EJB3.1（事务）

一个普通的接口与实现，一旦添加了几个Annotation后，就变成了EJB，被EJB容器管理，

成为EJB后，EJB容器可以为EJB提供依赖注入功能以及其他服务，其中一个重要的服务就是事务控制。下面是关于事务的一些概念。

1. Resource Manager

Resource Manager是用来控制单个资源的事务。比如JDBC Connection，可以通过Connection的commit和rollback控制数据库的事务。这种事务又可以叫做本地事务。

1. Transaction Manager

Transaction Manager 用来控制多个Resource的事务。或者说控制分布式事务。比如，多个JDBC Connection，或者JMS Connetion。分布式事务要求参与其中的各个资源不再自己控制事务，而是交由一个全局的事务控制，并且各个资源类的链接要支持XA协议。这样的事务又可以叫做全局事务。

1. Two Phases Commit

这个全局事务用来协调各个资源的工作方式。简单的说。Transaction Manger在控制事务的时候分两步，第一步让各个资源确认是否可以commit，第二步根据第一步的结果，向所有资源发出全部commit或者全部rollback的指令。

1. JTA

JTA是JavaEE关于全局Transaction的接口，EJB使用JTA作为处理全局事务的工具。

其中我们常见的是两个接口

1. javax.transaction.UserTransaction 接口
2. javax.transaction.TransactionManager接口

其实现由JavaEE服务器软件提供，

1. javax.transaction.UserTransaction

这是面向开发人员的接口，提供begin(), commit(), rollback()等方法，让开发人员可以手动控制事务。

1. javax.transaction.TransactionManager

这是面向EJB容器的接口，提供比UserTransaction更多的一些方法，通常情况由EJB容器去调用这些方法，但也并不强制不允许开发人员调用。

1. Container-Managered Transaction

Container管理事务又叫CMT，意思是由EJB容器管理事务，

EJB容器调用TransactionManager的方法来控制事务。

1. Bean-Managered Transaction

Bean管理事务又叫BMT，意思是Bean控制事务，或者说由开发者显示控制

开发人员调用UserTransaction的方法来控制事务。

一个简单的带有DB操作的JavaEE 应用

HelloWorldPojo 接口

**package** example.pojo;

**import** java.sql.Connection;

**public** **interface** HelloWorldPojo {

**public** **void** greeting(Connection conn);

}

HelloWorldPojoBean 实现

**package** example.pojo;

**import** java.sql.Connection;

**import** java.sql.PreparedStatement;

**import** java.sql.SQLException;

**public** **class** HelloWorldPojoBean **implements** HelloWorldPojo {

@Override

**public** **void** greeting(Connection conn) {

saveMessage(1, "Hello World EJB3.1 1!", conn);

saveMessage(2, "Hello World EJB3.1 2!", conn);

}

**private** **void** saveMessage(**int** count, String message, Connection conn) {

PreparedStatement ps = **null**;

**try** {

String sql = "INSERT INTO ATEST (ID, CONTENT) VALUES (?, ?)";

ps = conn.prepareStatement(sql);

ps.setInt(1, count);

ps.setString(2, message);

ps.executeUpdate();

ps.close();

} **catch** (SQLException e) {

**if** (ps != **null**) {

**try** {

ps.close();

} **catch** (SQLException e1) {

}

}

**throw** **new** RuntimeException(e);

}

}

}

注:由于篇幅原因，后面的例子中，saveMessage方法将省略。

greeting方法调用两次saveMessage，向数据库中插入数据。

一个调用这个类的Servlet

**package** example.servlet;

**import** java.io.IOException;

**import** java.sql.Connection;

**import** java.sql.SQLException;

**import** javax.annotation.Resource;

**import** javax.servlet.ServletException;

**import** javax.servlet.annotation.WebServlet;

**import** javax.servlet.http.HttpServlet;

**import** javax.servlet.http.HttpServletRequest;

**import** javax.servlet.http.HttpServletResponse;

**import** javax.sql.DataSource;

**import** example.pojo.HelloWorldPojo;

**import** example.pojo.HelloWorldPojoBean;

@WebServlet("/StudyTransactionServlet")

**public** **class** StudyTransactionServlet **extends** HttpServlet {

**private** **static** **final** **long** *serialVersionUID* = 1L;

@Resource

**private** DataSource dataSource;

**private** HelloWorldPojo helloWorld = **new** HelloWorldPojoBean();

**protected** **void** doGet(HttpServletRequest request,

HttpServletResponse response) **throws** ServletException, IOException {

Connection conn = **null**;

**try** {

conn = dataSource.getConnection();

} **catch** (SQLException e) {

**throw** **new** RuntimeException(e);

}

helloWorld.greeting(conn);

}

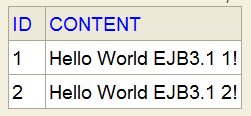
}

@Resource

**private** DataSource dataSource;

可以暂时忽略，只需要知道这样我们可以取得一个数据库的DataSource就可以了。

运行后的数据库状态。插入了两条数据。



注意，由于我们是在不同的数据库连接中观察数据库状态。所以这里看到的一般情况下都是程序提交事务后的状态。但我们还没有控制Transaction，为什么就程序commit了事务。

这是因为JDBC Connection有个属性叫AutoCommit，默认是true，所以每做一次数据库操作就提交一次。现在将Connection的AutoCommit属性设置为false，或者直接修改数据源配置

conn.setAutoCommit(**false**);

运行后数据库中的数据为空。所以并没有提交。

我们再将AutoCommit设置为true，然后模拟一下第二个处理失败的情况。

在两次调用saveMessage里针对第二次调用抛出一个RuntimeException

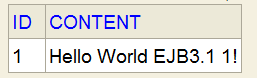
**if** (count == 2) {

**throw** **new** RuntimeException();

}

这个意思是，在第二次插入数据的时候抛错误。

运行后的数据库状态。插入了一条数据。



第一次数据还是提交了，即使第二次出错。原因如前面所讲，AutoCommit为True，

所以第一次数据库操作执行完就Commit了。注意这里第二次并不是Rollback了，而是没执行。一般来说是希望完整的处理过程中一旦出错，所有的处理都要rollback。综上所诉。在没有使用事务的时候，单靠Connection的AutoCommit是满足不了要求的。所以任何一个系统，事务控制都是不可缺少的。

现在来通过Resource Manager来控制单个的JDBC操作。使用的是

java.sql.Connection的commit和rollback方法。注意现在将AutoCommit设置为false。

**protected** **void** doGet(HttpServletRequest request,

HttpServletResponse response) **throws** ServletException, IOException {

Connection conn = **null**;

**try** {

conn = dataSource.getConnection();

} **catch** (SQLException e) {

**throw** **new** RuntimeException(e);

}

**try** {

helloWorld.greeting(conn);

conn.commit();

} **catch** (Exception e) {

**try** {

conn.rollback();

} **catch** (SQLException e1) {

**throw** **new** RuntimeException(e1);

}

}

}

这样无论是前面提到的正常情况，还是第二次处理异常的情况。都可以正确处理事务。

现在再加一个类，这个类做更新处理，考虑两种情况，一是在Servlet中调用，二是在第一个类中调用。

先看在Servlet中调用的情况。

HelloWorldPojo2 接口

**package** example.pojo;

**import** java.sql.Connection;

**public** **interface** HelloWorldPojo2 {

**public** **void** greeting(Connection conn);

}

HelloWorldPojoBean2

**package** example.pojo;

**import** java.sql.Connection;

**import** java.sql.PreparedStatement;

**import** java.sql.SQLException;

**public** **class** HelloWorldPojoBean2 **implements** HelloWorldPojo2 {

@Override

**public** **void** greeting(Connection conn) {

saveMessage(1, "Hello World EJB3.1 3!", conn);

saveMessage(2, "Hello World EJB3.1 4!", conn);

}

**private** **void** saveMessage(**int** count, String message, Connection conn) {

PreparedStatement ps = **null**;

**try** {

String sql = "UPDATE ATEST SET CONTENT = ? WHERE ID = ?";

ps = conn.prepareStatement(sql);

ps.setString(1, message);

ps.setInt(2, count);

ps.executeUpdate();

ps.close();

} **catch** (SQLException e) {

**if** (ps != **null**) {

**try** {

ps.close();

} **catch** (SQLException e1) {

}

}

**throw** **new** RuntimeException(e);

}

}

}

这个方法做更新处理

Servlet

**package** example.servlet;

**import** java.io.IOException;

**import** java.sql.Connection;

**import** java.sql.SQLException;

**import** javax.annotation.Resource;

**import** javax.servlet.ServletException;

**import** javax.servlet.annotation.WebServlet;

**import** javax.servlet.http.HttpServlet;

**import** javax.servlet.http.HttpServletRequest;

**import** javax.servlet.http.HttpServletResponse;

**import** javax.sql.DataSource;

**import** example.pojo.HelloWorldPojo;

**import** example.pojo.HelloWorldPojo2;

**import** example.pojo.HelloWorldPojoBean;

**import** example.pojo.HelloWorldPojoBean2;

@WebServlet("/StudyTransactionServlet")

**public** **class** StudyTransactionServlet **extends** HttpServlet {

**private** **static** **final** **long** *serialVersionUID* = 1L;

@Resource

**private** DataSource dataSource;

**private** HelloWorldPojo helloWorld = **new** HelloWorldPojoBean();

**private** HelloWorldPojo2 helloWorld2 = **new** HelloWorldPojoBean2();

**protected** **void** doGet(HttpServletRequest request,

HttpServletResponse response) **throws** ServletException, IOException {

Connection conn = **null**;

**try** {

conn = dataSource.getConnection();

conn.setAutoCommit(**false**);

} **catch** (SQLException e) {

**throw** **new** RuntimeException(e);

}

**try** {

helloWorld.greeting(conn);

helloWorld2.greeting(conn);

conn.commit();

} **catch** (Exception e) {

**try** {

conn.rollback();

} **catch** (SQLException e1) {

**throw** **new** RuntimeException(e1);

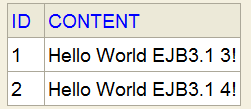
}

}

}

}

正常登录情况下，数据先登录再更新了。



如果在Bean2的第二次更新处理前抛出RuntimeException，则最后数据库中没有数据，因为rollback了

同样在Bean1的greeting方法里中调用Bean2的greeting方法。效果也是一样。这里就不赘述了。

前面的例子看上去都理所应当，但其实当中隐藏了一个概念。

就是Bean1和Bean2是使用的同一个Connection实例,最后的提交和回滚都是针对这个Connection实例的。

设想如果Bean1，Bean2使用的不是同一个Connection实例，那么情况会怎样。

最基本的就是Bean2根本更新不到数据。因为Bean1的处理在Bean1使用的Connection实例未提交之前。对于其他Connection来说是不可见的。

而且基于一次request处理绝大部分情况在都应该在同一个事务中这样一个原则。所以应该要保证一次request中的多个数据库处理应该使用同一个Connection。那么最容易想到的方式就是把Connection作为参数传递到各个类的方法里去。

现在，所有的应用的数据库操作都在使用ORM，比如Hibernate等等。我们在写代码的时候通常意识不到Connection，如果ORM采用的是Resource Manager来控制事务的话。或者说如果ORM采用的是Connection的commit和rollback来控制事务的话，那么一定要保证，所有的Dao调用数据库操作的时候，使用的Connection实例是同一个。事实上，ORM也确实是这么做的。他们会将Connection实例存放在ThreadLocal中，以保证在同一个线程中使用相同的Connection实例。

为了结合后面的概念，我觉得可以把这个概念称为为Connection的传播。意思是在一个线程内多个处理间的传播。

到此，我们了解了Resource Manager控制事务，比如java.sql.Connection的commit和rollback

以及所谓的Connection的传播。

现在我们来看看EJB的CMT是如何处理事务的。

将HelloWorldBean变为EJB

在HelloWorld上加上@Local 在HelloWorldBean上加上@Stateless，

现在HelloWorld就变成EJB，这个EJB由EJB容器管理了。

我们在Servlet中不需要再New了，而采取注入方式。

@EJB

**private** HelloWorld helloWorld;

@Resource

**private** DataSource dataSource;

**public** StudyTransactionServlet() {

**super**();

}

**protected** **void** doGet(HttpServletRequest request,

HttpServletResponse response) **throws** ServletException, IOException {

Connection conn = null;

**try** {

conn = dataSource.getConnection();

conn.setAutoCommit(**false**);

} **catch** (SQLException e) {

**throw** **new** RuntimeException(e);

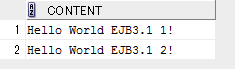
}

helloWorld.greeting(conn);

}

我们可以把刚才的两个Case跑一下，

1. 全成功



1. 第二个失败。



我们什么都还没做，事务控制就成功了。这就是EJB提供的功能CMT。我们虽然不需要实现CMT，但我们要知道他的式样。这样才能很好的应用。

1. CMT作用的单位

CMT和BMT都是针对一个EJB组件而言。我们通常可以这么说，一个EJB组件是CMT的还是BMT的。比如例子中的HelloWorldBean，这个就是EJB组件。而EJB组件是通过配置文件，或者在BeanClass上的Annotation来声明自己是CMT，还是BMT的。

@TransactionManagement(TransactionManagementType.*BEAN*)

@TransactionManagement(TransactionManagementType.*CONTAINER*)

所有的EJB组件默认是CMT的。

1. CMT事务的边界

CMT事务由EJB的方法调用开始，到调用结束为止。

1. CMT的commit和rollback

在下面的情况中CMT会rollback

EJB方法中抛出RuntimeException 或RemoteException的子类

EJB方法中抛出任意子定义Exception，如果Exception被如下标识

@ApplicationException(rollback=true)

public class BusinessException extends Exception {

}

EJB方法中调用了sessionContext.setRollbackOnly() 方法。

注意，前两种Exception发生的时候EJB容器会自动调用sessionContext.setRollbackOnly()方法。

上面以外的情况下，在方法结束时会Commit

1. CMT的约定

在CMT的事务控制范围内。不应该调用各个Resource Manager的自己的事务控制，

因为CMT采用的是JTA，全局事务控制。

以java.sql.Connection为例

Commit，rollback，setAutoCommit 方法不允许被调用

另外， javax.ejb.EJBContext的 getUserTransaction方法，

以及UserTransaction的所有方法都不允许被调用。

**注：上例中我们确实调用了setAutoCommit方法，因为调用是在CMT事务范围外。另外，即使AutoCommit设置为true，CMT也会忽略此设置。**

我们再来看看BMT是如何处理事务的。

@Stateless

@TransactionManagement(TransactionManagementType.*BEAN*)

**public** **class** HelloWorldBean **implements** HelloWorld {

@Resource

**private** UserTransaction userTransaction;

@Override

**public** **void** greeting(Connection conn) {

**try** {

userTransaction.begin();

} **catch** (NotSupportedException | SystemException e1) {

}

**try** {

saveMessage(1, conn, "Hello World EJB3.1 1!");

saveMessage(2, conn, "Hello World EJB3.1 2!");

userTransaction.commit();

} **catch** (Exception e) {

**try** {

userTransaction.rollback();

} **catch** (IllegalStateException | SecurityException

| SystemException e1) {

}

}

}

篇幅原因saveMessage方法省略，

@TransactionManagement(TransactionManagementType.*BEAN*)

声明这个EJB是BMT的。

@Resource

**private** UserTransaction userTransaction;

是JavaEE帮我们注入UserTransaction的写法。暂时不理解也不要紧。

我们通过userTransaction的begin,commit,rollback来控制事务。

1. BMT的边界

BMT以begin方法开始，commit,rollback方法结束事务

1. BMT的commit和rollback

BMT的commit,rollbak完全有程序控制。

1. BMT的约定

在BMT的事务控制范围内。不应该调用各个Resource Manager的自己的事务控制，

因为BMT采用的是JTA，全局事务控制。

以java.sql.Connection为例

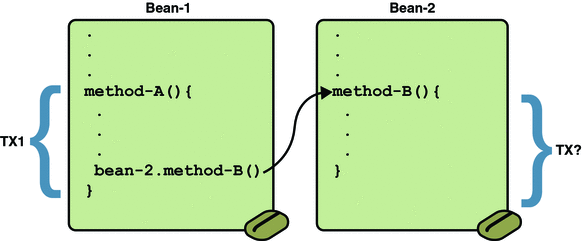
Commit，rollback，setAutoCommit 方法不允许被调用

了解了单独的Bean的事务控制之后，多个Bean的时候如何。考虑两种Case

1. Bean1 中调用Bean2的方法。
2. Servlet中先调用Bean1，再调用Bean2。

先看第一种 Bean1 调用Bean2. 这就要说到传播属性。

**注意事务传播属性只在CMT的Bean中有效**



比如Servlet里调用了Bean1的methodA，methodA又调用了Bean2的methodB

事务传播属性包含下面六种，他们定义了CMT的Bean的事务是如何接受已有事务的。

* Required
* RequiresNew
* Mandatory
* NotSupported
* Supports
* Never

|  |  |  |  |
| --- | --- | --- | --- |
| Transaction Attribute | Client’s Transaction | Business Method’s Transaction | 说明 |
| **Required** | **None** | **T2** | 如果没有事务，则创建一个事务 |
|  | **T1** | **T1** | 如果已有事务，则参与这个事务 |
| **RequiresNew** | **None** | **T2** | 如果没有事务，则创建一个事务 |
|  | **T1** | **T2** | 如果已有事务，依然创建一个事务 |
| **Mandatory** | **None** | **error** | 如果没有事务，则出错 |
|  | **T1** | **T1** | 如果已有事务，则参与这个事务 |
| **NotSupported** | **None** | **None** | 如果没有事务，不参与事务 |
|  | **T1** | **None** | 如果已有事务，依然不参与事务 |
| **Supports** | **None** | **None** | 如果没有事务，不参与事务 |
|  | **T1** | **T1** | 如果已有事务，则参与这个事务 |
| **Never** | **None** | **None** | 如果没有事务，不参与事务 |
|  | **T1** | **error** | 如果已有事务，则出错。 |

这些事务属性通过在Bean的Class级别或者是Method级别的Annotation来添加，当然也可以通过配置文件。比如

@TransactionAttribute(TransactionAttributeType.*REQUIRED*)（**默认**）

@TransactionAttribute(TransactionAttributeType.*REQUIRES\_NEW*)

先从字面上解释一下REQUIRED的流程。当Servlet调用Bean1的MethodA的时候，这时候并没有事务，EBJ容器将根据Bean1的MethodA上定义的REQUIRED的事务传播属性，开启一个事务，

MethodA调用Bean2的MethodB的时候，这时候已经有了一个事务，EBJ容器将根据Bean2的MethodB上定义的REQUIRED的事务传播属性，将MethodB加入到当前事务中

CMT Bean 调用 CMT Bean

**package** example.ejb;

**import** java.sql.Connection;

**import** java.sql.PreparedStatement;

**import** java.sql.SQLException;

**import** javax.ejb.EJB;

**import** javax.ejb.Stateless;

@Stateless

**public** **class** HelloWorldBean **implements** HelloWorld {

@EJB

**private** HelloWorld2 helloWorld2;

@Override

**public** **void** greeting(Connection conn) {

saveMessage(1, conn, "Hello World EJB3.1 1!");

saveMessage(2, conn, "Hello World EJB3.1 2!");

helloWorld2.greeting(conn);

}

**private** **void** saveMessage(**int** count, Connection conn, String message) {

**try** {

StringBuffer sqlb = **new** StringBuffer();

sqlb.append("INSERT INTO ATEST ").append("VALUES (?) ");

PreparedStatement ps = conn.prepareStatement(sqlb.toString());

ps.setString(1, message);

System.*out*.println(ps.executeUpdate());

ps.close();

} **catch** (SQLException e) {

**throw** **new** RuntimeException(e);

}

}

}

注意Class上默认为@TransactionManagement(TransactionManagementType.*CONTAINER*)

注意方法上默认为@TransactionAttribute(TransactionAttributeType.*REQUIRED*)

**package** example.ejb;

**import** java.sql.Connection;

**import** java.sql.PreparedStatement;

**import** java.sql.SQLException;

**import** javax.ejb.Stateless;

@Stateless

**public** **class** HelloWorldBean2 **implements** HelloWorld2 {

@Override

**public** **void** greeting(Connection conn) {

saveMessage(3, conn, "Hello World EJB3.1 3!");

saveMessage(4, conn, "Hello World EJB3.1 4!");

}

**private** **void** saveMessage(**int** count, Connection conn, String message) {

**try** {

StringBuffer sqlb = **new** StringBuffer();

sqlb.append("INSERT INTO ATEST ").append("VALUES (?) ");

PreparedStatement ps = conn.prepareStatement(sqlb.toString());

ps.setString(1, message);

System.*out*.println(ps.executeUpdate());

ps.close();

} **catch** (SQLException e) {

**throw** **new** RuntimeException(e);

}

}

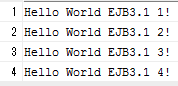
}

注意Class上默认为@TransactionManagement(TransactionManagementType.*CONTAINER*)

注意方法上默认为@TransactionAttribute(TransactionAttributeType.*REQUIRED*)

Servlet调用HelloWorld.greeting方法，由于这时候没有事务，所以HelloWorld.greeting处创建了一个事务，HelloWorld.greeting里调用了HelloWorld2.greeting，由于已经有了事务，所以这里就参与这个事务。也就是说两个方法在同一事务内。最后HelloWorld.greeting正常终了后commit.

正常结果



异常结果，在Bean2的第二个setMessage方法内抛出RuntimeException。

这里考虑两种情况，

1. 这个Exception在Bean1中不被处理。
2. 这个Exception在Bean1中被Catch，然后Bean1正常终了。

这两种情况都会导致最后的rollback

因为在任何一个CMT 的EJB方法中只要抛出了runtimeException等会导致rollback的Exception。

EJB会自动调用setRollbackOnly(true)方法。

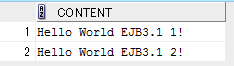
*REQUIRES\_NEW*事务属性

将HelloWorld2.greeting的事务属性改为

@TransactionAttribute(TransactionAttributeType.*REQUIRES\_NEW*)

Servlet调用HelloWorld.greeting方法，由于这时候没有事务，所以HelloWorld.greeting处创建了一个事务，HelloWorld.greeting里调用了HelloWorld2.greeting，虽然已经有了事务，但这里依然创建一个新的事务，并将前一个事务挂起。也就是说两个方法不在在同一事务内。还是上面的例子。

运行结果为



意思是，虽然HelloWorld2.greeting所在的事务Rollback了，但HelloWorld.greeting的事务依然Commit了

看一下BMT Bean调用BMT Bean

BMT的Bean，是由开发者自定义事务边界。

比如我们将HelloWorld2定义为 BMT的Bean，并在方法内控制事务

现在看一下BMT的Bean1调用BMT的Bean2的情况

Bean1的方法里使用UserTransaction,并调用Bean2的方法。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Bean1 | Bean2 | 条件 | 结果 | 备考 |
| BMT | BMT | Bean1  SetAutoCommit（true/false） | java.sql.SQLException: can't call setAutoCommit when the connection is JtaManaged |  |
| BMT | BMT | Bean2  SetAutoCommit（true） |  | 留意顺序 |
| BMT | BMT | Bean2  SetAutoCommit（false） |  |  |

这里有一些概念

1. Servlet或者BMT的Bean中可以是使用UserTransaction

（A Servlet or a BMT EJB can use a UserTransaction）

1. 无论是在BMT还是CMT的事务中，Bean不能调用Connection的commit,rollback和

setAutoCommit方法

1. BMT的Bean不能参与到一个已存在的事务中，如果在调用BMT的Bean的方法的时候已经存在了事务，则该事务将被挂起，直到BMT的Bean方法结束。

（BMT beans cannot participate in an existing transaction. If you call a BMT bean while a transaction is already in progress, the transaction will be suspended by the container prior to invoking the BMT bean and resumed after the method completes）

根据第一条，我们在Bean1这个BMT的Bean中使用UserTransaction没有问题。

根据第二条，我们在Bean1里操作connection的setAutoCommit方法将会出错。

根据第三条，我们在Bean1里调用Bean2的时候，已经有事务存在，这时候在调用Bean2

方法的时候，该事务将被挂起，也就是说Bean2将不存在在任何事务中，这时候，connection的AutoCommit属性起作用。当为True的时候，会先插入3和4，然后提交，Bean2方法执行完了以后，原有的事务被回复，然后Commit

1和2， 当为False的时候，3,4不会被提交，只有1,2被提交。

综上所述。我们的项目中，即使使用BMT的Bean，那也不可能将所有的Bean都设置为BMT。

再看一组例子。

Bean1是BMT，使用UserTransaction， Bean2是CMT， 事务传播属性为*REQUIRED*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Bean1 | Bean2 | 条件 | 结果 | 备考 |
| BMT | CMT | Bean2  SetAutoCommit（true/false） | java.sql.SQLException: can't call setAutoCommit when the connection is JtaManaged |  |
| BMT | CMT | 正常终了 |  | 留意顺序 |
| BMT | CMT | Bean2的第二个操作抛出任意的Exception  Bean1在发现Bean2出现Exception后调用rollback |  |  |
| BMT | CMT | Bean2的第二个操作抛出RuntimeException  Bean1 catch Bean2出现Exception后调用commit | javax.transaction.RollbackException: Unable to commit: transaction marked for rollback |  |
| BMT | CMT | Bean2的第二个操作抛出自定义RuntimeException并设置  rollback为false。  Bean1 catch Bean2出现Exception后调用commit |  |  |

继续一些概念。

1. CMT的Bean里不能使用UserTransaction，但可以接收一个UserTransaction发起的事务。(CMT beans cannot use a UserTransaction, but they can participate in a transaction started by a UserTransaction)

或者说CMT的Bean可以接受一个事务，不管他是来自一个UserTransaction还是来自EJB容器自己创建的。

1. CMT的Bean里抛出RuntimeException，即使在该Bean的地方Catch住Exception，EJB容器任然会setRollbackOnly为true,这时候再commit将出错。

(RuntimeExceptions thrown from a method of a CMT bean will cause the transaction to be marked for rollback, even if you catch and handle the exception in the calling code)

1. 可以通过自定义一个RuntimeException，并设置rollback为false来回避这个问题。

@ApplicationException(rollback=**false**)

**public** **class** MyException **extends** RuntimeException {

}

(Use @ApplicationException to disable this on a case by case basis for custom runtime exception classes)

根据以上的概念。这里有一个相对的最佳实践。

1. 一般的EJB都应该是CMT的。只有在边界的EJB才会考虑是用BMT还是CMT。
2. 如果是用BMT，那么其他参与事务CMT的Bean的事务传播属性一般情况下最好设置为

*MANDATORY*。这是为了防止这些CMT的Bean在无事务的情况下被使用。因为一旦设置为*REQUIRED*，在无事务的情况下使用这些Bean，EJB容器会创建事务。这有的时候会造成隐藏的错误。当然某些具体情况还需要具体分析。

1. 因为UserTransaction的begin,commit,rollback会抛出非RuntimeException，如果你不想这么做的话，可以使用CMT的Bean作为边界，事务属性设置为*REQUIRES\_NEW，*也能明确边界
2. 如果两个EJB是并列被Servlet调用的。那么这两个EJB应该都设为CMT，并且在Servlet中使用UserTransaction定义事务边界。

上文的一些概念来自[David Blevins](http://stackoverflow.com/users/190816/david-blevins)的在Stack overflow上的一次回答

<http://stackoverflow.com/questions/7196242/why-does-cmt-commit-on-exit-of-ejb-method-when-transaction-attribute-is-require>