事务

1 Java事务处理

事务管理是对一系列操作的管理，它最终只有两个结果，要么成功，要么失败。一旦失败，所有的操作将回滚到初始状态；一旦成功，才最终提交。

Java 事务的类型有三种：

1. JDBC 事务
2. JTA（Java Transaction API）事务
3. 容器事务

1.1 JDBC事务

JDBC 事务是用`Connection`对象控制的。JDBC Connection 接口（`java.sql.Connection`）提供了两种事务模式：自动提交和手工提交。

`java.sql.Connection`提供了以下控制事务的方法：

1. `public void setAutoCommit(boolean)`
2. `public boolean getAutoCommit()`
3. `public void commit()`
4. `public void rollback()`

使用 JDBC 事务界定时，可以将多个 SQL 语句结合到一个事务中。

JDBC 事务的一个缺点是事务的范围局限于一个数据库连接。一个 JDBC 事务不能跨越多个数据库。

1.2 JTA事务

JTA 是一种高层的、与实现无关的、与协议无关的 API，应用程序和应用服务器可以使用 JTA 来访问事务。

JTA 允许应用程序执行分布式事务处理，在两个或多个网络计算机资源上访问并且更新数据，这些数据可以分布在多个数据库上。

如果计划用 JTA 界定事务，那么就需要有一个实现 `javax.sql.XADataSource`、`javax.sql.XAConnection` 和 `javax.sql.XAResource` 接口的 JDBC 驱动程序。一个实现了这些接口的驱动程序将可以参与 JTA 事务。JDBC 驱动程序的 JTA 支持极大地增强了数据访问能力。

一个 `XADataSource` 对象需要一个 `XAConnection` 对象的工厂。`XAConnection` 是参与 JTA 事务的 JDBC 连接。

用户需要用应用服务器的管理工具设置 `XADataSource`。Java EE 应用程序用依赖注入或者 JNDI 查询数据源。一旦应用程序找到了数据源对象，它就调用 `javax.sql.DataSource.getConnection()` 以获得数据库的连接。

XA 连接与非 XA 连接的不同：

1. XA 连接参与了 JTA 事务，非 XA 连接参与了 JDBC 事务。
2. XA 连接不支持 JDBC 的自动提交功能。
3. 应用程序一定不要对 XA 连接调用 `java.sql.Connection.commit()` 或者 `java.sql.Connection.rollback()` 方法。
4. 应用程序应该使用 `UserTransaction.begin()`、`UserTransaction.commit()` 和 `UserTransaction.rollback()`。

1.3 容器事务

相对于编码实现 JTA 事务管理，我们可以通过 EJB 容器提供的容器事务管理机制（CMT）完成同一个功能，这项功能由 Java EE 应用服务器提供，我们称之为容器事务。

容器事务主要是 Java EE 应用服务器提供的。容器事务大多是基于 JTA 完成，这是一个基于 JNDI 的，相当复杂的 API 实现。

容器事务使得我们可以简单地指定将哪个方法加入事务，一旦指定，容器将负责事务管理任务。这是我们推荐的解决方式，因为通过这种方式我们可以将事务代码排除在逻辑编码之外，同时将所有困难交给 Java EE 容器去解决。

使用 EJB CMT 的另外一个好处就是程序员无需关心 JTA API 的编码，不过，理论上我们必须使用 EJB。

1.4 三种事务的比较

1. JDBC 事务控制的局限性在一个数据库连接内，但是其使用简单。
2. JTA 事务的功能强大，事务可以跨越多个数据库或多个 DAO，使用也比较复杂。
3. 容器事务，主要指的是 Java EE 应用服务器提供的事务管理，局限于 EJB 应用使用。

一般说来，在单个 JDBC 连接的情况下可以选择 JDBC 事务；在跨多个连接或者数据库情况下，需要选择使用 JTA 事务；如果用到了 EJB，则可以考虑使用 EJB 容器事务。

2 EJB的事务管理

EJB 有两种使用事务的方式：

1. 通过容器管理的事务，叫 CMT（容器事务）。 简化编程，不用显式地标记事务的边界，代码中并不包括表示事务的开始和事务结束的语句。EJB 默认采用 CMT。使用`@TransactionManagement(TransactionManagementType.CONTAINER)`标注表示使用 CMT。
2. 通过 Bean 管理的事务，叫 BMT（JDBC 事务或者 JTA 事务）。代码中需要使用表示事务开始和事务结束的语句来确定事务的边界。使用`@TransactionManagement(TransactionManagementType.BEAN)`标注表示使用 BMT。

2.1 CMT

使用 CMT 时，默认情况下，EJB 的每个方法代表一个事务，方法被调用时开始事务，调用结束时提交事务，方法执行中抛出异常时回滚事务。

注意：在一个方法中不允许事务嵌套或出现多个事务。

EJB 容器管理的事务不要求所有的方法与事务关联。可以通过设置事务属性来指定 EJB 的哪一个方法与事务相关联。通过将`@TransactionAttribute(TransactionAttributeType)`标注在方法之前来进行设置。

事务属性的类型可以取以下类型之一：

- `Required`（必须需要一个事务，默认值）
- `RequiredNew`（必须需要一个新事务）
- `Mandatory`（强制需要一个事务）
- `NotSupported`（不支持事务）
- `Supports`（支持事务）
- `Never`（永远不支持事务）

事务属性控制事务的范围。例如，Bean1 中的 methodA 方法开始某一事务，并在 methodA 的方法体中调用 Bean2 的 methodB，当 methodB 执行时，它是运行在由 methodA 启动的事务范围内呢，还是运行在一个新启动的事务范围内？这取决于 methodB 的事务属性，如下表所示。

事务属性	客户端事务（T1）	业务方法的事务（T2）
`Required`	无	T2
`Required`	T1	T1
`RequiredNew`	无	T2
`RequiredNew`	T1	T2
`Mandatory`	无	错误
`Mandatory`	T1	T1
`NotSupported`	无	无
`NotSupported`	T1	无
`Supports`	无	无
`Supports`	T1	T1
`Never`	无	无
`Never`	T1	错误

示例代码：

```
@TransactionManagement(TransactionManagementType.CONTAINER)
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
@Stateful
public class TransactionBean implements Transaction
{
    @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
    public void firstMethod() {...}

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void secondMethod() {...}

    public void thirdMethod() {...}

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

2.2 BMT

由于 CMT 依靠容器启动、提交和回滚事务，所以会限制事务的边界位置。而 BMT 则允许通过编程的方式来指定事务的开始、提交和回滚的位置。

JTA 事务中主要使用的是 `javax.transaction.UserTransaction` 接口，JDBC 事务中使用 `java.sql.Connection` 接口。

2.2.1 JTA事务编程方法

1. 使用 `@TransactionManagement(TransactionManagementType.BEAN)` 标注，表示这是一个 Bean 管理的事务。
2. 获得实现了 `UserTransaction` 接口的事务对象。
3. 使用 `UserTransaction` 接口的 `begin()` 和 `commit()` 方法划分事务边界。

获得实现了 `UserTransaction` 接口的事务对象的方法有以下几种：

1. 依赖注入，使用 `@Resource` 标注。例如：

```
@Resource
private UserTransaction userTransaction;
```

2. JNDI 查找
3. EJBContext。例如：

```
@Resource
private SessionContext context;

UserTransaction userTransaction = context.getUserTransaction();
```

示例代码：

```
@TransactionManagement(TransactionManagementType.BEAN)
@Stateful
public class TransactionBean implements Transaction
{
    @Resource
    UserTransaction utx;

    public void firstMethod()
    {
        utx.begin();
        // Perform transacted tasks
        utx.commit();
    }
}
```

2.2.2 JDBC事务的编程方法

示例代码：

```
@TransactionManagement(TransactionManagementType.BEAN)
@Stateful
public class TransactionBean implements Transaction
{
    public void firstMethod()
    {
        // ...
        Class.forName("com.mysql.jdbc.Driver");
        Connection conn = DriverManager.getConnection(url, user, pwd);
        conn.begin();
        // Perform transacted tasks
        conn.commit();
    }
}
```

3 JPA事务管理

JPA 中的事务是由`EntityManager`对象来管理的，`EntityManager`对象所使用的事务管理方式有两种，分别为 JTA 和 RESOURCE_LOCAL（JDBC 事务），即 Java Transaction API 方法和本地的事

务管理。

RESOURCE_LOCAL 事务是数据库本地的事务。它是数据库级别的事务，只能针对一种数据库，不支持分布式的事务。对于中小型的应用，可以采用 RESOURCE_LOCAL 管理 EntityManager 事务。

JTA 事务（Java Transaction API）可以支持分布式的事务——在两个或多个网络计算机资源上访问并且更新数据，这些数据可以分布在多个数据库上。如果系统采用的是分布式的数据库，那么只能选择 JTA 管理 EntityManager 事务。JTA 事务一般只能运行在 Java EE 环境中（EJB 服务器或者 Web 服务器）。

JPA 中的事务类型通过 persistence.xml 文件中的`transaction-type`元素配置。例如，配置事务为 JTA 方式的代码如下所示：

```
<persistence>
    <persistence-unit name ="demo" transaction-type="JTA">
        <non-jta-data-source>
            java:jboss/datasources/MySqlDS
        </non-jta-data-source>
    </persistence-unit>
</persistence>
```

如果使用 RESOURCE_LOCAL 管理事务，则配置代码如下所示：

```
<persistence>
    <persistence-unit name="demo" transaction-type="RESOURCE_LOCAL">
        <property name="eclipselink.jdbc.driver" value="com.mysql.jdbc.Driver" />
        <property name="eclipselink.jdbc.url" value="jdbc:mysql://localhost:3306/jpa" />
        <property name="eclipselink.jdbc.password" value="123456" />
        <property name="eclipselink.jdbc.user" value="root" />
    </persistence-unit>
</persistence>
```

不同的运行环境、不同的`EntityManager`对象所支持的事务类型也是不同的，如下表所示：

	Java EE 环境		Java SE 环境
	EJB 容器	Web 容器	
应用托管的 EntityManager	JTA（容器管理），RESOURCE_LOCAL	JTA（非容器管理），RESOURCE_LOCAL	RESOURCE_LOCAL
容器托管的 EntityManager	JTA（容器管理）	不支持	不支持

由此可见，容器托管的`EntityManager`只能运行在 EJB 容器中，只能采用 JTA 的方式管理事务。而在 Java SE 环境中，只能使用应用托管的`EntityManager`并且只能采用 RESOURCE_LOCAL 的方式管理事务。

3.1 EJB容器中应用托管的`EntityManager`的事务处理

例 1：JPA 使用 JTA 事务，且 EJB 使用容器管理事务。

```
@TransactionManagement(TransactionManagementType.CONTAINER) // 容器管理
@Stateless
public class StudentService implements IStudentService
{
    @PersistenceUnit(unitname="jpaUnit")
    private EntityManagerFactory emf;

    public Student findStudentById(Integer studentId)
    {
        EntityManager em = emf.createEntityManager();
        Student student = em.find(Student.class, studentId);
        em.close();
        return student;
    }

    public void placeRecord(Integer studentId, Record record)
    {
        EntityManager em = emf.createEntityManager();
        Student student = em.find(Student.class, studentId);
        student.getRecords().add(record);
        em.merge(student);
        em.close();
    }
}
```

当调用`placeRecord()`方法时，容器自动关联一个 JTA 事务，并开始事务，将该事务记为 A。

当`EntityManager`被创建的时候，会建立一个新的持久化上下文，并将`EntityManager`的事务附加到 JTA 的事务 A 中。

当`placeRecord()`方法调用结束的时候，事务 A 被提交。如果调用期间出现异常，则事务 A 被回滚。

例 2：与例 1 的配置相同，但是生成`EntityManager`对象的时间不同。

```java
@TransactionManagement(TransactionManagementType.CONTAINER) // 容器管理
@Stateless
public class StudentService implements IStudentService
{
    @PersistenceUnit(unitName = "jpaUnit")
    private EntityManagerFactory emf;

    private EntityManager em;

    @PostConstruct
    public void init()
    {
        em = emf.createEntityManager();
    }

    public Student findStudentByld(Integer studentId)
    {
        // 查询不需要关联事务
        Student student = em.find(Student.class, studentId);
        em.clear();
        return student;
    }

    public void placeRecord(Integer studentId, Record record)
    {
        /* 由于 EntityManager 对象是在当前 JTA 事务之外创建的,
         * 对于在事务的活动范围之外创建的 EntityManager 对象,
         * 需要调用 joinTransaction() 方法,
         * 才能将 EntityManager 对象的事务附加到当前的活动事务中
         */
        em.joinTransaction();

        Student student = em.find(Student.class, studentId);
        student.getRecords().add(record);
        em.merge(student);

        // 手动脱离当前持久化上下文
        em.flush();
        em.clear();
    }

    @PreDestroy
    public void destroy()
    {
        em.close();
    }
}
```

3.2 Web容器中应用托管的 `EntityManager` 的事务处理

```java
@WebServlet("/StudentService")
public class StudentService implements extends HttpServlet
{
    @PersistenceUnit(unitname = "jpaUnit")
    private EntityManagerFactory emf;

    @Resource
    private UserTransaction utx;

    public Student findStudentById(Integer studentId)
    {
        EntityManager em = emf.createEntityManager();
        Student student = em.find(Student.class, studentId);
        em.close();
        return student;
    }

    public void placeRecord(Integer studentId, Record record)
    {
        utx.begin();

        EntityManager em = emf.createEntityManager();
        Student student = em.find(Student.class, studentId);
        student.getRecords().add(record);
        em.merge(student);

        utx.commit();
        em.close();
    }
}
```

在 Web 容器中，无法使用容器管理的 JTA 事务。必须使用实现了`UserTransaction`接口的对象来完成事务边界的划分，从而完成事务的启动和提交。

`EntityManager`对象必须在事务边界内创建，否则需要调用`joinTransaction()`方法，让其附加到当前事务中。

3.3 Java SE环境下应用托管的`EntityManager`的事务处理

```java
public class StudentService
{
    private EntityManagerFactory emf;
    private EntityManager em;
    private Student student;

    public StudentService()
    {
        // Java SE 环境下通过 Persistence 类的静态方法获得实体管理器工厂
        emf = Persistence.createEntityManagerFactory("jpaUnit");
        em = emf.createEntityManager();
    }

    public Student findStudentById(Integer studentId)
    {
        student = em.find(Student.class, studentId);
        return student;
    }

    public void placeRecord(Integer studentId, Record record)
    {
        em.getTransaction().begin();
        student.getRecords().add(record);
        em.getTransaction().commit();
    }

    public void destroy()
    {
        em.close();
        emf.close();
    }
}
```

1. 通过调用`EntityManager`的`getTransaction()`方法获得本地事务对象。
2. 划分事务边界时要使用`begin()`和`commit()`方法。
3. 需要手动创建和关闭`EntityManagerFactory`、`EntityManager`对象。

3.4 无状态会话Bean中容器托管的`EntityManager`的事务处理

```java
@Stateless
public class StudentManager implements StudentManagerLocal
{
    @PersistenceContext(unitName = "jpaUnit")
    EntityManager manager;

    public void addStudent()
    {
        Student st = new Student();
        st.setName("Waee");
        st.setGender("male");
        st.setMajor("art");
        st.setAddress_id(1);
        manager.persist(st);
    }

    // ...
}
```

当调用实体管理器时，首先检查实体类是否处在持久化上下文中。若在，实体管理器使用该持久化上下文和其关联的事务；若没有关联任何持久化上下文，则创建一个新的持久化上下文，并将实体管理器与所在的事务关联。

当调用结束时，提交事务，在持久化上下文中托管的实体变为游离状态，并且持久化上下文也随着销毁。

3.5 有状态会话Bean中容器托管的`EntityManager`的事务处理

```java
@Stateful
public class StudentManager implements StudentManagerLocal
{
    @PersistenceContext(unitName = "jpaUnit",
                        type = PersistenceContextType.EXTENDED)
    EntityManager manager;

    SchoolEO school;

    pubilc void init(int school_num)
    {
        school = em.find(SchoolEO.class, school_num);
    }

    public void addStudent()
    {
        Student st = new Student();
        st.setName("Waee");
        school.getStudents.add(st);
        manager.merge(school);
    }

    // ...
}
```

`@PersistenceContext` 标注中配置 `type = PersistenceContextType.EXTENDED`，表明该 `EntityManager` 为超出事务范围的持久化上下文（扩展的持久化上下文）。扩展的持久化上下文只能绑定在有状态会话 Bean 上，持久化状态不会根据事务的关闭而销毁。

在事务提交之后，持久化上下文不会随着事务的提交而销毁，持久化上下文中被托管的实体对象不会变成游离态。

扩展的持久化上下文在整个会话期间持续。

扩展的持久化上下文必须需要一个有状态会话 Bean，不能应用在无状态会话 Bean 或者 Java SE 环境中。