции в JEE, управляемой менеджером РБД через API JPA:

```
  EntityManagerFactory emf = Persistence.createEntityManagerFactory("exUnitName",propertyMap);
  em = emf.createEntityManager();
  EntityTransaction et = em.getTransaction();
  et.begin();
  Query q = em.createQuery(jpqlStr);
  q.executeUpdate();
  et.commit();
  em.close();
  emf.close();
```

Rem: транзакция управляет только транзакционным ресурсом, она создается в его менеджере и охватывает вызовы к нему. Все остальное, даже попавшее между вызовами транзакционных границ, к транзакции не относится.

Глобальные транзакции (JTA)

Распределенная (глобальная) транзакция — транзакция, которая может включать в себя множество локальных транзакций внешних ресурсов и код самого приложения. Для управления ими нужен диспетчер, который называют менеджером транзакций и который взаимодействует с менеджерами ресурсов. При этом локальные транзакции обычно синхронизируются менеджером транзакций по протоколу двухфазной фиксации.

Rem: 2PC (Two Phase Commit) - механизм фиксации распределенных транзакций в 2 этапа: 1 — опрос менеджеров ресурсов (локальные транзакции) о готовности/неготовности фиксации; 2 — принятие глобального решения commit/rollback и финальная рассылка менеджерам ресурсов указания применить его.

В JEE глобальные транзакции описываются на высоком уровне Java Transaction API, поэтому их называют JTA-транзакциями. Существует два подхода использования JTA-транзакций:

- декларативный, в котором транзакцией управляет JEE-контейнер (Container Managed Transaction, CMT), а транзакционная политика декларируется в deploy-time метаинформацией. CMT в EJB-контейнере описана собственной спецификацией, а в остальных контейнерах JEE7 работает по спеке JPA 1.2 благодаря скрытому применению CDI-перехватчиков
- прикладной, в котором транзакцией управляет код самого транзакционного компонента (Bean Management Transaction, BMT). BMT доступна в EJB-контейнере, в остальных контейнерах ее можно теоретически подключить через JNDI.

Rem: по умолчанию в EJB используется JTA CMT

Ex: Пример JTA CMT в EJB-контейнере (границы транзакции совпадают с границами метода)

```
@TransactionManagement(TransactionManagementType.CONTAINER)
@Stateless
public class ExSLSB implements ExRemote {
@PersistenceContext(unitName = ”ExPU”)
EntityManager em;

public void exMethod(int empId, int taskId) {
  Bar bar = new Bar(33,”bar33”);
  em.merge(bar);
}
```

Ex: пример JTA CMT в CDI MBean (границы транзакции совпадают с границами метода)

```
@RequestScoped
public class ExampleBean {

   @Transactional
```

```
    public void exMethod() {
        ...
    }
}
```

```
Ex:  пример JTA BMT в EJB-контейнере

@TransactionManagement(TransactionManagementType.BEAN)
@Stateful
public class ExEJB implements ExEJBRemote {
  @PersistenceUnit(unitName="ExPU")
  private EntityManagerFactory emf;
  @Resource
  private SessionContext context;

  @Override
  public void test() {
    try {
      UserTransaction utx = context.getUserTransaction();
      utx.begin();
      Bar bar = new Bar(33,”bar33”);
      EntityManager em = emf.createEntityManager();
      em.merge(bar);
      utx.commit();
      em.close();
      emf.close();
      System.out.println(cache.contains(Bar.class, bar.getId()));
    } catch (Exception ex) {
      ex.printStackTrace();
      throw EJBException();
    }
  }
}
```

Rem: JTA оперирует только локальными транзакциями транзакционных ресурсов, а те живут в своих менеджерах и охватывают вызовы к ним. Все остальное между вызовами транзакционных границ или в транзакционных методах, к транзакциям не относится.

Глобальные транзакции. Схема, спецификации, JTA и JTS

В спецификациях, описывающих транзакционные механизмы JEE используются общепринятые стандарты, описанные организациями Open Group (OG), Object Management Group (OMG). Иными словами архитектура транзакций в JEE7 основана на стандартных специфицированных моделях.

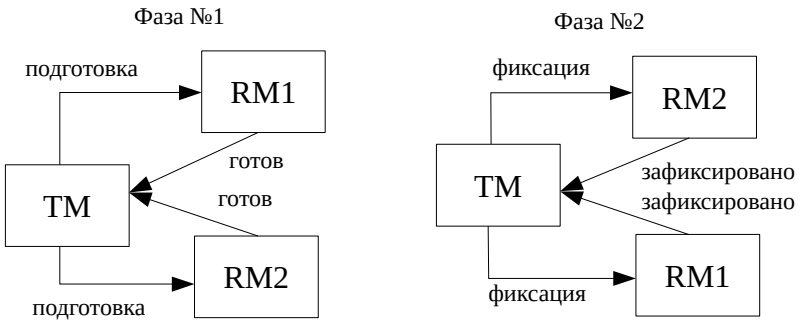
Непосредственное управление транзакцией ресурса возложено на его менеджер (Resource Manager, RM). Это программный компонент JEE, отвечающий за управление ресурсом. Он поддерживает собственные (локальные) транзакции ресурса, ведет их журналирование и регистрирует их в менеджере транзакций. Примером является РБД и

ее менеджер — JDBC (управляемый контейнером DataSource).

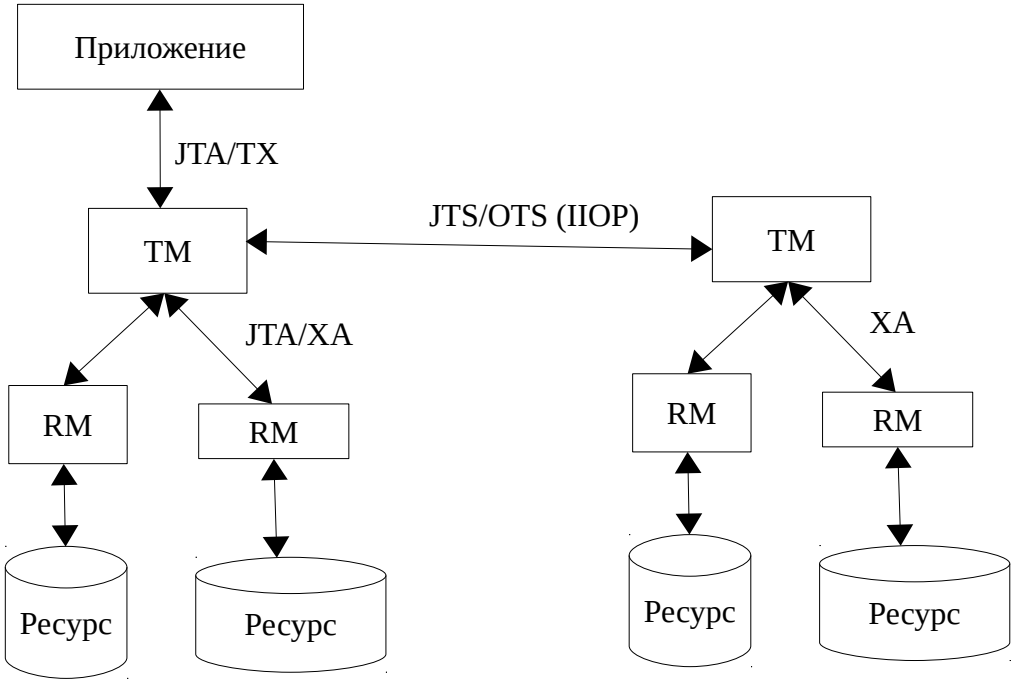
Менеджер транзакций (Transaction Manager, TM) - программный компонент JEE, отвечающий за управление распределенными транзакциями. Он создает их от имени приложения, синхронизирует, связывает их с менеджерами ресурсов (заключает с ними «сделки» или операции задействования) и дает последним команды на фиксацию или откат локальной транзакции. Именно TM отвечает за поддержание ACID транзакции. Менеджер транзакций использует в своей работе JEE-спецификации JTA и JTS.

JTA (Java Transaction API) – интерфейс на основе модели Open Group DTP, реализующий взаимодействие всех участников транзакций с TM. Он состоит из 3 частей: высокоуровневого клиентского TX-интерфейса для разметки транзакций (javax.transacton.UserTransaction); высокоуровневого серверного интерфейса для управления TM и прикладными клиентскими транзакциями (представлен интерфейсами TransactionManager, Transaction, Status и Synchronization пакета javax.transaction); низкоуровневого серверного XA-интерфейса, реализующего стандарт XA для взаимодействия TM и менеджеров ресурсов (javax.transaction.xa). В JEE7 используется спецификация JTA 1.2, куда, помимо прежних управления транзакциями и автоматической поддержки XA, добавлены возможности по использованию транзакций вне EJB, в т.ч. аннотации @Transactional @TransactionScoped для CDI MBean. Существуют встроенные и отдельные реализации JTA: встроенный TM сервера GlassFish, JBoss Transaction Manager, Atomikos и Bitronix JTA.

XA (eXtendedArchitecture) - стандарт Open Group, реализующий взаимодействие менеджеров (диспетчеров) ресурсов и распределенных транзакций в модели Open Group DTP. Диспетчер ресурсов может иметь любую природу, но должен поддерживать интерфейс XA. Сам XA использует протокол 2PC.



JTS (Java Transaction Service) - Java-реализация OMG OTS, спецификации службы, определяющей транзакционное поведение в разнородной распределенной среде, организованной по стандарту OMG CORBA. Служба JTS предназначена для использования в промышленных интеграционных (enterprise middleware) приложениях, давая возможность распространения транзакций между OTS-совместимыми TM по протоколу OMG IIOP. JTS поддерживает высокоуровневые интерфейсы JTA.



JTA-транзакции в EJB. Декларативное управление (CMT)

Транзакцией можно управлять явно в коде через API типа JTA. Это требует знаний + код, управляющий транзакциями, придется встраивать в бизнес-логику. Другой вариант — использовать метаданные, описывающие управление транзакциями. При этом бизнес-логика не трогается, а бизнес-код либо размечается транзакционными аннотациями @javax.ejb.TransactionAttribute, либо транзакционное управление описывается в XML-дескрипторе. С помощью этих аннотаций можно управлять транзакционным поведением отдельных методов, также EJB-контейнером реализовано транзакционное управление по-умолчанию. Такой тип транзакций называется CMT (Container Managed Transaction).

Жизненный цикл транзакции

Единицей работы UOW для CMT JTA в EJB-компоненте являются транзакционные операции внутри метода EJB. Транзакция начинается непосредственно до вызова метода и завершается сразу после его завершения. Т.е. CMT JTA оборачивают методы, если только те не помечены как полностью нетранзакционные.

При этом в UOW не попадают операции не подчиняющиеся ее транзакционному механизму, даже если они находятся в границах метода CMT EJB. По сути в UOW попадают только операции с объектами, оснащенными транзакционным механизмом (вроде EM, управляемых Entity, JMSContext) и не помеченные как нетранзакционные.

Все остальные операции не включаются в транзакцию, их взаимодействие с транзакцией будет подчинено общесистемным правилам (ограничен доступ к данным в транзакционных границах), а сама транзакция будет приостановлена во время их выполнения, а потом возобновлена.

Все вызовы транзакционных объектов (EJB, MDB, PC и т.д.) рассматриваются транзакцией на предмет успешности. Если все они прошли и не вызвали серьезных исключений, то соответствующие изменения фиксируются и становятся постоянными. Если же произошло системное исключение (System Exception) или прикладное исключение (Application Exception) с параметром rollback=true, выброшенное одним из методов UOW, то транзакция завершается досрочно и все изменения откатываются.

Решение commit/rollback принимается менеджером транзакций (TM) EJB-контейнера для CMT. А также клиентом, либо в коде самого транзакционного объекта для BMT.

При расширении транзакции, в нее могут вовлекаться методы других EJB-компонентов, при этом TM координирует транзакционные механизмы разных серверов (процессов) таким образом, чтобы локальные задачи образовывали единый UOW. Выполнение такого составного UOW является распределенной транзакцией и проводится по протоколу двухфазной фиксации (2PC). Распределенные транзакции автоматизированы в EJB и с т.з. разработчика без разницы, является транзакция локальной или распределенной.

Rem: При распространении транзакций от EJB к EJB другого сервера используются распределенные транзакции.

В EJB 3.2 (JEE7) появилась возможность включения в транзакции методов обратного вызова (callback) обработки событий жизненного цикла при помощи @TransactionalAttribute(REQUIRES_NEW), причем REQUIRES_NEW – единственное возможное значение.

Транзакционные границы

Границей транзакции в CMT EJB считается совокупность управляемых EJB-контейнером объектов, управляемых транзакцией (EM, PC, JMSContext и т.д.) затронутых в UOW, т.е. в границах метода. Эти объекты называют транзакционными.

Все остальное, даже затронутое в рамках метода EJB, в транзакционные границы не входит.

По ходу транзакции, ее границы расширяются за счет тех транзакционных объектов, методы которых были вызваны в прежних границах. Помимо EJB, в границы включаются и PC и EM и т.д. При завершении транзакции ее границы исчезают.

Transaction Propagation — расширение транзакции - расширение границ транзакции при

вовлечении в эту транзакцию новых объектов.

В качестве примера возникновения транзакционных границ можно рассмотреть тестовый метод <T>wrapInTx(Callable<T> task). Транзакция начинается вместе с потоком задачи и включает всю бизнес-логику метода. С завершением метода завершается и поток и транзакция. При удачном завершении транзакция фиксируется, т.е. изменения состояния становятся постоянными, в т.ч. PC сбрасывается в РБД. По ходу транзакции может произойти ошибка (выброс исключения в самом коде wrapInTx или в вызванных им методах). В зависимости от ошибки может произойти либо откат транзакции (rollback), либо нет. Таким образом обеспечивается атомарность.

Помимо потока транзакции, на транзакционные границы неявно влияют атрибуты транзакционного управления (метаданные в виде аннотаций или XML-дескриптора) и явно код JTA.

Политики расширения транзакции. Transaction Attributes

EJB-контейнер дает возможность автоматического управления транзакциями CMT при помощи транзакционных атрибутов. Эти атрибуты заданны метаданными - аннотациями или через XML-дескриптор. Атрибуты могут определять режим управления на уровне метода или класса. Режимы управления (атрибуты):

- NOT_SUPPORTED
- SUPPORTS
- REQUIRED (по-умолчанию)
- REQUIRES_NEW
- MANDATORY
- NEVER

Аннотация вида @TransactionAttribute(TransactionAttributeType.xxx), где xxx – значение атрибута Enum - TransactionAttributeType, позволяет в deploy-time определять транзакционный режим управления как метода, так и класса EJB в целом.

Аннотации уровня метода приоритетнее аннотаций уровня класса, а теги XML-дескриптора <trans-attribute> приоритетнее аннотаций. Поэтому аннотации уровня класса удобно использовать как поведение по-умолчанию. Есть и встроенное транзакционное поведение по-умолчанию EJB-контейнера: если не указано никаких метаданных, то неявно используется атрибут TransactionAttributeType.REQUIRED.

Описание транзакционных атрибутов

	Клиент вовлечен в транзакцию?	Действие
NOT_SUPPORTED	Да	Клиентская тр. приостанавливается до завершения вызова

	Нет	Выполнение метода вне транзакций
SUPPORTS	Да	Метод вовлекается в клиентскую транзакцию
	Нет	Выполнение метода вне транзакции
REQUIRED	Да	Метод вовлекается в клиентскую транзакцию
	Нет	EJB-контейнер запустит новую транзакцию на время вызова
REQUIRES_NEW	Да	Клиентская тр. приостанавливается до завершения метода, EJB-контейнер запускает новую транзакцию в границах метода
	Нет	EJB-контейнер запустит новую транзакцию в границах метода
MANDATORY	Да	Метод вовлекается в клиентскую транзакцию
	Нет	Выброс клиенту javax.ejb.EJBTransactionRequiredException
NEVER	Да	Выброс клиенту javax.ejb.EJBException
	Нет	Выполнение метода вне транзакции

NOT_SUPPORTED

Метод приостанавливает транзакцию вызвавшего его клиента. Т.е. вызывающая метод транзакция замораживается и не распространяется, пока метод не завершится. В результате код метода на транзакцию не влияет, равно как и она не влияет на его результат. После завершения метода транзакция возобновляется. Типичный пример использования — подтверждение приема/обработки MDB сообщения от JMS-провайдера, которое выполняется при успешном завершения onMessage() MDB-компонента и не должно откатываться.

SUPPORTS

Метод включается в транзакционные границы вызывающего клиента, при этом сам метод не должен быть частью другой транзакции, равно как и взаимодействовать с чужими транзакционными объектами. Если клиент не принадлежит никакой транзакции, то и метод не включается ни в какую транзакцию. Такой атрибут применяется обычно для операций чтения, где части клиентов необходима транзакция с поддержкой высокого уровня изоляции.

REQUIERD

Метод требует вовлечения в транзакцию. Если клиент не связан ни с какой транзакцией, то порождается временная транзакция, в которую включается метод и которая живет до его завершения. Если вызывающий клиент находится в границах некоторой транзакции, то метод также вовлекается в эту транзакцию, при этом, если метод приводит к откату транзакции, то также выбрасывается исключение javax.transaction.RollbackException, которое уведомляет клиента о произведенном откате. Данный атрибут является атрибутом по-умолчанию и применяется там, где метод изменяет данные, но неизвестно, будет ли он вызываться клиентом, запустившим собственную транзакцию. Примером можно считать методы EM, которые по-

умолчанию работают только в рамках транзакции.

REQUIRES_NEW

Метод включается в новую временную транзакцию, которая живет до его завершения. В границы этой транзакции входят объект-хозяин метода и транзакционные объекты, связанные с методом. Если клиент был вовлечен в какую-либо транзакцию, то она приостанавливается на время работы метода. Атрибут удобен для транзакционных задач, выполнение которых взаимонезависимо с результатами клиентских транзакций. Как пример — выполнение журналирования, которое должно выполняться независимо от результата клиентской транзакции, в свою очередь, сбой записи сообщения в журнал не должен повлиять на клиента.

MANDATORY

Метод должен обязательно включаться в транзакцию клиента. Если клиент не вовлечен в транзакцию, то будет выброшено runtime-исключение javax.ejb.EJBTransactionRequiredException. Такой атрибут используется для методов, которые являются составной частью более крупных клиентских задач, а исключение сообщает клиенту, что метод не может выполняться самостоятельно и вне транзакционных границ. Пример — снятие денег со счета, которое не имеет самостоятельного смысла вне более общих операций.

NEVER

Метод должен выполняться вне какой-либо транзакции. Если клиент вовлечен в транзакцию, то будет выброшено runtime-исключение javax.ejb.EJBException. Такой атрибут может использоваться для методов, которые работают с нетранзакционными ресурсами, например, с нежурналируемой файловой системой, чтобы указать клиенту на невозможность отката при сбое.

Нетранзакционные EJB (nontransactional EJB)

Службы без состояния (Stateless) обычно используются для вспомогательных операций в read-only режиме чтения из РБД. Объекты EJB, реализующие такие методы, не обязаны поддерживать ACID транзакции, поскольку изменение состояния РБД никак не сказывается ни на их состоянии, ни на выполнении их методов. Кроме того, включение бина в транзакционные границы может заблокировать используемые в транзакции записи РБД. Это дорого с вычислительной точки зрения, поэтому имеет смысл выключать из транзакции лишние бины. Делается это при помощи значения TransactionAttributeType.NOT_SUPPORTED. Это положительно влияет на производительность и доступность служб.

Rem: специальной транзакционной изоляции EJB-компонентам не требуется. Транзакционная изоляция не нужна SFSB, SLSB и MDB, т.к. объекты этих EJB-компонентов

и так блокируются EJB-контейнером при выполнении методов для обеспечения потокобезопасности. К тому же SLSB, MDB, @Singleton допускают транзакцию только в границах метода и потому она не влияет на общую политику их изоляции.

Transaction & MDB

Клиент взаимодействует с MDB только через посредника, поэтому MDB ничего не знает о клиентской транзакции. В результате в MDB надо либо мутить свою транзакцию, либо не использовать ее вовсе. Отсюда вытекают 2 режима транзакционного управления:

1. NOT_SUPPORTED – не использовать транзакции
2. REQUIRED — использовать свою транзакцию (режим по-умолчанию)

Transaction & Endpoint WS

Поскольку в JEE7 распределенные транзакции для SOAP WS не стандартизированы, то и расширение клиентской транзакции на конечные точки WS не предусмотрено. Если вызов WS был сделан из транзакционного метода клиента, то сбой на сервере вызовет исключение, которое может привести к откату транзакции как на сервере, так и к откату клиентской транзакции. Но если серверный метод обработки WS отработал штатно, а сбой произошел позже, например, при сетевой передаче, то клиентская транзакция откатится, но отката серверной не произойдет.

Поэтому для endpoint транзакционный режим управления MANDATORY не используется.

Transaction & EM

В спецификации EJB 3 настоятельно рекомендуется, чтобы любое обращение к EM шло в рамках транзакции. Поэтому когда обращение к CM EM идет через CMT-объект допускаются только следующие режимы: REQUIRED, REQUIRES_NEW, MANDATORY. Существует исключение при использовании SFSB и ES EM.

Пример расширения транзакции (Transaction Propagation)

Для демонстрации расширения транзакций разбирается простой пример из серии BlackJack.

1. Перевод $100 со счета A на счет B
2. Проверка балансов счетов
3. Провоцирование отката транзакции

В @Stateless bankBean назначается подходящее транзакционное управление для методов.

```
@TransactionAttribute(TransactionAttributeType.SUPPORTS)
@Override
BigDecimal getBalance(long accountID) {...}
```

getBalance(...) – read-only метод. Сам по себе read-only метод не требует для работы ACID-свойств и отдельно его можно запускать без транзакции. Но если getBalance(...) запускается в рамках клиентской транзакции, в которой связанные с ним данные (по балансу) изолированы, то его надо включать в состав этой транзакции, для доступа к еще не зафиксированным изменениям. Такое расширение транзакции и обеспечивает режим управления SUPPORTS.

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
@Override
void transfer(long accountIDFrom, long accountIDTo, BigDecimal amount)
```

transfer(...) - write-метод, поэтому требует для работы ACID-свойств. Т.е. либо надо распространять на этот метод клиентскую ACID-транзакцию, либо создавать для него новую временную транзакцию. Такое поведение и обеспечивает режим управления REQUIRED.

JUnit-тестирование распространения клиентских транзакций на методы bankBean:

```
@Test
public void transferRetainsIntegrity() throws Throwable
{
   ...
   this.executeInTx(new CheckBalanceOfAccountTask(xxxId, expectedInitialXxx),
         new CheckBalanceOfAccountTask(blackjackId, expectedInitialBlackjack));
  final BigDecimal oneHundred = new BigDecimal(100);
  bank.transfer(xxxId, blackjackId, oneHundred);
  this.executeInTx(new CheckBalanceOfAccountTask(xxxId,
    expectedInitialXxx.subtract(oneHundred)), new
CheckBalanceOfAccountTask(blackjackId,
    expectedInitialBlackjack.add(oneHundred)));
...
```

Вначале 2 проверки баланса getBalance(...) — игрока и BlackJack, которые благодаря режиму SUPPORTS включаются в клиентскую транзакцию, организованную методом-оберткой SLSB через executeInTx(...), где они запускаются параллельно в разных потоках. Затем делается перевод $100 со счета игрока xxx на счет Blackjack, причем клиентская транзакция отсутствует, но благодаря режиму REQUIRED метод transfer(...) создает свою и выполняется в ней. Далее вновь проверяется измененный баланс в единой транзакции.

Затем тестируется искусственно-вызванный откат клиентской транзакции.

```
...
boolean gotExpectedException = false;
final Callable<Void> transferTask = new Callable<Void>() {
  @Override
  public void call() throws Exception {
    bank.transfer(xxxId, blackjackId, oneHundred);
    return null;
  }
};
  try {
    this.executeInTx(transferTask, new CheckBalanceOfAccountTask(xxxId,
expectedInitialXxx.subtract(oneHundred).
    subtract(oneHundred)), new CheckBalanceOfAccountTask(blackjackId,
expectedInitialBlackjack.add(oneHundred).add(oneHundred)),
    ForcedTestExceptionTask.INSTANCE);
  }
  // Expected
  catch (final ForcedTestException fte) {
    gotExpectedException = true;
  }
  Assert.assertTrue("Did not receive expected exception as signaled from the test; was not
rolled back", gotExpectedException);
  this.executeInTx(new CheckBalanceOfAccountTask(xxxId,
expectedInitialXxx.subtract(oneHundred)),
    new CheckBalanceOfAccountTask(xxxId, expectedInitialBlackjack.add(oneHundred)));
```

Создается callable-задача transferTask в который завертывается еще один перевод transfer(...) $100 со счета игрока Xxx на счет Blackjack. Эта задача transferTask вместе с проверкой измененных балансов и выбросом исключения вновь объединяется в UOW при помощи executeInTx(...) и режимов SUPPORTS и REQUIRED, расширяющих клиентскую транзакцию. Внутри UOW все задачи выполняются параллельно, но несмотря на успешные проверки измененного баланса, все заканчивается неудачей из-за выброса в какой-то момент исключения callable-Enum'ом ForcedTestExceptionTask.INSTANCE. Это исключение вызовет откат транзакции. После этого опять вызывается проверка баланса, доказывающая откат транзакции.

Rem: транзакция не распространяется при асинхронном вызове метода @Asynchronous

JTA-транзакции в EJB. Явное управление (BMT)

Явное или программное управление транзакциями рекомендуется только в случае, когда не хватает автоматического неявного режима, обеспечиваемого EJB-контейнером и метаданными. Последнее имеет ясный и строгий описательный характер (аннотации или xml), что уменьшает риск ошибки и разделяет бизнес-логику и транзакционное управление. Кроме того в модели CMT EJB-контейнер автоматически создает транзакционные границы, включающие код метода и автоматом распространяет транзакцию на другие транзакционные объекты. Транзакции, явно управляемые из кода приложения, называются BMT (Bean Management Transaction). В отличие от CMT-

модели, границы BMT находятся внутри кода метода или методов (для SFSB) .

Вообще явное транзакционное управление осуществляется через службу JTS, которая предоставляет полный и мощный API, позволяющий работать напрямую с менеджерами транзакций и менеджерами ресурсов (JDBC, JMS и т.д.). Служба JTS, предназначенная для кросс-платформенного взаимодействия (IIOP) транзакционных механизмов сложна и требует тщательного контроля за транзакционными границами. Поэтому обычно применяется более простой интерфейс JTA. Он реализует модель распределенной транзакции DTP, созданную организацией Open Group. JTA находится в пакете javax.transactions и состоит из 3-х частей:

- высокоуровневый клиентский TX-интерфейс для разметки транзакций (представлен интерфейсом javax.transacton.UserTransaction)
- высокоуровневый серверный интерфейс для управления TM и прикладными клиентскими транзакциями (представлен интерфейсами TransactionManager, Transaction, Status and Synchronization пакета javax.transaction)
- низкоуровневый серверный XA-интерфейс, реализующий стандарт XA для взаимодействия TM и менеджеров ресурсов (javax.transaction.xa)

Клиентский интерфейс доступен для EJB и рекомендован для использования клиентскими приложениями. XA-интерфейс используется EJB-контейнером для координации транзакций с ресурсами (РБД и т.д.).

Явная транзакция описывается для приложений интерфейсом JTA javax.transacton.UserTransaction. Этот интерфейс доступен для клиента через JNDI. Для самого приложения также доступно получение экземпляра через внедрение @Resource и метод getUserTransaction() объекта EJBContext . Последнее можно применять при наличии ссылки на SessionContext или MessageDrivenContest, если внедрение по каким-то соображениям нежелательно. Явная транзакция доступна только в BMT-объектах (сессионные и MDB, помеченные @TransactionManagement(TransactionManagementType.BEAN)).

```
Ex: пример получения и использования клиентом UserTransaction через JNDI
…
Context ctx = new InitialContext();
UserTransaction utx = ctx.lookup(“java:comp/UserTransaction”);
utx.begin();
…
utx.commit();
```

Для явного управления собственной транзакцией допускаются только аннотированные @javax.ejb.TransactionManagement с атрибутом value, равным javax.ejb.TransactionManagementType.BEAN сессионные бины и MDB. Эти бины часто называют BMT-бинами. При этом безотносительно способа получения доступа к BMT-бину, будет использовать один и тот же экземпляр UserTransaction, т.е. для каждого BMT-бина создается свой единственный экземпляр UserTransaction.

Rem: при попытке получить экземпляр UserTransaction в CMT-бине, будет выброшено исключение java.lang.IllegalStateException.

Ex: пример управления собственной транзакцией в BMT через клиентский TX-интерфейс

```
@TransactionManagement(TransactionManagementType.BEAN)
@Stateful
public class ExEJB implements ExEJBRemote {
  @PersistenceUnit(unitName="ExPU")
  private EntityManagerFactory emf;
  @Resource
  private SessionContext context;

  @Override
  public void test() {
    Bar bar = new Bar();
    bar.setId(33);
    bar.setName("bar33");
    try {
      UserTransaction utx = context.getUserTransaction();
      utx.begin();
      EntityManager em = emf.createEntityManager();
      em.persist(bar);
      utx.commit();
      em.close();
      emf.close();
      System.out.println(cache.contains(Bar.class, bar.getId()));
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

Rem (к примеру выше): EM либо должен создаваться внутри транзакции, либо присоединяться к ней методом EM joinTransaction(), т.к. при создании PC EJB-контейнеру нужна активная JTA-транзакция. Если ее нет, то синхронизации с РБД не будет.

Ex: пример управления собственной транзакцией в JTA BMT через серверный TM-интерфейс (GlassFish)

```
@Singleton
@TransactionManagement(TransactionManagementType.BEAN) ExEJB {
// Получение серверного TransactionManager в GlassFish по вендор-специфическому имени
@Resource(mappedName = "java:appserver/TransactionManager")
TransactionManager txManager;

/* При обратном вызове @PostConstruct-метода EJB-контейнером, создается новая собственная транзакция, т.к. такой вызов идет вне транзакционного окружения. При вызове метода другим EJB или приложением используется уже имеющаяся активная транзакция, а если ее нет, то создается новая. */
```

```
@PostConstruct
  public void initialize() throws Exception
  {
    final Transaction tx = txManager.getTransaction();
    final boolean startOurOwnTx = tx == null;
    // если активной транзакции нет (@PostConstruct вызов или завершение старой тр.) – создать
новую тр.
    if (startOurOwnTx){
      txManager.begin();
    }
    try{
     //... работа в транзакции: логика, вызовы ejb, em ...
    }
    catch(final Throwable t) {
      // любое исключение в свежесозданной тр. (@PostConstruct вызов) = ее откату при завершении
      if(startOurOwnTx) {
        txManager.setRollbackOnly();
      }
    }
    finally
    {
      // Разметка конца свежеиспеченной транзакции — либо фиксация (сигнал для EM на сброс
изменений в РБД и т.д.), либо откат, если ранее тр. была помечена setRollbackOnly()
      if (startOurOwnTx){
        txManager.commit();
      }
    }
  }
}
}
```

Rem (к примеру выше): javax.transaction.TransactionManager – интерфейс, предназначенный для Application Server и позволяющий полноценно управлять транзакциями, в т.ч. предоставляя прямой доступ к объекту javax.transaction.Transaction, через который можно регистрировать и синхронизировать XA-ресурсы.

Rem (к примеру): можно рассматривать этот пример как решение задачи подключения @PostConstruct – метода к транзакционному окружению.

Жизненный цикл и транзакционные границы BMT

В отличие от CMT, начало и конец BMT задаются явно, вызовом менеджера транзакций через TM или TX интерфейс. В UOW также попадают только транзакционные операции (т.е. операции с транзакционными объектами вроде EM, JMSContext, управляемых Entity и т.п.).

В транзакционные границы попадают только транзакционные объекты. Все остальное, даже затронутое между вызовами начала и конца транзакций, транзакцией не управляется и взаимодействует с ней как внешняя часть системы (с изоляцией

транзакционных данных). При этом надо учитывать особенности EJB-компонентов, в которых определяется эта BMT.

В SLSB/BMT транзакция должна протекать в границах метода, т.к. следующий клиентский запрос может быть обработан другим экземпляром.

В Singleton/BMT транзакция также должна протекать в границах метода, т.к. его состояние доступно всем клиентам и размазывание транзакции может приводить к сбоям в совместном использовании.

В SFSB/BMT допускается взаимодействие с транзакцией, распределенной по множеству его методов. Т.е. транзакция SFSB/BMT может начинаться в одном его методе и заканчиваться в другом т.к. клиент работает с одним и тем же экземпляром SFSB/BMT в границах одной сессии.

Если клиент вовлечен в некоторую чужую транзакцию, то при обращении к методу BMT она приостанавливается до окончания вызова этого метода. При этом неважно, порождалась ли в методе BMT собственная транзакция или же была подхвачена активная транзакция в одном из предыдущих методов.

BMT не распространяется на другие BMT-объекты (прерывается при окончании внутренней транзакции) из-за отсутствия поддержки вложенных транзакций.

Rem: не рекомендуется размазывать управление транзакцией по нескольким методам BMT (?), т.к. в результате можно получить ошибку, подвисшую транзакцию или даже потерю ресурса. Хорошей практикой является разрешать методу самостоятельно определять транзакционные границы.

Rem: короткоживущая BMT может включать вызов CMT-метода (?). Но для длинных BMT в SFSB это невозможно, т.к. ж.ц. CMT-объекта (SLSB/TS PC и т.п.) определяется одним клиентским вызовом.

Пример расширения BMT

В границы транзакции при расширении вовлекаются только транзакционные объекты.

Ex: транзакционные и нетранзакционные объекты в SFSB

```
@TransactionManagement(TransactionManagementType.BEAN)
@Stateful(name="ExEJB")
public class TestSFSB implements ExRemote {
    @PersistenceUnit(unitName="TestPU")
    private EntityManagerFactory emf;
    private String field;
    @EJB(beanName="")
    private ExRemote2 ejb2;
   @Resource
```

```
  private SessionContext context;

  …
  @Override
  public void test() {
   Bar bar = new Bar();
   bar.setId(33);
   bar.setName("bar33");
   try {
     UserTransaction utx = context.getUserTransaction();
     utx.begin();
     EntityManager em = emf.createEntityManager();
     em.merge(bar);
     utx.commit();
     bar.setName("bar34");
     this.field = “init field”;
     ejb2.setField(“init field");
     utx.begin();
     em.joinTransaction();
     em.merge(bar);
     this.field = “modified field”;
     ejb2.setField("modified field");
     utx.rollback();
     em.close();
     em = emf.createEntityManager();
     emf.close();
    } catch (Exception ex) {
     ex.printStackTrace();
    }
  }
}
```

В данном примере используется BMT в SFSB. В первой транзакции создается EM, который сливает Enitty-объект bar с персистентным слоем, фиксация транзакции приводит к обновлению РБД. Во второй транзакции подключается тот же EM, проводится новое слияние с PC измененного bar, меняется поле field бина, также меняется поле внедренного EJB ejb2, но вместо фиксации проводится откат. В результате в РБД ничего не изменится, PC слой также не будет меняться, поскольку EM является транзакционным объектом. Но поле field и объект ejb2 будут изменены, несмотря на откат, т.к. они не транзакционные.

MDB & BMT

MDB также может управлять транзакцией, т.е. быть BMT-бином. При этом границы транзакции лежат внутри метода, т.к. MDB не поддерживает сессию.

В случае MDB/BMT потребляемые им сообщения не включаются в транзакционные границы. Т.е. при сбое транзакции в методе onMessage() автоматической переотправки сообщения не будет, т.к. JMS-провайдер не узнает о сбое. В случае MDB/CMT

сообщение включается в границы транзакции и JMS-провайдер может повторно отправить сообщение при откате транзакции.

Однако есть еще механизм подтверждения успешного потребления сообщений, который оповещает JMS-провайдера об успехе/неудаче обработки посланного им сообщения. Этот механизм автоматически отправляет «успех», если метод onMessage() отработал штатно и «неудача», если onMessage() выбросил непроверяемое RuntimeException. В ответ на «неуд» JMS-провайдер может повторно отправить сообщение. Поэтому при необходимости переотправки сообщения в случае сбоя его обработки, рекомендуется выбрасывать в onMessage() непроверяемое EJBException.

Rem: вендоры используют собственные механизмы определения числа повторных попыток отправки неподтвержденных сообщений для BMT и NotSupported MDB. JMS-провайдер также может определить зону «мертвых сообщений», куда будут помещаться те из них, что превысили планку попыток повторной отправки. Эту мертвую зону можно затем обрабатывать вручную.

Все остальные объекты кроме сообщений включаются в транзакционные границы onMessage().

Heuristic Decisions

Транзакции обычно управляются транзакционным менеджером (TM), который обеспечивает им ACID-характеристиками, взаимодействуя с несколькими менеджерами ресурсов БД, EJB, серверов и т.д. Поскольку менеджеры ресурсов обладают локальными транзакционными механизмами, требуется согласование всей распределенной транзакции для сохранения ACID-свойств. Для этого сервер использует протокол двухфазной фиксации (2PC). При этом TM выступает в роли посредника между транзакционными ресурсами и обеспечивает согласованную фиксацию или откат локальных транзакций, опрашивая их менеджеры ресурсов.

Rem: 2PC поддерживается не всеми EJB-серверами

Heuristic Decision (эвристическое решение) — ситуация, в которой один или более менеджеров ресурсов принимает одностороннее решение о фиксации или откате своей локальной транзакции вопреки решению менеджера (диспетчера) распределенной транзакции.

Это может произойти на втором этапе 2PC в одном из ресурсов из-за сбоя, неполучения команды из-за потери сетевого соединения с распределенным диспетчером, в результате одностороннего отката из-за перегрузки и т.д. HD может нарушить ACID-характер распределенной транзакции. Интерфейс UserTransaction выбрасывает различные исключения, связанные с HD.

Rem: кроме того может сбоить сам распределенный диспетчер транзакций, например из-за остановки приложения. В этом случае локальные транзакции остаются в подвешенном

состоянии и должны (обычно по таймауту) самостоятельно сделать откат, чтобы не блокировать доступ к ресурсам. Т.к. обычно локальные ресурсы ведут журналы транзакций (Durable, надежность), распределенный диспетчер может завершить транзакцию после возобновления работы.

UserTransaction

Интерфейс javax.transaction.UserTransaction обязателен для реализации EJB-контейнером (остальная часть JTA и JTS, согласно спецификации, необязательна для реализации EJB-контейнером).

Методы UserTransaction:

- begin()
- commit()
- int getStatus()
- rollback()
- setRollbackOnly()
- setTransactionTimeout(int)

begin() - метод, который связывает вызвавший его поток с новой транзакцией, которая может быть затем распространена на другие транзакционные объекты, поддерживающие существующие транзакции. Метод может выбросить 2 исключения: java.lang.IllegalStateException означает, что метод begin() вызван в рамках текущей активной транзакции (ее надо закончить перед запуском новой, т.к. вложенные не поддерживаются); org.omg.CORBA.SystemException означает, что TM (EJB-контейнер) столкнулся с непредвиденной ошибкой состояния.

Rem: в JEE7 не поддерживаются вложенные транзакции. Повторный вызов begin() в рамках уже активной транзакции вызовет исключение NotSupportedException (?)

commit() — метод фиксации транзакции, который вызывается в потоке, связанном с текущей активной транзакцией. После выполнения, поток отвязывается от транзакции. Выбрасываются: IllegalStateException — метод commit() вызван в потоке, не связанном ни с какой активной транзакцией; SystemException — TM (EJB-контейнер) столкнулся с непредвиденной ошибкой состояния; TransactionRollbackException — транзакция откатывается, хотя должна быть зафиксирована, это может произойти, если один из локальных ресурсов не смог обновиться или ранее был вызван метод setRollbackOnly(); HeuristicRollbackException — один из локальных ресурсов принял эвристическое решение об откате своей локальной транзакции; HeurisitcMixedException – часть локальных ресурсов приняла эвристическое решение об откате, а другая о фиксации.

rollback() - метод откатывает активную транзакцию, связанную с вызывающим потоком. Выбрасываются: SecurityException — текущий вызывающий поток не имеет права делать откат; SystemException — TM (EJB-контейнер) столкнулся с непредвиденной ошибкой состояния; IllegalStateException — с вызывающим потоком не связано ни

одной активной транзакции.
(?) Как выполняется метод rollback – выбрасывается ли исключение и какое (?)

setRollbackOnly() - метод помечает текущую активную транзакцию на откат вне зависимости от того, будет ли она завершена успешно или нет. Вызывать метод могут BMT (?) и клиенты, связанные с транзакцией. Выбрасываются: IllegalStateException — с вызывающим потоком не связано ни одной активной транзакции; SystemException — TM (EJB-контейнер) столкнулся с непредвиденной ошибкой состояния. Метод удобен при использовании в BMT вложенных вызовов CMT-методов (где используется только флаг отката через SessionContext) для единообразия транзакционной логики . Кроме того флаг дает возможность обработки отката в методах верхнего уровня при возникновении сбоя на нижних (цепочка генерации и обработки исключений).

setTransactionTimeout(int) — метод назначает время жизни транзакции в секундах. При вызове с 0-аргументом и по-умолчанию TM (EJB-контейнер) назначает время жизни по-умолчанию. Метод должен вызываться после begin(). Выбрасывается: IllegalStateException — с вызывающим потоком не связано ни одной активной транзакции; SystemException — TM (EJB-контейнер) столкнулся с непредвиденной ошибкой состояния.

getStatus() — метод выдает числовой код статуса. Коды определены в javax.transaction.Status. При помощи статуса можно отслеживать состояние транзакции, связанной с объектом UserTransaction. Выбрасываются: SystemException — TM (EJB-контейнер) столкнулся с непредвиденной ошибкой состояния.

Status

Интерфейс javax.transaction.Status содержит только int-константы, которые для UserTransaction имеют следующий смысл:

- STATUS_ACTIVE = с объектом UserTransaction связана активная транзакция, TM уже запустил транзакцию, но еще не начал процесс 2PC. Заблокированные транзакции все еще считаются активными.
- STATUS_COMMITED = с объектом UserTransaction связана активная транзакция, хотя ее фиксация уже состоялась. Это возможно в ситуации эвристического решения (один или более локальный ресурс самовольно сделал откат), иначе бы статус был STATUS_NO_TRANSACTION.
- STATUS_COMMITING = с объектом UserTransaction связана активная транзакция, TM принял решение о фиксации, но еще не успел ее доделать.
- STATUS_MARKED_ROLLBACK = с объектом UserTransaction связана активная транзакция, помеченная (возможно методом setRollbackOnly()) на откат.
- STATUS_NO_TRANSACTION = не существует активной транзакции, связанной с объектом UserTransaction. Возможно транзакция была успешно зафиксирована, возможно не было создано ни одной транзакции. Этот статус может использоваться для предотвращения выброса IllegalStateException.

- STATUS_PREPARED = с объектом UserTransaction связана активная транзакция, подготовленная к фиксации, т.е. выполнена 1-я фаза 2PC.
- STATUS_PREPARING = с объектом UserTransaction связана активная транзакция, идет подготовка к фиксации, т.е. выполняется 1-я фаза 2PC.
- STATUS_ROLLBACK = с объектом UserTransaction связана активная транзакция, хотя ее откат уже состоялся. Вероятно произошло эвристическое решение (т.е. один или более локальный ресурс самовольно зафиксировал свои локальные изменения), иначе бы статус был STATUS_NO_TRANSACTION.
- STATUS_ROLLING_BACK = с объектом UserTransaction связана активная транзакция, выполняется ее откат.
- STATUS_UNKNOWN = с объектом UserTransaction связана активная транзакция, но ее статус временно не определяется. Почти наверное следующий запрос getStatus() выдаст уже определенный результат.

Методы rollback EJB Context

BMT-бины используют интерфейс UserTransaction, CMT-бины не могут использовать объекты UserTransaction и получать их через JNDI ENC или EJBContext, вместо этого им доступны транзакционные методы интерфейса javax.ejb.EJBContext: setRollbackOnly() и getRollbackOnly(), взаимодействующие с активной (контейнерной) транзакцией. Метод setRollBackOnly() помечает транзакцию на откат и накладывает вето на ее фиксацию для любых участников, включая сам EJB-контейнер.

Rem: методы setRollbackOnly() и getRollbackOnly() могут вызываться только в рамках CMT и когда вызывающий метод отмечен атрибутом: REQUIRED, REQUIRES_NEW или MANDATORY. В противном случае контейнер будет возбуждать исключение java.lang.IllegalStateException.

Метод getRollbackOnly() выдает true, если транзакция уже помечена на откат. Этот метод удобно использовать для оптимизации выполнения транзакции. Пометить транзакцию на откат может сам EJB-контейнер в случае выброса некоторых исключений. Можно перехватывать эти исключения внутри транзакционных границ и проверять при помощи getRollbackOnly(), стоит ли выполнять код дальше. Если транзакция помечена на откат, то смысла нет. Если же исключение не привело к откату, то можно переделать сбойный участок или самостоятельно пометить транзакцию на откат при помощ setRollbackOnly().

Rem: альтернативой методам setRollbackOnly()/getRollbackOnly() является выброс исключений.

Rem: BMT-бины не используют транзакционные методы EJBContext, вместо них используются методы UserTransaction.

JTA транзакции в EJB. Транзакционные SFSB (TSFSB)

Взаимодействие сессионных EJB с РБД зависит от типа бина. SLSB без диалогового

состояния должен обновлять данные сразу, до завершения метода. В случае SFSB появляется возможность включения метода в транзакцию, в которой участвуют другие методы. И тогда фиксация или откат изменений в РБД будут связаны с этой транзакцией. Типичным примером сессионного кэширования данных для последующей отправки единого запроса в РБД является реализация корзины покупателя в интернет-магазине.

Интерфейс javax.ejb.SessionSynchronization облегчает кэширование изменений для последующей их записи в РБД.

Transactional SFSB (TSFSB) — SFSB, реализующий интерфейс javax.ejb.SessionSynchronization. Этот интерфейс предоставляет callback-методы информирования SFSB о его вовлеченности в транзакцию и прохождения ее этапов. Кроме того, TSFSB получает новое состояние – Transaction Method-Ready State.

Схема T-M-R состояния

Методы SessionSyncrhonization:

- afterBegin() - callback, запускаемый перед началом JTA транзакции
- beforeCompletion() - callback, запускаемый перед фиксацией транзакции
- afterCompletion(boolean commited) — callback, срабатывающий после завершения транзакции (true – после фиксации, false – после отката)

Если метод SessionSyncrhonization вызывается и работает вне транзакционного контекста, то он ведет себя как обычный метод SFSB, т.е. находится в M-R состоянии. Но если вызов был осуществлен в транзакции или метод породил собственную, то бин переходит в T- M-R состояние. Методы SessionSyncronization вызываются только в T-M-R состоянии.

Переход в T-M-R состояние

При обращении в рамках транзакции к транзакционному методу TSFSB, последний становится ее частью. И до передачи вызова этого метода от EJB-объекта (proxy) к EJB-экземпляру, происходит вызов callback-метода afterBegin() из SessionSyncrhonization, который также присоединяется к той же транзакции. В этом методе, в границах транзакции, могут быть сосредоточены операции чтения из РБД и сохранение данных в полях экземпляра, т.е. кэширование.

T-M-R

Как только выполнился метод afterBegin(), все клиентские запросы этой же транзакции идут напрямую в экземпляр TSFSB.

Когда SFSB, в частности и TSFSB, попадает в границы транзакции, все его методы

должны выполняться только в этой же транзакции. Не допускается выполнение метода в другом транзакционном контексте, в частности, вне транзакций. При этом не важно, был ли клиентский запрос сделан из другой транзакции или же метод SFSB пытался создать собственную транзакцию - в итоге будет выброшено исключение. Т.о. будет ошибочным вызов REQUIRES_NEW, NEVER, NOT_SUPPORTED - методов для SFSB, связанного с активной транзакцией. Также не допускается удаление экземпляра SFSB в границах активной транзакции, т.е. вызов в активной транзакции @Remove-метода приведет к выбросу исключения.

Когда транзакция завершается фиксацией, экземпляр TSFSB оповещается об этом вызовом вовлеченного в нее же метода beforeCompletion(), где можно записать в РБД весь кэш. Если же транзакция завершается откатом, то метод beforeCompletion() не вызывается, чтобы не делать бессмысленную работу.

После завершения транзакции, независимо от ее успешности всегда вызывается callback-метод afterCompletion(boolean). Вызов идет с параметром true в случае фиксации транзакции, т.е. при вызове до этого beforeCompletion(). А в случае отката — с параметром false, в этом случае можно восстановить исходное значение состояния экземпляра TSFSB.

JTA транзакции в JEE7. CMT-компоненты, отличные от EJB

В JEE7 к транзакционным объектам, помимо EJB, были добавлены CDI MBean, следовательно, Servlet, JAX-RS, JAX-WS (?)

Сделано это при помощи стандартного (начиная с JEE6) фрэймворка перехвата: определенный в JEE7 перехватчик перехватывает вызовы специально-помеченных методов указанных объектов и оборачивает их в JTA-транзакцию, которая управляется контейнером (CMT).

Для пометки транзакционных методов служит аннотация @javax.transaction.Transactional. Она может применяться к классу целиком или к его отдельным методам (приоритетнее) и содержит следующие атрибуты:

- value = Transactional.TxType {REQUIRED, REQUIRED_NEW, SUPPORTS, NOT_SUPPORTED, MANDATORY, NEVER} – режим расширения транзакции, аналогичный @TransactionAttribute CMT/EJB. По-умолчанию используется REQUIRED.
- rollbackOn = Class[] — классы исключений, которые должны вызывать откат транзакций (имеет смысл прикладных неоткатывающих исключений @ApplicationException в CMT/EJB)
- dontRollbackOn = Class[] — классы исключений, которые не должны вызывать откат транзакций (имеет смысл системных и прикладных откатывающих исключений CMT/EJB)

Rem: атрибут dontRollbackOn приоритетнее rollbackOn, что также можно использовать при одновременном аннотировании на уровнях класса и методов.

Rem: перехватчик для @Transactional не воспринимает метаданные @ApplicationException и поэтому переопределение реакции на исключения необходимо делать в атрибутах @Transactional.

Откат в ответ на все исключения:

```
@Transactional(rollbackOn={Exception.class})
```

Пометка системного (непроверяемого) исключения IllegalStateException как прикладного без отката:

```
@Transactional(rollbackOn={IllegalStateException.class})
```

Откатывать в ответ на все исключения SQL за исключением предупреждений:

```
@Transactional(rollbackOn={SQLException.class},
dontRollbackOn={SQLWarning.class})
```

Также для CDI MBean может применяться аннотация @TransactionScoped, которая задает жизненный цикл внедряемых объектов в пределах транзакции.

```
Ex: пример транзакционной разметки JTA CMT в CDI MBean

@TransactionScoped
public class TxScopedBean {
   public int number;

   public int getNumber() {return number;}
   public void setNumber(int number) {this.number = number;}
}

@RequestScoped
public class ExampleBean {

   @Inject
   private TxScopedBean tXScopedBean;

   @Transactional
   public void foo() {
      tXscopedBean.setNumber(1);
   }

   @Transactional
   public void bar() {
      System.out.print(tXscopedBean .getNumber());
   }
```

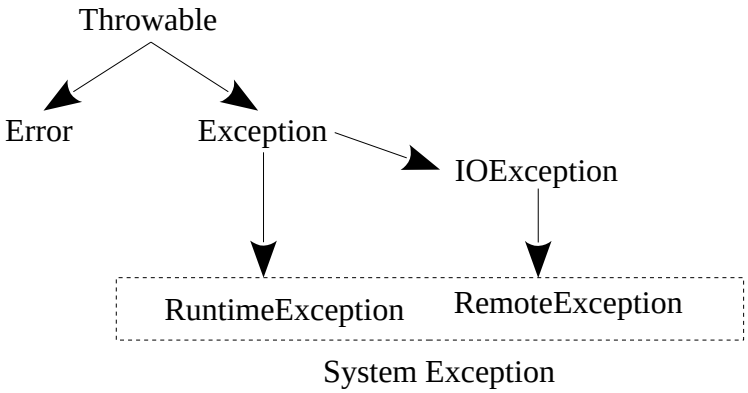
```
}
```

В примере задан транзакционно-ограниченный CDI MBean TxScopedBean и другой CDI MBean с транзакционными методами foo() и bar(). Если вызвать последовательно foo() и bar(), то bar()напечатает 0 вместо 1, т.к. каждый метод создаст свою транзакцию, в которой будет внедрен свой экземпляр tXScopedBean.

JTA транзакции и исключения

Application Exception vs System Exception

Исключения в java порождаются классом java.lang.Throwable. Они делятся на подклассы java.lang.Error (серьезные системные сбои, используемые на этапе компиляции) и java.lang.Exception (исключения приложений, используемые в run-time). Исключения Exception делятся на непроверяемые (unchecked) — те, что JVM пробрасывает из метода автоматом и проверяемые (checked) – те, которые надо проверять в коде при помощи try/catch или пробрасывать из метода самостоятельно (оператором throws). К непроверяемым исключениям относится лишь подкласс java.lang.RuntimeException и его наследники, все остальные исключения, включая корневой Exception, относятся к проверяемым.



EJB-контейнер делит исключения на системные (system) и прикладные (application). Системное исключение — сбой внутри EJB-контейнера, выброс такого исключения прерывает текущий бизнес-процесс. Предполагается, что системные исключения не будут обрабатываться клиентом, поэтому они оборачиваются EJB-контейнером в одну из исключений-оберток и в таком виде возвращаются клиенту. Прикладные исключения трактуются как сбои в бизнес-логике и не являются основанием для прерывания или отката бизнес-процесса. Предполагается, что прикладные исключения будут обрабатываться клиентом, которому они и передаются в исходном виде.

System Exception

К системным исключениям с т.з. EJB-контейнера по-умолчанию относятся непроверяемые RuntimeException (в т.ч. подкласс javax.ejb.EJBException) и класс java.rmi.RemoteException с наследниками.

Для RemoteException и RuntimeException различаются механизмы преобразования системных исключений в прикладные при помощи @javax.ejb.ApplicationException.

При возникновении системного исключения EJB-контейнер обязан выполнить 3 шага:

1. Откатить активную транзакцию
2. Записать лог в журнал
3. Ликвидировать сбойный экземпляр EJB

Все системные исключения, выбрасываемые бизнес-методами и не помеченные @ApplicationException перехватываются EJB-контейнером. Выброс системного исключения в callback-методе (вроде @PostConstruct, @PostActivate и т.п.) приравнивается EJB-контейнером к выбросу из бизнес-метода.

Механизм записи логов исключений в журнал и их формат оставлены в спецификации EJB 3.1 на усмотрение вендора.

При возникновении системного исключения экземпляр EJB уничтожается, аннулируются все зависимости от него (ссылки на объект EJB) и собирается мусор, поскольку EJB-контейнер считает такие бины нестабильными и небезопасными. Влияние на приложение ликвидации экземпляра EJB зависит от типа этого бина. Для SLSB потеря экземпляра из пула несущественна. Для SFSB выброс экземпляра будет означать потерю диалогового состояния, т.е. разрыв сессии с клиентом и аннуляцию клиентской ссылки на бин. При повторном обращении клиента по этой же ссылке, клиент получит от EJB-контейнера исключение NoSuchEJBException, относящееся к подклассу RuntimeException.

Rem: как будет вести себя удаленная ссылка (proxy) при удалении экземпляра ejb, зависит от вендора.

В случае MDB, исключение, выброшенное onMessage() или callback-методами @PostConstruct и @PreDestroy, приведет к ликвидации экземпляра MDB. Для MDB/BMT переотправка сообщения JMS-провайдером зависит от того, когда было послано сообщение о неподтверждении приема. В случае MDB/CMT EJB-контейнер не подтвердит прием (?)транзакцию(?) и JMS-провайдер отправит сообщение вновь.

EJB-контейнер перехватывает системные исключения сессионных бинов и повторно выбрасывает RuntimeException для клиента, невзирая на его тип — удаленный или локальный.

Если клиент инициировал транзакцию, которая затем распространилась на метод EJB, системные исключения этого метода будут перехвачены и преобразованы EJB-контейнером в javax.ejb.EJBTransactionRolledbackException – подкласс RuntimeException. Т.о. клиент получит более точную информацию об откате транзакции. Если же клиент не включил метод в свою транзакцию, то системное

исключение будет преобразовано после перехвата в javax.ejb.EJBException – также подкласс RuntimeException. Поскольку внетранзакционные вызовы, как правило, идут к read-only методам (?), то EJBException обычно будет выбрасываться после выброса исключений небизнес-подсистемами (JDBC/SQLExceptiion, JMS/JMSException). Разработчик бина может программно перехватывать и обрабатывать подобные системные исключения вместо вызова EJBException, но это требует хорошего понимания их влияния на транзакцию. Обычно практикой же будет просто выброс EJBException или @ApplicationException в режиме rollback, что позволит EJB-контейнеру автоматом откатить транзакцию и ликвидировать сбойный экземпляр бина.

ApplicationException

Прикладные исключения выбрасываются в ответ на сбои в бизнес-логике. Они приходят к клиенту без перепаковки EJB-контейнером, в сериализованном виде (RMI), в отличие от системных. И по-умолчанию не вызывают отката транзакции, позволяя клиенту восстанавливать транзакцию после сбоя. Прикладные исключения часто используются для оповещения клиента об ошибках валидации данных. В этом случае исключение выбрасывается до выполнения основных задач и ясно дает указывает, что это не ошибка подсистем вроде JDBC, JMS, RMI, JNDI и т.д.

Аннотация уровня класса @javax.ejb.ApplicationException служит для объявления класса исключения прикладным. Ее параметр rollback=true заставляет контейнер откатывать транзакцию при выбросе, по-умолчанию rollback=false.

```
Ex.:
@ApplicationException(rollback=true)
public class ExApplicationException extends Exception {...}
```

Выброс такого исключения ExApplicationException бизнес-методом EJB будет автоматически откатывать транзакцию. Однако бизнес-логика приложения (обычно в перехватчике) может перехватывать подобные исключения и повторно запускать транзакцию, предотвращая, например, остановку бизнес-процесса (разрыв сессии).

Аннотацией @ApplicationException можно помечать и классы системных исключений, т.е. наследников RuntimeException и RemoteException, превращая их в прикладные исключения. Этом может быть полезно при нежелании автоматической перепаковки отдельных исключений в EJBException или отката транзакции в ответ на их выброс.

Взаимодействие исключений сессионных EJB и Entity

неясная таблица

JTA транзакции. Рекомендации по использованию транзакций в JEE

CMT

В CMT стандартным UOW является метод объекта. Вызов метода определяет логические границы транзакции, в рамках которой, помимо операций UOW, также надо обрабатывать прикладные исключения и устанавливать при необходимости флаг отката. Обработку прикладных исключений и отката транзакции рекомендуется производить в коде самого вызываемого метода, а не делегировать их по объектам и методам, вызываемым по ходу транзакции, поскольку так легче отслеживать цепочку исключений и место отката при отладке, читать и поддерживать код. Перед выполнением вычислительно-сложных операций рекомендуется проверять флаг отката методом getRollbackOnly() с тем, чтобы не делать ненужной работы.

При выборе атрибутов транзакции лучше перестраховаться и потребовать транзакции даже там, где она не нужна, нежели выполнить UOW без нее. К примеру, надо быть осторожным при установке атрибута SUPPORTS на уровне класса (вместо REQUIRED по-умолчанию), поскольку все методы класса после этого по-умолчанию могут быть выполнены вне транзакций.

Желательно отмечать прикладные исключения @ApplicationException, это позволит более гибко управлять откатом транзакции и подстрахует от пропуска вызова setRollbackOnly().

BMT

К достоинствам BMT можно отнести возможность точного управления транзакционными границами и ее жизненным циклом. Вместе с этим BMT сложнее распространяется и требует ручной обработки исключений (rollback). Поэтому крупные методы, содержащие внутри себя BMT, лучше разбить на малые CMT-методы.

Ругательства

- UOW = Unit Of Work = единица работы
- ACID = Atomicity Consistency Isolation Durable = атомарность согласованность изолированность надежность
- Open Group = конторка впаривающая стандарты, opengroup.org
- DTP = Distributed Transaction Processing = модель распределенной транзакции Open Group
- XA = eXtendedArchitecture = расширенная архитектура = стандарт Open Group, реализующий взаимодействие диспетчеров ресурсов и распределенных транзакций в DTP. Диспетчер ресурсов может иметь любую природу, но должен поддерживать интерфейс XA. Сам XA использует протокол 2PC.
- JTA = Java Transaction API = интерфейс, реализующий взаимодействие всех участников транзакций с TM. Он состоит из 3 частей: высокоуровневый клиентский интерфейс для разметки транзакций (javax.transacton.UserTransaction); низкоуровневый XA-интерфейс, реализующий стандарт XA (и, следовательно,

2PC) для взаимодействия TM и менеджеров ресурсов (javax.transaction.xa); высокоуровневый серверный интерфейс для управления TM и прикладными клиентскими транзакциями (представлен интерфейсами TransactionManager, Transaction, Status and Synchronization пакета javax.transaction). В JEE7 используется спецификация JTA 1.2, куда, помимо прежних управления транзакциями и автоматической поддержки XA, добавлены возможности по использованию транзакций вне EJB, в т.ч. аннотация @TransactionScoped для CDI MBean. Существуют встроенные и отдельные реализации JTA: встроенный TM сервера GlassFish, JBoss Transaction Manager, Atomikos и Bitronix JTA.

- OMG = Object Management Group = группа управления объектами = еще одна конторка, впаривающая стандарты платформо-независимого взаимодействия в ООП, разработчик стандарта CORBA.
- OTS = Object Transaction Service = OMG-спецификация службы, определяющей транзакционное поведение в разнородной распределенной среде, организованной по стандарту OMG CORBA. JTS = Java Transaction Service = Java-реализация OMG OTS. Поддерживает высокоуровневые интерфейсы JTA. Реализует спецификацию OTS, в т.ч. распространение транзакции между OTS-совместимыми TM по протоколу IIOP. Предназначена для использования в промышленных интеграционных (enterprise middleware) приложениях.
- EJB/JTA/JTS = использующийся в EJB-контейнере TM в свою очередь использует реализации JTA и JTS при работе с транзакциями.
- CMT = Container Management Transaction = транзакция, управляемая контейнером
- CMT-объект = Container Management Transaction – объект = объект, вовлеченный в управляемую EJB-контейнером транзакцию
- BMT = Bean Management Transaction = транзакция, управляемая объектом бина
- BMT-объект = объект, управляющий собственной транзакцией
- RM = Resource Manager = менеджер ресурсов = программный компонент, отвечающий за управление ресурсом. Он поддерживает собственные (локальные) транзакции ресурса и регистрирует их в менеджере транзакций.
- TM = Transaction Manager = Менеджер Транзакций = программный компонент, отвечающий за управление транзакциями, он создает их от имени приложения, связывает их менеджерами ресурсов (заключает с ними «сделки» или операции задействования) и дает им команды на фиксацию или откат локальной транзакции. Именно TM отвечает за поддержаение ACID - свойства транзакции.
- 2PC = TPC = Two Phase Commit = двухфазная фиксация = механизм фиксации распределенных транзакций в 2 этапа: 1 - опрос менеджеров ресурсов (локальные транзакции), 2- принятие глобального решения commit/rollback и финальная рассылка менеджерам ресурсов указания применить его.
- HD = Heuristic Decision = эвристическое решение
- Callback-метод = метод, который будет вызываться из вне приложения
- TSFSB = Transactional SFSB = транзакционный бин с поддержкой состояния
- T-M-R State = Transaction Method-Ready State
- JPA = Java Persistence API. Это спецификация JEE, описывающая интерфейс фреймворка, реализующего технологию Java Persistence. Persistence – технология управления объектно-реляционным отображением (ORM = Object Relation

Mapping) реляционной БД (РБД) на программные объекты. По сути Java Persistence - это эмулятор Java-объектной БД, имитирующий ее некоторым персистентным (синхронизированным с РБД) слоем связанных и управляемых java-объектов на основе взаимодействия с реальной РБД.

- EM = Entity Manager. Система управления Entity-объектами, в JPA задана интерфейсом javax.persistence.EntityManager. EM осуществляет объектно-реляционное отображение (ORM) между Entity и связанной с заданным источником данных (Data Source) РБД, создает запросы к персистентному слою (JP QL) или напрямую в РБД (Native SQL), отвечает за синхронизацию состояния персистентного слоя с РБД и осуществляет поиск Entity-объектов.
- Persistence Context (PC) — персистентный слой (окружение), совокупность управляемых EM Entity-объектов, эмулирующая объектную БД. Персистентый слой автоматически создается и поддерживается JPA-движком в процессе выполнения в коде операций EM, т.е. в режиме run-time.
- TS PC = Transaction Scoped PC. PC, который живет в пределах JTA-транзакции. После завершения транзакции TS PC ликвидируется, а все его Entity-объекты отвязываются от EM (становятся detach), иными словами TS PC является персистентным слоем без поддержки состояния (stateless) — выстрелил-забыл. TS PC можно организовать только в рамках JEE приложения.
- ES PC = Extended Scoped PC. PC, который живет столько же, сколько и stateful EJB, в котором и должен создаваться, причем все транзакции в рамках этого stateful EJB будут взаимодействовать именно с этим PC.
- TS EM = Transaction Scope EM. EM, который порождает TS PC.
- ES EM = Extended Scope EM. EM, который порождает ES PC.
- AM EM = Application Managed EM. EM, который управляется приложением — такой способ характерен для JSE приложений. При использовании AM EM на код приложения ложится поддержка потокобезопасности, закрытия EM и подключение к транзакциям JTA или RESOURCE_LOCAL (командой em.joinTransaction).
- CM EM = Container Managed EM. EM, который управляется JEE-сервером, что соответствует JEE-приложениям. Потокобезопасность и закрытие CM EM обеспечиваются JEE-движком. При работе с CMEM JEE-движок автоматически применяет глобальный (контейнерный) транзакционный механизм JTA.
- Функциональное программирование = парадигма, в которой программа представляет собой результат взаимодействия математических функций (операторов). Важнейшим отличием от императивного программирования является отсутствие понятия состояния объекта. Т.е. функциональная программа зависит лишь от входа и всегда даст тот же результат при повторном вызове с тем же аргументом. Это позволяет оптимизатору кэшировать результаты типовых запросов, менять порядок вызова и распараллеливать вычисления без ущерба результату программы. Одним из примеров можно считать шаблонные языки Xpath/XSLT.
- Императивное программирование = парадигма, в которой программа представляет собой последовательность тактов смены состояния объекта. Ключевым понятием тут является переменная, позволяющая хранить часть состояния и менять свое

значение в последующих тактах программы.

Литература

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

Изучаем Java EE 7, Энтони Гонсалвес, стр. nnn

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

http://www.kumaranuj.com/2013/06/jpa-2-entitymanagers-transactions-and.html