자바트렌젝션 프로그래밍

Transaction의 개념과 이해, 분산 Transaction의 작동원리와 WAS에서의 Distributed Transaction의 구현 원리, Java Transaction 관련 API와 예제, EJB에서의 Transaction들을 개념위주로 설명하고, 간단한 예제 코드를 통해서 그 이해를 돕도록 한다.

1. What is Transaction?

트렌젝션이란, 중단 없이 시작에서부터 종료까지 한번에 수행되어야 하는 하나의 작업단위를 이야기한다. 수행이 끝난 후에는 중간에 작업이 실패되었을 경우에는 작업 수행전의 상태로 그대로 돌아가야 한다.

이해를 돕기 위해서 쉽게 예를 들어서 설명하도록 하자, A계좌에서 B계좌로 1,000원을 계좌 이체를 한다고 가정하자. 이 작업은 다음과 같은 순서로 이루어지게 된다.

1) A계좌에서 1,000원을 인출한다.
2) B계좌에 1,000원을 더한다.

만약 위의 작업을 수행하던 도중 A계좌에서 1,000원이 인출된 후에, 은행 시스템의 오류로 인해서 B계좌에 1,000원이 더해지지 않는다면, 중간에 1,000원이라는 돈은 공중에서 증발해 버린 것이 된다. 이럴 때는 다시 계좌이체를 수행하기 이전의 상태로 되돌려서 A계좌로부터 1,000원을 인출하지 말아야 한다.

그래서 위의 1), 2) 작업은 한꺼번에 이루어져야 한다. 계좌이체 작업과 같이 한번에 이루어져야 하는 작업을 트렌젝션이라고 부른다.

이처럼 트렌젝션은 쇼핑몰의 주문 결제나, 예매와 같이 Mission Critical한 작업에 있어서 필수적인 개념이라고 할 수 있다.

2. Transaction Attribute “ACID”

트렌젝션은 크게 4가지 특성을 가지는데 Atomicity, Consistency, Isolation, Durability로, 이 네 가지를 줄여서 ACID라고 부른다.

그럼 이제부터 이 ACID속성 각각에 대해서 좀 더 상세하게 알아보도록 하자.

1) Atomicity (원자성)

Database modifications must follow all or nothing.
원자성이란, 하나의 트렌젝션이 하나의 단위로만 처리가 되어야 한다는 것이다. 하나의 트렌젝션 안에 여러 가지 step 의 트렌젝션이, 하나로 처리가 되어야 한다. 위의 계좌 이체처럼, 계좌에서 돈을 빼고, 그 돈을 다른 계좌에 넣는 것과 같이 두 개 이상의 step으로 구성되어 있더라도, 계좌 이체라는 하나의 트렌젝션으로 처리가 된다.
그래서, 어느 step에서라도 트렌젝션이 실패가 되었을 경우에는 모든 상태가 트렌젝션 수행 전으로 rollback 되어서, 이전 상태를 유지해야 한다.
즉 트렌젝션의 원자성은 트렌젝션이 완전히 수행되거나, 아무것도 수행되지 않는 All or Nothing의 이미지를 가지게 된다.

2) Consistency (일관성)

States that only valid data will be written to the database
트렌젝션이 종료된 후에 저장되는 데이타는 오류 없는 데이타만 저장되어야 한다.
즉, 계좌이체 과정에서, 인출은 되었는데, 다른 계좌로 돈이 안넘어 갔다던지, 트렌젝션이 끝난 후에, 잘못된 데이타 값으로 변경 되었던지, 데이타베이스 constraint가 깨졌던지 했을 때, Consistency가 잘못 되었다고 이야기하고, 이런 경우에 그 트렌젝션 내용을 저장하지 말고, 이전 상태로 rollback 되어야 한다.

3) Isolation (격리성)

Multiple transactions occurring at the same time not impact each other execution
격리성이란, 트렌젝션 중에 변경된 내용이 트렌젝션이 완료되기 전까지는 다른 트렌젝션에 영향을 주어서는 안 된다는 이야기이다.
Tx A라는 트렌젝션 수행 중 EMP 테이블의 값을 변경 했을때, Tx A가 완료되지 않은 상태에서 Tx B가 EMP 테이블의 값을 참고할 경우에, Tx A에 의해 변경된 값을 참고하는 것이 아니라(아직 Tx A가 완료되지 않았기 때문에) 기존의 값을 참고하게 해야 한다.

4) Durability (지속성)

Ensure that any transaction committed to the database will not be lost.

지속성이란, 트렌젝션이 완료된 후에의 값이 영구저장소에 제대로 기록이 되어야 한다는 의미이다. 트렌젝션이 완료되었는데, 디스크 IO에러, 네트워크 등으로 그 값이 제대로 기록이 되지 않거나 해서는 안 된다.

3. Distributed Transaction (Global Transaction)

1) Transaction in source code

그러면, 일반적으로 프로그래밍 코드 상에서, 트렌젝션을 사용하는 방법에 대해서 알아보도록 하자.

```
Begin Transaction
// Do transaction

if(error) then roll back

  Prepare Transaction.
if (prepare transaction failed) then rollback
else Commit Transaction

End Transaction
```

일반적인 트렌젝션 프로그램은 위와 같은 코드 구조를 가지고 있다.

먼저 Begin Transaction을 시작하면, 여기부터가 하나의 Transaction임을 알리는 boundary의 역할을 한다.

// Do transaction 부분에서 Transaction에 대한 실질적인 트렌젝션 처리 코드가 들어간다.

(SQL문장 등등..)

If(error) then roll back :트렌젝션 처리 중에 에러가 발생하면, rollback을 실행해서, 트렌젝션 수행 전 상태로 되돌린다. 트렌젝션에 대한 처리가 끝나면, Prepare Transaction 을 실행해서, commit이 가능한지 각각의 트렌젝션 대상 (Resource Manager, DBMS와 같은..)에 체크를 한다.

If(prepare transaction field) then rollback
Else Commit Transaction

Commit이 될 준비가 됐으면 commit을 실행하고, 아닌 경우에는 rollback을 한다. 모든 트렌젝션이 완료되었으면 End Transaction으로 이 트렌젝션을 종료한다.

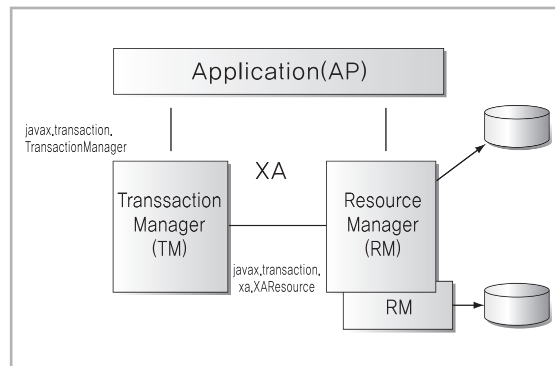
언뜻 보기에 당연하고 단순한 과정으로 보일 수 있으나, 이는 Transaction을 처리하는 대부분의 미들웨어에서 공통적으로 사용하는 방법이니, 꼭 머리 속에 익혀 두도록 하자.

J2EE EJB에서의 트렌젝션은 대부분 이런 트렌젝션의 처리 과정을 EJB Container에서 자동으로 처리해주기 때문에(CMT : Container Managed Transaction Model) 실질적으로 개발자가, 직접적으로 트렌젝션을 Handling할 일은 그렇게 많지는 않다.

2) Distributed Transaction and XA

그러면 이제 미들웨어의 중요한 기능이라고 할 수 있는 분산 트렌젝션(Distributed transaction, or Global transaction이라고도 한다.)에 대해서 알아 보도록 하자.

분산 트렌젝션이란, 두개 이상의 Resource (RDBMS등등)을 이용해서 하나의 트렌젝션으로 처리하는 것을 이야기 한다. 분산 트렌젝션을 수행하기 위한 시스템의 구조를 잠깐 살펴 보도록 하자.



〈 그림 1. 분산 트렌젝션을 수행하기 위한 미들웨어의 대략적 구조 〉

분산 트렌젝션을 구성하는 구조에는 크게 3가지 요소가 결합된다. AP, TM, RM이다.

Application

각각을 알아 보면, AP는 Application, 즉 User가 트렌젝션을 발생시키고 프로그래밍 하는 부분을 말한다. 우리가 개발하게 될 부분은 이 부분이다.

Resource Manager

다음으로는 RM은 Resource Manager로, 각각의 Resource를 컨트롤 해주는 기능을 갖는다. JAVA에서는 일종의 JDBC 드라이버와 같은 기능을 한다고 생각하면 되고, 실제로 RM의 기능을 하는 모듈은 DBMS Vendor에서 JDBC드라이버 패키지에 넣어서 배포하도록 되어 있다. 단순히 RDBMS 뿐 아니라, RM을 가지고 있는 Resource로는 XA를 지원하는 ISAM, JMS와 같은 메시징 시스템에서부터 상당히 많은 종류가 있다.

Transaction Manager

마지막으로 TM은 Transaction Manager의 약자로, 전체 분산 트렌젝션을 관리해주는 역할을 한다. WAS에 이 기능이 내장되어 있다.

XA는 eXtended Architecture로 Open Group에 의해서 정의되었으며, Global Transaction을 관리하기 위한 귀약과, RM과 TM사이의 프로토콜을 정의하고 있다.

그러면 어떻게 트렌젝션이 수행이 되는지, 대략적인 시나리오를 한번 살펴보도록 하자.
아래와 같은 트렌젝션 수행 코드가 있다고 하자.

```
Begin Transaction
 Xid = get new transation id
 doTransaction(Xid,RM1)
 doTransaction(Xid,RM2)

 commitTransaction(Xid)
End Transaction
```

Begin Transaction
Xid = get new transaction id

먼저 트렌젝션이 수행되면, 그 트렌젝션에 대한 Global Transaction Id인 Xid가 생성이 된다.
그리고, DB1과 DB2에 각각 연결되어 있는 RM1과 RM2이 연결되어 있을 때,

doTransaction(Xid,RM1)
doTransaction(Xid,RM2)

AP가 직접 RM을 통해서 DB작업을 실행한다.

이때 Xid를 같이 RM에게 보내서 지금 실행되는 작업이 어떤 트렌젝션에 관련된 작업인지를 식별할 수 있게 해준다. 물론, 이 작업 중에, error가 발생했을 경우에는 roll back을 수행한다. 여기까지 작업이 종료되면,

commitTransaction(Xid)

을 하게 되는데. 이 작업은 TM과 RM간의 작업이다. TM이 RM1,RM2에게 commit할 준비가 되었는지를 각각의 RM에 물어서 체크를 한다. 이때 RM은 동시에 여러 개의 Transaction이 수행중일 수 있다. (RDBMS가 한번에 하나의 트렌젝션만 처리 하고 있지는 않다. 당연히 여러 개의 트렌젝션을 처리하고 있다.) 그래서, 어떤 트렌젝션을 체크할 것인지를 구별하기 위해서, TM은 RM에게 Xid를 실어서 보낸다. 이렇게 TM이 RM에게 Xid로 식별된 트렌젝션이 commit할 준비가 되었는지를 묻는 과정을 prepare라고 한다.

Prepare가 되었으면 TM은 RM에게 commit을 하도록 지시한다. 만약 prepare가 실패하면 TM은 RM에게 rollback을 지시한다.

이렇게 prepare과정과 commit과정 두 단계를 거치는 것을 two phase commit (2pc)라고 한다. 분산 트렌젝션에서 많이 나오는 개념이니까는 잘 알아두도록 하자.

EndTransaction

트렌젝션을 종료하고, 그 내용을 반영한다.

지금까지 살펴본 내용은 분산 트렌젝션의 아주 일반적인 작동 원리를 살펴보았다. (실제로 XA프로토콜을 이용해서, 분산 트렌젝션을 처리하는 과정은 이보다 좀더 복잡하다. 좀 더 자세한 사항은 XA 관련 Spec이나, Transaction 관련 서적을 참고하기 바란다.)

4. Transaction Isolation Level

데이타베이스 환경에서는 일반적으로 Single User가 아니라 Multi User 환경을 지원하기 때문에, 여러명은 동시에 같은 데이타를 읽거나 insert/update하는 일이 생긴다. 이 과정에서, 데이타의 일관성(Consistency)와 동시성(Concurrency)를 보장해 주어야 한다.

■ **Data Consistency** – 데이타 일관성은, 어느 사용자가 데이타를 ACCESS 하던지, 같은 내용의 데이타를 볼 수 있는 속성
■ **Data Concurrency** – 데이타 동시성은, 동시에 여러 사용자가 같은 데이타를 ACCESS할 수 있는 속성

일반적으로 데이터 베이스는 여러 개의 트렌젝션을 하나씩 순차적(serially)으로 처리하는 것이 아니라 동시에 여러 트렌젝션을 처리한다. 이 과정에서 동시에 처리되는 트렌젝션간의 변경된 데이타를 각각 어떻게 적용할 것인가에 대한 규칙이 Transaction Isolation Level이다.

예를 들어 설명해 보자.

1) 트렌젝션 TX_A가 시작 되었고, 동시에 트렌젝션 TX_B가 시작되었다.
2) TX_A에서 SELECT NAME FROM TABLE_A WHERE ID=1 을 수행하였다. 결과는 ORIGIN이 나왔다고 가정하자.
3) TX_B에서 UPDATE TABLE_A SET NAME=" CHANGED" WHERE ID =1
4) TX_A에서 다시 SELECT NAME FROM TABLE_A WHERE ID=1 을 수행하였다.
5) TX_B를 COMMIT하였다.
6) TX_A에서 다시 SELECT NAME FROM TABLE_A WHERE ID=1 을 수행하였다.

이런 과정을 거쳤을 때, 4)에는 CHANGED가 나올까? 아니면 ORIGIN이 나올까? 6)에서도 CHANGED가 나올까 ORIGIN이 나올까? 즉 어떤 트렌젝션이 데이타를 변경하였을 때 같은 데이타를 억세스하는 트렌젝션에 대해서 어떤 영향을 미치느냐를 정하는 규칙(Isolation Level)에 따라서 나올 수 있는 결과가 다르게 된다.

1) Preventable Phenomena

서로 다른 트렌젝션에 어떻게 영향을 주느냐에 따라서 3가지 Preventable Phenomena로 나뉘어 지는데.. 그 내용을 살펴보자.

Dirty Read

어떤 트렌젝션이 데이타를 변경하고, COMMIT을 하지 않았을 때에도, 다른 트렌젝션에서 그 값을 읽으면 변경된 값이 반영되는 경우이다.

1) TX_A가 시작되고, SELECT NAME FROM TABLE_A WHERE ID=1에서 ORIGIN이 출력되었다고 가정하자.
2) TX_B에서 UPDATE TABLE_A SET NAME=" CHANGED" WHERE ID =1를 실행한 후
3) TX_A에서 SELECT NAME FROM TABLE_A WHERE ID=1은 CHANGED가 출력된다.

즉 2)에서 UPDATE한 내용을 COMMIT하지 않았는데도 반영이 되면 이를 Dirty Read라고 한다.

NonRepeatable Read (Fuzzy Read)

어떤 트렌젝션이 데이타를 변경 (update와 delete만 해당됨)하고, COMMIT을 하면, COMMIT된 내용은 다른 트렌젝션에 반영된다.insert된 내용은 반영되지 않는다.

1) TX_A가 시작되고, SELECT NAME FROM TABLE_A WHERE ID=1에서 ORIGIN이 출력되었다고 가정하자.
2) TX_B에서 UPDATE TABLE_A SET NAME=" CHANGED" WHERE ID =1를 실행한 후
3) TX_A에서 SELECT NAME FROM TABLE_A WHERE ID=1하면 ORIGIN이 출력된다.
4) TX_B를 COMMIT한 후
5) SELECT NAME FROM TABLE_A WHERE ID=1하면 CHAGNED가 출력된다.

Phantom Read

어떤 트렌젝션이 새로운 데이타를 삽입(Insert)하고, COMMIT하면, COMMIT된 내용은 다른 트렌젝션에 반영된다. (NonRepeatable read와 같으나, NonRepeatable을 update, delete가 Phantom Read는 insert된 내용이 반영된다.)

1) TX_A가 시작되고, SELECT COUNT(*) FROM TABLE_A WHERE GRP=1에서 10이 출력되었다고 가정하자.
2) TX_B에서 INSERT INTO TABLE_A (ID,GRP) VALUES(100,1);
3) TX_A에서 SELECT COUNT(*) FROM TABLE_A WHERE GRP=1하면 10 이 출력된다.
4) TX_B를 COMMIT한후
5) TX_A에서 SELECT COUNT(*) FROM TABLE_A WHERE GRP=1하면 11 이 출력된다

2) Isolation Level

ANSI/ISO SQL 표준 (SQL92)에서는 4가지 Isolation Level을 지원하는데. 그 내용은 간단하게 다음과 같은 표로 정리된다.

Isolation Level	Dirty Read
Read uncommitted	Possible
Read committed	Not Possible
Repeatable read	Not Possible
Serializable	Not Possible
NonRepeatable Read	**Phantom Read**
Possible	Possible
Possible	Possible
Not Possible	Possible
Not Possible	Not Possible

3) How to setup Oracle and Java Isolation Level

많이 사용하고 있는 Oracle DBMS의 경우에는 Default가 Read Commited로 설정이 되어 있으며, Oracle 9(현재)까지, Isolation Level은 Read commited와, Serializable 두가지만 지원하고 있다.

Oracle에서 Isolation Level을 변경하는 방법은 SQL PLUS에서 LEVEL에 따라서 다음과 같은 명령어를 치면 된다.

```
SET TRANSACTION ISOLATION LEVEL READ COMMITED;
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

JAVA에서도 데이타베이스 Isolation Level에 따라 알맞은 Isolation Level을 지정해 줘야 하는데, 그 방법은 다음과 같다.

```
Connection conn = DriverManager.getConnection(url, uid,
pwd);
conn.setTransactionIsolation(Connection.TRANSACTION_R
EAD_COMMITTED);
 System.out.println(conn.getTransactionIsolation());
```

지금까지 간단하게 Isolation Level에 대해서 살펴보았다. 트렌젝션에 관련된 내용이고 심심치 않게 언급되는 내용이라서 간단하게 정리해 보았다. 좀 더 자세한 내용은 Oracle 홈페이지에서 Oracle 8i Concepts Release 8.1.5 A67781-01 문서의 27장 Data Concurrency and Consistency 부분을 살펴보기 바란다.

5. Java transaction

그럼 이제부터 자바에서 어떻게 트렌젝션을 처리하게 되어 있는지 그 구조를 살펴보고, 각각의 API들에 대해서 간단하게 살펴보도록 하자.

1) Java Transaction Model

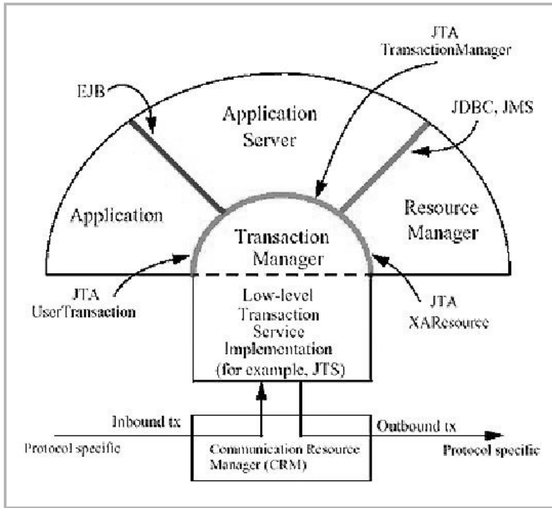
Java Transaction을 구성하는 API로는 크게 JTS(Java Transaction Service)와 JTA (Java Transaction API)로 구성이 된다.

JTS는 Transaction Manager 에 대한 Spec이고, JTA는 Application, Application Server, Resource Manager들간의 Interface 규약 등을 정의하는 Spec이다.

일반적으로, Java로 직접 Transaction Programming을 할 경우에는 JTA Interface를 통해서 JTS를 사용하게 된다.
쉽게 이야기해서 JTS는 Transaction Manager, 그리고 JTA는 Transaction Manager의 외부 인터페이스로 이해하면 된다.

이미 알고 있는 사람도 있겠지만, JTS를 지원하는 Transaction Manager간에는 Global Transaction이 가능하다. 예를 들면 WebLogic에서 EJB들을 호출하고, 그 EJB들에서 IBM Websphere의 EJB를 호출 하면, 그것이 하나의 트렌젝션으로 묶이어서 처리할 수 있다는 이야기다.

이것이 가능한 이유는, JTS로 구현된 Transaction Manager의 경우 CORBA의 OTS(Object Transaction Service)라는 프로토콜로 Transaction Manager간에 통신을 할 경우에는, 하나의 Global Transaction으로 처리가 가능하게 되는 것인데, 이는 OTS에서 TM간의 Transaction을 하나로 핸들링 할 수 있는 프로토콜을 제공하기 때문이다.



〈그림 2. JAVA Transaction API들간의 관계 〉

그러면 각각의 Java Transaction API들을 살펴보도록 하자.

2) JTA

그러면 이런 일련의 트렌젝션 과정이 Java에서는 어떻게 구현되는지를 간단하게 살펴보고 넘어가도록 하자. Java에는 트렌젝션을 처리하기 위한 API로 Java Transaction API(이하 JTA)를 제공한다. JTA에는 TM, AP, RM간 상호작용을 정의하기 위해서, 아래와 같이 주요 다섯 가지 인터페이스를 정의하고 있다.

- User Transaction Interface
- Transaction Manager Interface
- Transaction Interface
- XAResource Interface
- Xid Interface

a) User Transaction Interface

javax.transaction.UserTransaction 클래스로, AP와 TM간의 인터페이스를 나타낸다.
실제로 JTA를 이용해서 프로그래밍을 할 때는 이 User Transaction Interface를 사용한다.(즉 우리가 직접 Transaction Programming 을 할 때, 직접적으로 사용하게 되는 Interface다.)

EJB Server내에서의 구현은

```
UserTransaction utx = ctx.getUserTransaction();

// start Transaction
utx.begin();

// do transaction
 :

utx.commit();
```

이런식으로 구현된다. (이렇게 EJB내에서 구현할 때는 Transaction Model이 TX_BEAN_MANAGED 일 때만 사용한다.-Bean Managed Transaction)

b) Transaction Manager Interface

javax.transaction.TransactionManager interface로, TM와 Application Server간의 인터페이스를 정의한다. 주로 각 트렌젝션간의 경계(boudary)를 관리한다.
하나의 트렌젝션에 대해서 Transaction Context를 가지게 하고, 이 트렌젝션을 사용하는 Thread들간의 연관관계를 정의한다. 고로, 여러 Thread가 하나의 Transaction으로 묶이는 것과 같은 Global Transaction(분산 트렌젝션)이 가능해지는 것이다.

Transaction Manager Interface에서는 트렉젠셕과의 경계관리를 다음과 같은 원리로 수행하게 된다.
Transaction Context를 관리하고, 이 각각의 Transaction Context를 Transaction object에 의해 encapsulation 한다. 트렌젝션을 사용하는 모든 쓰레드들은 Transaction Context에 대한 reference를 가지게 되고, 만약 트렌젝션을 사용하지 않는 쓰레드는 그 값을 null로 세팅한다.

이렇게 Transaction Manager가 Transaction을 관리하기 위해서는 몇 가지 주요 기능을 제공하는데, 그 기능들은 아래와 같다.

■ Starting a Transaction
전체 트렌젝션을 시작한다. 이 단계에서, Transaction object의 Transaction Context를 해당 Thread와 Binding한다.

■ Completing a Transaction
TransactionManager.commit/rollback을 이용해서, 현재

commit를 호출한 Thread의 트렌젝션을 완료(commit)한다. 트렌젝션의 완료가 끝나면, Thread의 Transaction Context를 null로 세팅한다.

■ Suspending and Resuming Transaction
현재 calling Thread가 수행중인 Transaction을 suspend하거나, resume시킬 수 있다. Transaction 수행코드 중간에, suspend를 시키면, suspend중에 수행된 명령(SQL등)은 그 Transaction에 포함되지 않는다.

c) Transaction Interface

Transaction Interface는 트렌젝션을 수행하기 위한, Transaction 그 자체의 정보를 기억한다. 즉, 트렌젝션을 수행하기 위해서 필요한, Resource 목록을 기록하고, 각 트렌젝션의 commit과 rollback을 관리한다.

Transaction이 생성되면, 각 Transaction에 관여 되는 Resource(RDBMS etc..) 들의 List를 Transaction object에 바인딩 한다. (transaction.enlistResouce)
각 Transaction object가 관련된 Resource list를 유지하고 있기 때문에. TM이 현재 Transaction과 관련된 RM을 일괄적으로 Control (commit시키거나, rollback을 시키는 것들)이 가능하게 된다.

그 외에도, 각 Transaction에 대한 Equality Check와 Hash code 관리, synchronization 관리 등을 수행한다. 자세한 내용은 JTA Spec을 참고하기 바란다.

d) XAResource Interface

TM과 RM간의 인터페이스를 정의한다. X/OPEN XA Spec에 따른 Resource Manager의 Implementation을 Java로 표현해 놓은 것이다.
 각각의 Resource마다, 이 XAResource Interface가 제공되는데, 각각의 Resource의 local transaction의 begin, commit, prepare 등을 수행한다. 각각의 Resource에 dependent되는 만큼, XA Interface의 Implementation Class들은 JDBC Driver를 제공하는 vendor쪽에서 제공하게 된다.

XAResource Interface의 대략적인 흐름을 보면 다음 코드와 같다.

```
XADataSource xaDS;
XAConnection xaCon;
XAResource xaRes;
Xid xid;
Connection conn;
Statement stmt;
int ret;

xaDS = getDataSource(); // vendor에서 제공되는 코드를 사용
xaCon = xaDS.getXAConnection(“jdbc_user”,
“jdbc_password”);
xaRes = xaCon.getXAResource();

con = xaCon.getConnection();
stmt = conn.createStatement();

// create new global transaction id
xid = new
MYXid(formatId,globalTransactionId,branchQualifier);

try{
  xaRes.start(xid,XAResource.TMNOFLAGS); // start
transaction
  stmt.executeUpdate(sql);
  xaRes.end(xid,XAResource.TMSUCESS); // end
transaction

  ret = xaRes.prepare(xid); // prepare
  if(ret == XAResource.XA_OK) xaRes.commit(xid,false); //
commit
}catch(Exception e){
}finally{
stmt.close();
conn.close();
xaCon.close();
```

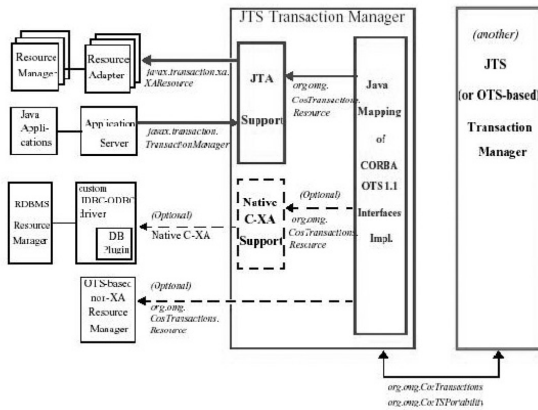
〈예제. 하나의 Resource에 대해서 하나의 트렌젝션을 수행하는 예제〉

위의 코드처럼, Resource마다, 트렌젝션을 구별하는 것은 각각의 트렌젝션의 ID, 즉 xid를 이용해서, 어떤 트렌젝션인지를 구별한다. XA Interface를 이용한 프로그래밍은 뒤에서 Oracle과 WebLogic의 XA Interface를 이용한 예제에서 자세히 살펴보도록 하자.

e) Xid Interface

Xid는 각각의 트렌젝션을 식별하기 위한 global transaction ID를 Java로 구현해놓은 Interface이다. 여기에는 X/Open의 XID structure에 따른 값이 들어가 있으며, transaction format ID, global transaction ID, branch qualifier 이 세 가지 정보가 저장된다.

3) JTS



4) JTA Support in the Application Server

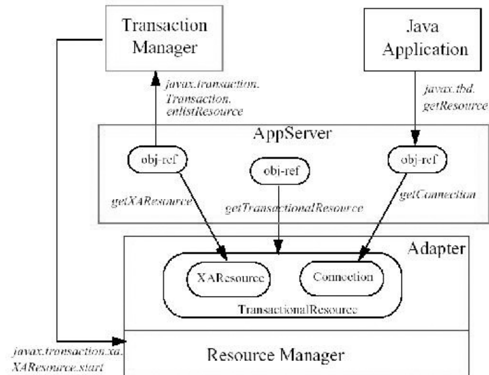
그러면 다음으로, 이 JTA/JTS 인터페이스들이 어떻게 Java Application Server에서 서로 상호 작용을 하는지 알아보도록 하자.
(※이 내용은, Application Server 내부에서 어떻게 각각의 클래스와 인터페이스가 상호 작용을 하고, 작동원리가 어떻게 되는지를 살펴보기 위한 것이다. 일반적인 트렌젝션 프로그래밍을 할 때는 꼭 알아둘 필요는 없으니 참고만 하도록 하자.)

※ TM : Transaction Manager

일반적으로 Application이, Application Server를 통해서, Transaction을 수행하는 모델의 위의 그림과 같다.
Application이 Application Server에 접속해서 Connection을 얻어서 Transaction을 수행하게 되고, Application Server 내에서는 Transaction Manager를 이용해서 Transaction 관리를 하게 되며, Transaction Manager는 Resource vendor (eg. SAP, Oracle, Message Queue etc..) Resource Adapter를 이용해서, 각각의 Resource (eg. ERP, RDBMS etc.)에 접근을 하여, Transaction 제어를 하게 된다.

Resource Adapter는 vendor에서 제공되는데, JDBC Driver Package 같은 것이 Resource Adapter의 대표적인 예가 된다. Resource Adapter내에는 여러 클래스가 있는데, 크게 4가지 종류로 나누어 볼 수 있다. ResourceFactory, Transactional Resource, Connection, XAResouce 이다.

ResourceFactory는 Transactional Resource를 생성하는 역할을 한다. RDMS에서 XADataSource가 이 분류에 속한다.
여기서 생성된 Transactional Resource는 직접 Resource로의 Connection(XAConnection)을 연결하고, 해당 Resource의 Resource Manager와 접근하기 위한 Interface를 제공하며, Application에서 사용할 Connection (DB Connection)객체와 Transaction Manager에서 사용할 XAResource Interface를 제공한다. 아래 그림을 보면서 다시 정리해 보도록 하자.

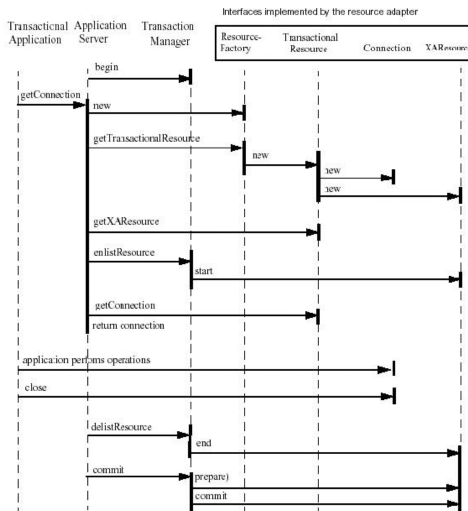


Application Server는 Transaction Manager와, Java Application과 상호작용을 하게 되어있다. Application Server는 Transaction Manager에게는 XAResource를, 그리고, Java Application에게는 Connection객체를 넘겨준다. Application Server 자체는, 각각의 Resource들과 연결될 수 있는 Transactional Resource(XAConnection)을 가지고 있다.

이를 하나의 가상 시나리오로 정리를 해 보자
1. Java Client에서 TX_REQUIRED 트렌젝션 속성을 가지고 있는 EJB를 CALL을 한다. Java Client에서는 Transaction 정의를 하지 않았다고 가정하면, Transaction은 EJB에서 부터 시작된다. Application Server는

Transaction Manager에게 Global Transaction을 시작하도록 한다. TransactionManager.begin

2. Transaction이 시작되었으면, Java Application은 데이타베이스를 접근하기 위해서 DB Connection을 요청한다.
3. Application Server는 JDBC 드라이버(resource adapter) 에 접근하여, XADataSource (Resource Factory) 객체를 얻어온다.
4. 이 XADataSource로 부터, XAConnection(Transactional Resource)을 얻어온다. (getTransactionalResource)
5. XAConnection은, XAResource와, Connection을 얻어온다.
6. Application server는 이렇게 얻어온, XAResource를 Transaction에 포함시키고 (enlistResource), XAResource Interface를 이용하여, 각각의 Resource로 하여금 Transaction을 시작하도록 한다. (start)
7. Application server는 XAConnection으로 부터 얻어온 DBConnection을 Java Application에게 넘겨준다. (getConnection/returnConnection)
8. Java Application은 받아온 Connection을 이용하여 Transaction을 수행하고, Connection을 close한다. (application performs operations, close)
9. Connection이 close되면, Application Server는 각각의 Resource를 Transaction으로부터 해제하고, (delistResource) 각각의 Resource에게 Transaction이 끝났음을 알린다. (end)
10. Application Sever에서 그 Transaction을 commit하면, Transaction Manager는 각각의 XAResource Interface를 통해서, 각각의 Resource에 prepare와 commit 메세지를 보내서 Transaction을 완료한다. (prepare,commit)



6. Java Transaction Programming

많은 경우에 Java에서 Transaction Programming을 할 경우에는 EJB를 통해서 Application Server가 자동적으로 Transaction처리를 하도록 하지만, Bean Management Transaction Model을 이용하는 EJB나, Servlet/JSP 등에서 종종 직접 트렌젝션을 처리하도록 하는 경우가 있다.

Java에서 직접 트렌젝션 프로그래밍을 할 경우에는 JDBC의 XA Interface를 이용하는 방법과, JTA를 이용하는 방법 크게 두 가지가 있다. 이 두 방식의 가장 큰 차이는 JDBC XA Interface를 이용하면, Transaction 관리를 DBMS의 Transaction Manager와, Application에서 담당해야 하고, JTA를 사용할 경우에는 Application Server의 Transaction Manager가 Transaction 관리를 하게 된다.

1) JDBC XA API Based Transaction Programming

JDBC XA API를 사용하는 경우는 위에서도 언급했듯이, DBMS의 Transaction관리 기능을 이용해서, Application에서 직접 Transaction을 관리하도록 프로그래밍을 한다.

JDBC 2.0에서부터는 Global Transaction 프로그래밍을 위한 Interface로, javax.sql.XAConnection과, javax.sql.XADataSource Interface를 제공한다. 말 그대로, Java XA Interface를 이용해서, XA Protocol에 맞춰서, 프로그래밍을 하는 것이다.

아래는 Oracle JDBC XA Driver를 이용해서, 2PC를 구현한 예제이다.

```
import java.sql.*;
import javax.sql.*;
import java.util.*;
import oracle.jdbc.driver.*;
import oracle.jdbc.pool.*;
import oracle.jdbc.xa.OracleXid;
import oracle.jdbc.xa.OracleXAException;
import oracle.jdbc.xa.client.*;
import javax.transaction.xa.*;
import java.io.*;

class OracleXARun{

    // DB Connection Info
```

```
    final static String DB_FROM_ADDR = "127.0.0.1";
    final static int DB_FROM_PORT = 1521;
    final static String DB_FROM_SID = "ORCL";
    final static String DB_FROM_USER = "scott";
    final static String DB_FROM_PASSWORD= "tiger";

    // DB Connection Info
    final static String DB_TO_ADDR = "127.0.0.1";
    final static int DB_TO_PORT = 1521;
    final static String DB_TO_SID = "ORCL";
    final static String DB_TO_USER = "scott";
    final static String DB_TO_PASSWORD = "tiger";

    // main
    public static void main(String args[]){
   OracleXARun oxa = new OracleXARun();

   try{
   oxa.XARun();
   }catch(Exception e){
       e.printStackTrace();
   }
    }

    void XARun()
   throws XAException,SQLException
    {
   // step 1. open connection
   // step 1-1. create XA Data source (ResourceFactory)
   XADataSource xds1 =
getXADataSource(DB_FROM_ADDR,DB_FROM_PORT

,DB_FROM_SID,DB_FROM_USER,DB_FROM_PASSWORD);
   XADataSource xds2 =
getXADataSource(DB_TO_ADDR,DB_TO_PORT
         ,DB_TO_SID,DB_TO_USER,DB_TO_PASSWORD);

   // step 1-2. make XA connection (Transactional
Resource)
   XAConnection xaconn1 = xds1.getXAConnection();
   XAConnection xaconn2 = xds2.getXAConnection();

   // step 1-3. make connection (DB Connecction)
   Connection conn1 = xaconn1.getConnection();
   Connection conn2 = xaconn2.getConnection();
```

```
   // step 2. get XA Resource (XAResource)
   XAResource xar1 = xaconn1.getXAResource();
   XAResource xar2 = xaconn2.getXAResource();

   // step 3. generate XID (Transaction ID)
   // 같은 Global Transaction ID를 가지고, 다른 Branch
Transaction ID
   // 를 갖는 Transaction ID 를 생성한다.
   Xid xid1 = createXid(1);
   Xid xid2 = createXid(2);

   // step 4. xa start (send XA_START message to
XAResource)
   xar1.start(xid1,XAResource.TMNOFLAGS);
   xar2.start(xid2,XAResource.TMNOFLAGS);

   // step 5. execute query(execute Transaction)
   // FROM DB의 ACCOUNT계좌에서 1000원빼고, TODB에
ACCOUNTNO가 1000이고 BALANCE가 1000인 레코드를
   // insert한다.
   String sql ;

   Statement stmt1 = conn1.createStatement();
   sql = "update accountfrom set balance=balance-1000
where accountno=1000";
   stmt1.executeUpdate(sql);

   sql = " insert into accountto values(1000,1000)";
   Statement stmt2 = conn2.createStatement();
   stmt2.executeUpdate(sql);

   // step 6. xa end (Transaction이 종료되었음을 알린다.)
   xar1.end(xid1,XAResource.TMSUCCESS);
   xar2.end(xid2,XAResource.TMSUCCESS);

   // step 7. xa prepare (xa prepare를 한다.)
   int prep1 = xar1.prepare(xid1);
   int prep2 = xar2.prepare(xid2);

   // step 8. xa commit
   // step 8-1. check prepare stat
   // 양쪽 다 prepare가 성공 하였으면 commit할 준비를 한다.
   // ※ XA_RDONLY는 update가 없이 select등의 Read Only
만 있는 Transaction
   // 이 성공하였을때,
   boolean docommit=false;
```

```
    if( (prep1 == XAResource.XA_OK || prep1 ==
XAResource.XA_RDONLY)
         && (prep2 == XAResource.XA_OK || prep2 ==
XAResource.XA_RDONLY) )
    {
        docommit = true;
    }

    if(docommit){
        // XA_RDONLY는 이미 commit이 되어 있기 때문에 따로
commit하지 않는다.
        if(prep1 == XAResource.XA_OK)
         xar1.commit(xid1,false);
        if(prep2 == XAResource.XA_OK)
         xar2.commit(xid2,false);
    }else{
        // roll back 하는 부분
        if(prep1 != XAResource.XA_RDONLY)
xar1.rollback(xid1);
        if(prep2 != XAResource.XA_RDONLY)
xar2.rollback(xid2);
    }

    // step 9. close connection
    conn1.close();
    conn2.close();
    xaconn1.close();
    xaconn2.close();

    conn1 = null;
    conn2 = null;
    xaconn1 = null;
    xaconn2 = null;
     }// XARun

     Xid createXid(int bids) throws XAException{
    byte[] gid = new byte[1]; gid[0] = (byte)9;
    byte[] bid = new byte[1]; bid[0] = (byte)bids;
    byte[] gtrid = new byte[64];
    byte[] bqual = new byte[64];

    System.arraycopy(gid,0,gtrid,0,1);
    System.arraycopy(bid,0,bqual,0,1);

    Xid xid = new OracleXid(0x1234,gtrid,bqual);
```

```
    return xid;
     }// createXid

     XADataSource getXADataSource(String dbAddr,int
port,String sid
          ,String userId,String password)
    throws SQLException,XAException
     {
    OracleDataSource oxds = new OracleXADataSource();
    String url = "jdbc:oracle:thin:@127.0.0.1:1521:ORCL";
    oxds.setURL(url);
    oxds.setUser(userId);
    oxds.setPassword(password);

    return (XADataSource)oxds;
     }// getXADataSource

}// class OracleXARun
```

〈 예제 Oracle JDBC XA Driver 를 이용한 2PC 구현 〉

이 예제를 실행 하기 위해서 몇 가지 준비가 필요하다.

1. 이 예제를 테스트한 환경은 하나의 데이타베이스에서 (하나의 RM)에서 테스트를 했다. 먼저 첫 번째 DB에 accountFrom 이라는 테이블을, 두 번째 DB에는 accountTo 라는 이름의 테이블을 만들도록 하자.

```
// 첫 번째 DB에 만들어야 할 테이블
CREATE TABLE ACCOUNTFROM(
        ACCOUNTNO NUMBER,
        BALANCE NUMBER,
        PRIMARY KEY(ACCOUNTNO)
);
INSERT INTO ACCOUNTFROM VALUES(1000,10000);

// 두 번째 DB에 만들어야 할 테이블
CREATE TABLE ACCOUNTTO(
        ACCOUNTNO NUMBER,
        BALANCE NUMBER,
        PRIMARY KEY(ACCOUNTNO)
);
```

〈 Oracle JDBC XA Driver를 실행하기 위한 데이타 베이스 테이블 〉

2. final static으로, 선언되어 있는 첫 번째 DB와 두 번째 DB의 연결 정보를 알맞게 수정한다.

```
    final static String DB_XXXX_ADDR = "127.0.0.1"; // Oracle DB IP
    final static int DB_XXXX_PORT = 1521; // Oracle DB Port
    final static String DB_XXXX_SID = "ORCL"; // Oracle DB SID
    final static String DB_XXXX_USER = "scott"; // Oracle DB User ID
    final static String DB_XXXX_PASSWORD= "tiger"; // Oracle DB User Password
```

3. 컴파일하고 실행한다.
실행을 하고 sqlplus를 이용해서 accountfrom과 accountto를 select 해보면, accountfrom에는 accountno가 1000인 row의 balance는 1000이 빠졌을 것이고, accountto 테이블에는 accountno가1000이고, balance가 1000이 row가 생성되었을 것이다.

다시 실행을 해 보면, accountto 테이블에 이미 accountno가 1000이 row가 있기 때문에, 무결성 제약조건에 의해서 오라클 에러를 출력하고, 트렌젝션을 rollback하게 된다.

지금까지, JDBC XA Interface를 이용해서 2PC를 구현하는 것과, JAVA에서의 Transaction처리 과정에 대해서 살펴보았다. 쉽지는 않은 내용이지만, 이 내용을 잘 알아놓으면, 나중에 복잡한 트렌젝션 프로그래밍을 할 때, 도움이 될 것이다.

2) JTA Based Transaction Programming

다음으로는 JTA Interface를 이용해서 실제 Transaction 프로그래밍을 하는 과정을 알아보자. 보통 EJB나 일반 애플리케이션에서 Web Application Server(WAS)등의 Transaction Manager등을 이용해서 Transaction 관리를 할 때, JTA를 이용해서 프로그래밍을 한다.

Transaction Manager가 있을 때에는 특정한 목적이 아니라면, 굳이 직접 XA 인터페이스를 사용해서 프로그래밍을 할 필요 없이, 이 JTA만 사용하면 보다 쉽게 Transaction 프로그래밍을 할 수 있다.

JTA 프로그래밍을 하기 위한 기본적인 순서를 살펴 보면

1. User Transaction 객체(tx)를 얻어온다.
2. 얻어온 User Transaction 객체 tx를 이용하여, User Transaction을 시작한다. tx.begin();
3. 각각의 Resource (DBMS, JMS)과 연결을 한 후, 각각의 쿼리 등의 작업을 수행한다.
   Resource들과 연결할 때 주의해야 할 사항이 몇 가지가 있다. 일단 Resource Adapter는 XA를 지원해야 한다. Oracle의 경우, JDBC Driver를 oracle.jdbc.driver.OracleDriver가 아니라, oracle.jdbc.xa.client.OracleXADataSoruce를 사용해야 한다.

   데이타베이스 Connection (java.sql.conn)을 얻어 올 때에도 DriverManager.getConnection을 이용 하는 것이 아니라, WAS에 의해서 관리되는 Resource Factory를 통해서 얻어와야 된다. DBMS의 경우에는 DataSource 등을 통해서 Connection을 얻어오게 된다.

4. 만약 에러가 발생했으면 tx.rollback();으로 전체 트렌젝션을 roll back시킨다.
5. 정상적으로 트렌젝션이 완료 되었을 경우에는 tx.commit(); 으로 전체 트렌젝션을 commit한다.

그럼 간단한 예제를 만들어 보기로 하자.

예제는 앞에서 만들었던 내용과 똑같이 하나의 DBMS의 ACCOUNTFROM이라는 테이블에서 ACCOUNTNO가 1000인 레코드의 BALANCE값을 1000을 감소시키고, ACCOUNTTO라는 테이블에 ACCOUNTNO가 1000, BALANCE가 1000인 ROW를 INSERT하는 내용이다.

이전 예제와 같이 각각의 DBMS에 ACCOUNTFROM과, ACCOUNTTO라는 테이블을 생성하자.

생성이 끝났으면 WebLogic을 설정하자

1. JDBC Connection Pool을 설정한다.

웹로직을 실행하고, 웹로직 CONSOLE에 접속한다.
(http://localhost:7001/console)
좌측 네비게이션 트리에서, Services > JDBC > ConnectionPools를 선택하고, Configure a new JDBC Connection Pool을 선택한다.

Configure>General Tab에서
Name : OracleXAPool
URL : jdbc:oracle:thin:@127.0.0.1:1521:ORCL (각자 환경에 맞게 변경)
Driver Classname:
oracle.jdbc.xa.client.OracleXADataSource (일반 JDBC 드라이버랑 다르기 때문에 주의!!)
Properties :
user=scott
password=tiger

입력한 후, Targets Tab에서 해당 서버에 Targeting을 한다.

2. TXDataSource를 설정한다.

좌측 네비게이션 트리에서, Services > JDBC > TX Data Sources를 선택하고, Configure a new JDBC TX Data Source을 선택한다.
일반 DataSource를 사용하는 것이 아니라, 웹로직의 경우에는 global transaction을 사용하기 위해서, TX DataSource를 사용한다.

Configuration 탭에서 내용을 입력한다.
Name : OracleTXDS
JNDI Name : OracleTXDS
PoolName : OracleXAPool (앞에서 지정한 풀 이름)

생성된 TX Data Source를 Target Tab에서 해당 서버로 지정한다.

```
<%@ page contentType="text/html;charset=euc-kr"
     import="javax.naming.*
       ,javax.transaction.*
       ,java.sql.*
       ,javax.sql.*
       ,java.util.*"
%>
<%!

  // DB Connection Info
  final static String DB_FROM_DATASOURCE=
"OracleTXDS";
  final static String DB_TO_DATASOURCE="OracleTXDS";
```

```
  //final static String DB_TO_DATASOURCE=
"OracleTXDS_Chaeju";

  void JTARun()
    throws Exception
  {

    // step 1. JNDI Lookup and get UserTransaction
Object
    Context ctx = null;
    Hashtable env = new Hashtable();

    // Parameter for weblogic
    env.put(Context.INITIAL_CONTEXT_FACTORY,
       "weblogic.jndi.WLInitialContextFactory");
    env.put(Context.PROVIDER_URL,"t3://localhost:7001");
    env.put(Context.SECURITY_PRINCIPAL,"system");
    env.put(Context.SECURITY_CREDENTIALS,"weblogic");

    ctx = new InitialContext(env);
    System.out.println("Context Created :"+ctx);

    // step 2. get User Transaction Object
    UserTransaction tx = (UserTransaction)

ctx.lookup("javax.transaction.UserTransaction");
    // step 3 start Transaction
    System.out.println("Start Transaction :"+tx);
    tx.begin();

    try{
    // step 4. doing query
      // step 4-1. get Datasource
      DataSource xads1 =
(DataSource)ctx.lookup(DB_FROM_DATASOURCE);
      DataSource xads2 =
(DataSource)ctx.lookup(DB_TO_DATASOURCE);

      System.out.println("Datasource from :"+xads1);
      System.out.println("Datasource to :"+xads2);

      // step 4-2. get Connection
      Connection conn1 = xads1.getConnection();
      Connection conn2 = xads2.getConnection();

      conn1.setAutoCommit(false);
```

```
        conn2.setAutoCommit(false);

        System.out.println("Connection from :"+conn1);
        System.out.println("Connection to :"+conn2);

        // step 4-3 execute query
        Statement stmt1 = conn1.createStatement();
        String sql = "update accountfrom set
balance=balance-1000 where accountno=1000";
        stmt1.executeUpdate(sql);
        System.out.println("Statement 1 passed");

        Statement stmt2 = conn2.createStatement();
        sql = "insert into accountto values(1000,1000)";
        stmt2.executeUpdate(sql);
        System.out.println("Statement 2 passed");

        // step 4-4. close connection
        stmt1.close();
        conn1.close();

        stmt2.close();
        conn2.close();

        System.out.println("connection closed");
        System.out.println("tx :"+tx);

        tx.commit();
        System.out.println("Transaction Commited");
      }catch(Exception e){
        System.out.println("Transaction rolled back due to :"
+e);
        tx.rollback();
      }// try-catch
      System.out.println("End Transaction");

      ctx.close();
   }// JTARun

%>
<%
    JTARun();
%>
```

〈예제, WebLogic을 이용한 JTA User Transaction 2PC 구현〉

실행결과는 웹로직 콘솔에 나타난다. (WINDOWS의 경우에는 DOS창이나 로그에.)

```
Context Created :javax.naming.InitialContext@585571
Start Transaction
:ClientTM[myserver+10.136.0.133:7001+mydomain+]
Datasource from
:weblogic.jdbc.common.internal.RmiDataSource@334bd3
Datasource to
:weblogic.jdbc.common.internal.RmiDataSource@334bd3
Connection from
:weblogic.jdbc.rmi.SerialConnection@1b12fb
Connection to
:weblogic.jdbc.rmi.SerialConnection@7b3d5c
Statement 1 passed
Statement 2 passed
connection closed
tx :ClientTM[myserver+10.136.0.133:7001+mydomain+]
Transaction Commited
    End Transaction
```

〈 정상적인 치료 결과 〉

7. EJB Transaction Model

지금까지 트렌젝션이 무엇인지, 그리고 Java에서 어떤 식으로 트렌젝션이 구현되는지에 대해서 살펴보았다. 이번에는 J2EE의 핵심 기술이라고도 할 수 있는 EJB에서의 트렌젝션 처리에 대해서 알아보기로 하자.

1) Declarative transaction demarcation (선언적 트렌젝션 정의)

EJB에서의 트렌젝션은 앞에서 우리가 봐왔던 복잡한 트렌젝션 처리들을 EJB내부에서 대부분 EJB Container를 통해서 자동으로 처리해주도록 되어 있다. EJB에서는 트렌젝션 자동 처리를 위해서 몇 가지 트렌젝션 모델을 미리 정해놓고 있는데, 이를 Declarative transaction이라고 하고, EJB Deployment Descriptor에만 그 내용을 정의해 놓으면 간단하게 트렌젝션을 핸들링 할 수 있다. 그럼 지금부터, Declarative Transaction의 종류와, 그 내용에 대해서 살펴보도록 하자.

a) Required:

Required로 정의된 메소드들은 모두 트렌젝션내에서 수행되게 된다. 해당 메소드가 CALL이 되었을때, 이미 트렌젝션이 수행되고 있을 경우에는 그 트렌젝션이 Propagation이 돼서, 그 트렌젝션 내에서 수행이 되는 것이고, 수행중인 트렌젝션이 없을 경우에는 새로운 트렌젝션을 만들어서, 그 새로운 트렌젝션 내에서 메소드를 수행한다.

Required 속성을 사용하면 수행되는 액션이 트렌젝션의 보호 아래서 수행되기 때문에, 데이타를 바꾸는 작업(update, delete)중에 유용하게 사용될 수 있다.

b) Required New:

Required New로 정의된 메소드는 언제나 새로운 트렌젝션내에서 수행하게 된다. 해당 메소드가 호출 되었을 때, 호출한 클라이언트가 트렌젝션을 시작하지 않은 상태이면 (즉 트렌젝션 내가 아니라면) 새로운 트렌젝션을 시작하게 된다.

반대로, 만약에 클라이언트가 트렌젝션을 시작해서, 해당 메소드가 호출되는 시점이 트렌젝션 중이라면, 이미 수행중인 트렌젝션을 잠시 중지하고, 호출된 메소드를 처리하기 위한 새로운 트렌젝션을 처리하여, 메소드가 수행된 후에 이전에 중지된 트렌젝션을 다시 수행한다.

이렇게 되면, 만약 Required New 속성으로 지정된 트렌젝션은 외부 트렌젝션과 상관없이 작동이 된다. 예를 들어 설명하자.

```
TX_A_BEGIN
DoSomething1

RequiredNewMethod(); // Required New – TX_B

DoSomething2
TX_A_END
```

TX_A안에서 RequiredNewMethod()라는 메소드가 Requried New라는 속성으로 정의되었을 경우, DoSomething1까지 수행한 후, TX_A를 SUSPEND한 다음에, RequiredNewMethod()라는 메소드 안에 정의된 내용들을 새로운 트렌젝션 TX_B 내에서 수행하고, 수행된 내용을 commit 한다. 그리고 TX_A로 복귀하여, RESUME한 후, DoSomething2를 수행한 후, 트렌젝션 A를 끝낸다.

TX_B가 TX_A 안에 포함된 형태이기는 하지만, TX_B가 rollback되었다고, TX_A가 rollback되지는 않는다(failure localized).
반대로, DoSomething2에서 TX_A가 rollback되었다고, TX_B가 rollback되지 않는다. 이런 특성은 log를 남기는 기능등에서 유용하게 사용될 수 있다. 즉 트렌젝션내에 포함은 되어 있지만 전혀 별도의 트렌젝션으로 수행이 되는 것이다.

대신 무조건 새로운 트렌젝션을 생성하기 때문에, 그에 따른 시스템 오버헤드가 있기 때문에 남용할만한 속성은 아니다.

c) Supports

Support 속성으로 정의된 메소드는 호출하는 클라이언트의 트렌젝션 속성을 그대로 이어받는다. 즉 Support 속성으로 정의된 메소드를 클라이언트가 트렌젝션 범위 내에서 호출 했으면 이 메소드도 같은 트렌젝션 범위 내에서 수행이 되고, 만약 클라이언트가 트렌젝션 없이 메소드를 호출했으면 마찬가지로 해당 메소드 역시 트렌젝션이 없이 수행이 된다.

d) NotSupported:

NotSupported 속성으로 정의된 메소드는 트렌젝션범위 밖에서 수행이 된다. 호출하는 클라이언트가 트렌젝션을 가지고 있지 않고 메소드를 호출하였다면 마찬가지로 트렌젝션이 없이 호출되나, 만약에 클라이언트가 트렌젝션 범위 내에서 NotSupported 속성으로 정의된 메소드를 호출하면, 기존의 트렌젝션을 suspend한후, 메소드의 내용을 처리한 후에 다시 기존의 트렌젝션을 resume하여 기존의 트렌젝션을 마저 수행하게 된다.

e) Mandatory:

Mandatory 속성으로 정의된 메소드는 꼭 트렌젝션 SCOPE내에서 실행되어야 한다. 즉 Mandatory로 정의된 메소드는 그 메소드를 호출하는 클라이언트에서 꼭 트렌젝션을 시작해서, 메소드가 트렌젝션 범위 내에서 수행되도록 해야 한다.

만약 Mandatory로 정의된 메소드를 트렌젝션을 시작하지 않은 상태로 호출할 경우에는 TransactionRequiredException이나 TranscationRequredLocalException이라는 에러를 내게 된다.

f) Never:

Never 속성으로 정의된 메소드는 기존 트렌젝션 범위 내에서 실행될 수 없으며, 메소드 자체도 새로운 트렌젝션을 생성하지 않는다. 즉 Never로 정의된 메소드는 트렌젝션 범위 내에서 수행될 수 없다.

이 속성은 client managed transaction을 수행하지 못하게 서버 쪽에 명시적으로 정의해줄 수 있으며, 해당 메소드가 다른 트렌젝션에 포함되지 않도록 정의할 때 유용하게 사용된다.

그러면 지금까지 살펴본 Declarative Transaction 속성들을 간단하게 표로 정리해 보도록 하자.

Transaction Attribute	Client's Transaction	EJB Method's Transaction	Comment
Required	NO_TRAN	NEW TRAN	
	TRAN A	TRAN A	
Required New	NO_TRAN	NEW TRAN	
	TRAN A	NEW TRAN	
Support	NO_TRAN	NO_TRAN	
	TRAN A	TRAN_A	
Not Supported	NO_TRAN	NO_TRAN	
	TRAN A	NO_TRAN	
Mandatory	NO_TRAN	Error	
	TRAN A	TRAN A	
Never	NO_TRAN	NO_TRAN	
	TRAN A	Error	

(※ Client's Transation은 트렌젝션을 시작하는 쪽에서의 Transaction 상태를 나타낸다. NO_TRAN은 트렌젝션이 없이 시작하는 경우를 이야기 한다. EJB Method's Transaction 은 트렌젝션 속성과 Client's Transaction 에 따라서, EJB Method가 어떤 트렌젝션으로 실행되는지를 나타낸다. TRAN_A는 Client 트렌젝션을 그대로 이어받은 경우이며, NEW_TRAN은 Client 트렌젝션과 상관없이 새롭게 트렌젝션을 생성해서 수행하는 경우를 나타낸다.)

2) Transaction 속성들의 사용

그러면, 이 Declarative Transaction Model들은 어떤 상황에서 사용이 될까? 앞에서도 중간 중간 언급 했지만, 이해를 돕기 위해서, 각각의 트렌젝션 모델이 유용하게 사용될 수 있는 내용을 간단하게 정리해보자

- o DATA를 READ할 때 – Supports
- o DATA를 UPDATE할 때 – Required
- o 트렌젝션을 지원하지 않는 Resource를 사용하고자 할 때 – NotSupported

※ 참고로 MDB (Message Driven Bean)에서는 Required 와 Not Supported 두 가지 속성만이 사용 가능하다.

3) EJB Deployment Descriptor에서 Transaction 속성 정의

이렇게 정의된 트렌젝션 속성들을 실제로 EJB에 적용하기 위해서는 EJB Deployment Descriptor 중 ejb-jar.xml에 EJB의 각 메소드 별로 정의할 수 있으며, 그 형식은 다음과 같다.

ejb-jar.xml에서 <assembly-descriptor> 아래의 <container-transaction> 속성을 아래와 같이 정의 한다.

```
<container-transaction>
<method>
 <ejb-name>EJBNAME</ejb-name>
 <method-name>EJBMETHOD_NAME</method-name>
</method>
<trans-attribute>여기에 앞에서 설명한 트렌젝션 속성을 쓴
다.</trans-attribute>
</container-transaction>
```

4) Bean Managed Transaction

지금 까지는 J2EE에서 미리 지정해놓은 Declarative Transaction Model에 대해서 살펴보았다. 이외에도 EJB에서는 EJB Container가 아니라, 직접 사용자가 프로그래밍을 통해서 Transaction을 관리할 수 있는데, 이를 BMT (Bean Managed Transaction)이라고 한다.

Bean Managed Transaction은, email이나, file등과 같이 실제적으로 트렌젝션 기능을 지원하지 않는 resource에 대해서, 트렌젝션 처리를 하도록 임의적으로 구현하거나, 내지는 stateful session bean등에서, 하나의 메소드가 아니라, 여러개의 메소드에 걸쳐서, Transaction을 진행하고자 할 때, 매우 유용하게 사용할 수 있다.

Bean Managed Transaction을 프로그래밍 하기 위해서는 직접 EJB Context로 부터, UserTransaction Context를 얻어서 Transaction을 관리해야 하는데. 그 절차와 방법은 간단하다.

먼저, javax.transaction.UserTransaction 을 EJB Context로 부터 얻어온다.
다음 얻어온 UserTransaction 객체의 begin 메소드를 통해서 트렌젝션을 시작한다.
다음 트렌젝션 관련된 명령을 수행한 다음, 트렌젝션이 끝났으면, commit을 오류가 발생한 경우에는 rollback을 하면 된다.

그러면 간단한 샘플 코드를 통해서 어떻게 Bean Managed Transaction이 구현되는지 살펴보기로 하자.

```
// Bean Managed Transaction Session Bean
import javax.transaction.*;
  :

public class BeanManagedTransaction implements
SessionBean{
 SessionContext ctx = null;

 public void setSessionContext(SessionContext ctx){
  this.ctx = ctx;
 }// setSessionContext

 public void startTransaction(){
  UserTransaction utx = ctx.getTransaction();
  utx.begin(); // 여기서 트렌젝션을 시작한다.
  }

 public void doSomethingA() throws Exception{
      :
  try{
    // do something about db transactions...

  }catch(Exception e){
      :
    UserTransaction utx = ctx.getTransaction();
    utx.rollback();
    throw new SomeException();
  }finally{
    conn.close();
  }// try-catch
 }// doSomethingA

 public void doSomethingB() throws Exception{
  // simliar to doSomethingA
  // do something about db transactions..
 }// doSomethingB


 public void endTransaction(){
  UserTransaction utx = ctx.getTransaction();
try{
      :
    utx.commit(); // 트렌젝션을 끝내고, 그 내용을 commit한다.
  }catch(Exception e){
    utx.rollback();
  }
 }

}// class BeanManagedTransaction
```

```
// in client code

BeanMangedTransaction bmt = .....;
 :
try{
bmt.startTransaction();
bmt.doSomethingA();
bmt.doSomethingB();
bmt.endTransaction();
}catch(Exception e){
  // transaction has failed
}
```

이 예제는 Session Bean에서, 트렌젝션을 시작하고, 두 개의 메소드를 통해서 DB작업을 한 후에, 트렌젝션을 종료하는 로직을 간단하게 표현한 내용이다. (위에는 Session Bean 클래스, 아래는 이 EJB를 호출하는 Client클래스 내용이다.)

Client 에서 BeanManagedTransaction SessionBean 객체 bmt를 생성해서 이 내용들을 모두 호출 하는데, 먼저 startTransaction에서 User Transaction을 생성하고, doSomethingA와 doSomethingB에서, DB작업을 수행한다. 수행 중에 만약 에러가 발생하면, UserTransactionContext를 얻어서, rollback을 하고, Exception을 Throw해서, 진행중인

Transaction을 stop하도록 한다.트렌젝션이 끝나면, commit을 실행하여, 내용을 반영한다. 잘 보면 하나의 Method가 아니라, 4개의 Method에 걸쳐서 트렌젝션이 수행되고 있다.

여기서 주의할 점 중의 하나가 이 Session Beans는 Stateful Session Bean이라는 것이다. Stateless의 경우에는 상태가 유지가 되지 않기 때문에, 여러 메소드를 통해서 트렌젝션을 수행할 경우에는 제대로 수행이 되지 않는다.

이 Bean managed transactions 속성은 ejb-jar.xml 에 정의해 줘야 하는데, 예제에서 사용한 Bean Managed Transaction Bean의 경우에는 다음과 같이 transaction-type을 Bean으로 지정하면 된다.

```
〈ejb-jar〉
 :
  〈session〉
    〈display-name〉BeanManagedTransaction〈/display-
name〉
    〈ejb-name〉BeanManagedTransaction〈/ejb-name〉
      :
    〈session-type〉Stateful〈/session-type〉
    〈transaction-type〉Bean〈/transaction-type〉
     :
  〈/session〉
```

Bean Managed Transaction 프로그래밍을 할 때는 몇 가지 주의할 점이 있는데, 첫 번째는 앞에서도 설명했듯이, 여러 Method에 걸쳐서 트렌젝션이 수행될 때는 Stateful Session Bean 만을 사용해야 한다. Stateless Session Bean의 경우에는 시작된 User Transaction은 반드시 시작된, 그 메소드 안에서만 종료되어야 한다.

또한 Bean Managed Transaction은 모든 종류의 Bean에서 가능한 것이 아니라 Session Bean과 Message Driven Bean에서만 사용이 가능하다. Entity Bean에서는 불가능하다.

5) Transaction in Entity Beans

그렇다면, 여기서 자연스럽게 하나의 질문이 생길 수 있는데, Entity Bean에서는 어떻게 Transaction을 처리하는지 살펴볼 필요가 있다.

Entity Beans에서는 ejbLoad와, ejbStore에 대한 처리를 Declarative Transaction Model 중에서, Support 속성으로 처리하게 되어 있다. 즉 Entity Bean을 호출하는 Client의 Transaction을 그대로 사용하여 EJB를 호출하게 된다.

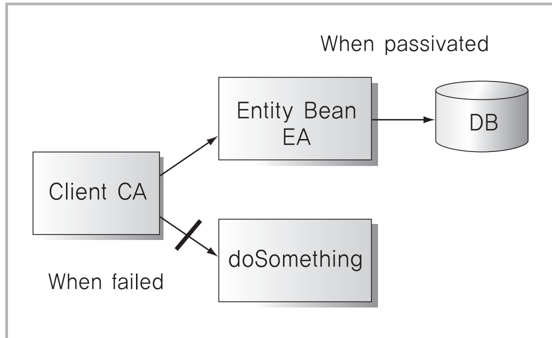
Client가 Transaction A 내에서 사용되고 있으면, 그에 의해 호출되는 Entity Bean의 ejbLoad와 ejbStore는 Transaction A내에서 실행된다. 그럼 만약에 Client가 Transaction없이 수행될 때는, 역시 Transaction 없이 실행이 된다.
이런 경우 몇 가지 문제를 일으킬 수 있는데, Bean이 데이타를 기록하는 과정이 Transaction 없이 수행되기 때문에, Entity Bean이 ejbStore등을 이용해서 Data를 write하는 과정에서, 다른 Entity Bean이 동시에 Data를 Writing할 때 데이타가 잘못 들어가거나, 또는 다른 Client에서 같은 Data를 write할 때도 같은 문제를 발생시킬 수 있다. 일반적으로 트렌젝션 없이 수행될 때 나올 수 있는 문제들이 그대로 발생한다. (즉 트렌젝션이 보장이 안 되는 상황이 발생한다.)

그렇다면 Entity Bean이 실행되는 Transaction 속성이 Required면 되지 않을까 하는 생각을 할 수 있다. Required 속성이면 호출하는 client가 transaction이 없을 때 새로운 transaction을 만들어서 그 내에서 업데이트 작업을 수행한다. 잘 알고 있듯이 Entity Bean은 Activation과, Passivation이라는 작업이 있다, (이 내용에 대해서는 좀 더 알고 싶다면, EJB관련 서적을 미리 참고하기 바란다.) 그리고 WAS에 따라서 그 내용을 별도로 Caching하기도 한다. Entity Bean이 Passivation이 될때, Entity Bean이 Client와 다른 Transaction으로 수행되고 있다면, EJB는 그 변경된 내용을 아직 client의 작업이 아직 끝나지 않았음에도 DB에 writing하고 commit을 해버린다.

예를 들어 설명해 보자, 은행 계좌 이체를 수행하는 client CA가 있다. ClientCA는 EntityBean EA를 호출하여 값을 설정한 후, doSomething이라는 함수를 호출하여 어떤 작업을 수행한 후, 다신 EntityBean EA를 호출하여 DB값을 업데이트하는 시나리오를 가지고 있다고 하자.

그런데, EntityBeanEA를 update하던 중에, Entity Bean EA의 Passivation이 일어 났다고 하자, 그러면 EJB 내용은 DB에 update가 될 테고, Entity Bean EA는 새로운 트렌젝션을 만들어서 수행이 되기 때문에, 그 내용을 DB에 commit해 버린다.

그 후, client CA에서 doSomething이라는 함수를 이용하여 무엇인가를 실행하려고 할때, doSomething으로의 call이 실패했을 경우 어떻게 될까? Entity Bean EA에서 이미 commit을 해버렸기 때문에, 의도하지 않은 데이타가 들어갈 수 있다. 실제로 프로그램을 작성한 사람은 완전하게 client CA의 call이 끝나고, 그 내용을 DB에 반영하려고 했는데도, passivation에 의해서 DB 업데이트가 발생해 버린다.



이런 이유 때문에 EJB Entity Bean Container Management Persistance(CMP)를 사용할 경우에는 가급적이면 해당 EJB Entity Bean을 호출 하는 client가 transaction을 가지고 호출하도록 해야 한다.

6) EJB에서 Transaction 관련 API

EJB에서는 Transaction에 관련하여 몇가지 Method를 사용한다, 알아 놓으면 매우 편리하기 때문에, 하나씩 집어 보도록 하자.

a) setRollbackOnly()/getRollbackOnly()

setRollbackOnly()라는 메소드를 호출하면, 해당 Transaction의 경우에는 commit이 되지 않는다. 트렌젝션에서 MA, MB, MC라는 세 개의 메소드를 호출하는 도중에, MB에서 setRollbackOnly 메소드가 호출 되면, 그 순간 트렌젝션이 rollback이 되는 게 아니라, MC까지 수행하고, Transaction의 맨 마지막 순간에, rollback을 한다.

setRollbackOnly는 이와 같이, rollback은 해야 하지만 뒤에 있는 Logic까지 수행해야 하는 경우에, Transaction의 마지막 단계에 rollback을 하도록 mark해주는 역할을 한다.

이 메소드는 Container Management Transaction에서만 사용할 수 있으며, 그 중에서도 Required, RequiresNew, Mandatory 속성에서만 사용할 수 있다.

그리고, rollback이 마크된 여부를 파악하기 위해서, getRollbackOnly()라는 메소드를 지원한다.

b) afterBegin,afterCompletion,beforeCompletion

EJB에서는 Session Bean을 이용해서 Container Managed Transaction을 구현할 때, javax.ejb.SessionSynchronization interface를 이용해서, 몇 가지 call back event 메소드를 제공한다. 트렌젝션 수행 상태, 시작/Commit 전후에 어떤 특별한 액션을 취하고 싶다면 이 interface를 implement하면 된다.

```
public interface javax.ejb.SessionSynchronizaton{
    public void afterBegin() throws
EJBException,RemoteException;
    public void afterCompletion(boolean commited) throws
EJBException,RemoteException;
    public void beforeCompletion() throws
EJBException,RemoteException;
}// SessionSynchronization
.
```

afterBegin()은 새로운 트렌젝션이 생기고, 트렌젝션을 시작하자 마자, 맨 처음 불려지는 메소드이다. beforeCompletion은 commit이 되기 전에 최종적으로 실행되는 메소드이다.
afterCompletion은 commit/rollback이 되자 마자 수행되는데, commit이 된 경우에는 commited value가 true, 반대의 경우에는 false로 체크된다. (afterCompletion은 트렌젝션이 끝난 후에 실행되기 때문에 트렌젝션 Context 밖에서 수행된다.)

이 메소들은 주로, Data sync, caching, format 체크 등에 유용하게 사용할 수 있다.

7) Programming Two-Phase Commit using EJB

지금부터 웹로직과 EJB를 이용해서 간단하게 두 개의 Oracle DB에 Two Phase commit을 구현해 보도록 하자. 우리가 구현할 내용은 지난주에 구현했던 것과 같은 내용을 이번에는 EJB로 구현하는 내용이다.

A계좌에서 계좌이체를 해서 B계좌로 금액을 옮기는 내용을 Entity Bean과 Session Bean을 이용해서 구현할 것이다.

a) ORACLE에 해당 계정에 XA 권한 주기

step 1. sql plus에 시스템 사용자로 로그인한다.

```
% sqlplus  sys/CHANGE_ON_INSTALL@<DATABASE
ALIAS NAME>
```

step 2. 사용자 개정에 XA를 쓸 수 있는 권한을 부여한다.

```
% grant select on DBA_PENDING_TRANSACTIONS to
public
```

b) ORACLE에 테이블 만들기

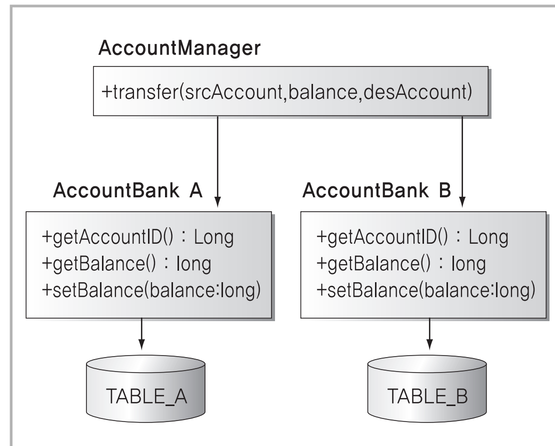
2PC를 구현하기 때문에, 두 개의 다른 ORACLE DB에 테이블을 만들어야 한다.

```
DB A :
SQL> create table TABLE_A(
  2  ID NUMBER UNIQUE,
  3  ACCOUNT NUMBER,
  4  PRIMARY KEY(ID)
5  );
6 insert into TABLE_A VALUES(1,1000000);

DB B:
SQL> create table TABLE_B(
  2  ID NUMBER UNIQUE,
  3  ACCOUNT NUMBER,
  4  PRIMARY KEY(ID)
5  );
```

c) EJB 코딩

그러면 EJB를 직접 만들어보자, AccountManager, AccountBankA, AccountBankB 3개의 EJB를 만들것이다. AccountManager EJB는 Session Bean으로 transfer라는 메소드를 가지고 있다. transfer메소드는 AccountBankA EJB를 이용하여 TABLE A에서 balance 만큼의 금액을 srcAccount 계정에서 빼서, AccountBankB EJB를 이용하여, 다른 DB TABLE_B에 update 하도록 되어 있다.



그럼 AccountBankA 빈즈를 보자.
AccountBankA 빈즈는 간단한 EntityBean이다. 값을 set/get하는 역할만을 수행한다. AccountBankB의 내용도 AccountBankA 빈즈와 같기 때문에 AccountBankB의 소스 코드는 생략하기로 한다.

```
// AccountBankA.java
package com.javastudy.twopc;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;

public interface AccountBankA extends EJBObject {
    public  Long getAccountId() throws RemoteException;
    public  long getBalance() throws RemoteException;
    public  void setBalance(long balance) throws RemoteException;

}
```

```
// AccountBankA_Home.java
package com.javastudy.twopc;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;
import javax.ejb.FinderException;

public interface AccountBankA_Home extends EJBHome{
    public AccountBankA create(Long accountID,long balance)
        throws RemoteException,CreateException;

    public AccountBankA findByPrimaryKey(Long accountID)
        throws RemoteException,FinderException;
}
```

```
// AccountBankA_EJB.java
package com.javastudy.twopc;

import javax.ejb.CreateException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;

public abstract class AccountBankA_EJB implements EntityBean {
      public abstract Long getAccountId();
      public abstract void setAccountId(Long accountId);

      public abstract long getBalance();
      public abstract void setBalance(long balance);

      public Long ejbCreate(Long accountId,long balance)
           throws CreateException {
              setAccountId(accountId);
              setBalance(balance);

              return null;
           }// ejbCreate

      public void ejbPostCreate(Long accountId,long balance)
           throws CreateException{}

      public void ejbActivate() {}
      public void ejbLoad() {}
      public void ejbPassivate() {}
      public void ejbRemove(){}
      public void ejbStore(){}
      public void setEntityContext(EntityContext ctx){}
      public void unsetEntityContext(){}

}
```

다음은 이 AccountBankA와 AccountBankB 빈즈를 컨트롤 하는 AccountManager Session Bean이다.

```
//AccountManager.java
package com.javastudy.twopc;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;

public interface AccountManager extends EJBObject{
      public String transfer(long srcAccount,long balance,long
desAccount)
           throws RemoteException;
}


//AccountManager_EJB.java
package com.javastudy.twopc;

import javax.ejb.CreateException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.naming.InitialContext;
import javax.naming.Context;
import javax.naming.NamingException;
import javax.rmi.PortableRemoteObject;

public class AccountManager_EJB implements SessionBean
{
      public String transfer(long srcAccount,long amount,long
desAccount){
           AccountBankA accountA = null;
           AccountBankB accountB = null;
           String result_msg = null;
           // AccountA와 AccountB Entity Bean을 생성한다.
           accountA = getAccountBankA(srcAccount);
           if(accountA == null) return "cannot cat account A:"
+srcAccount;

           accountB = getAccountBankB(desAccount);
           if(accountB == null) return "cannot cat account B:"
+desAccount;

           /* start transaction */
           try{
           // AccountA와 AccountB Entity Bean에서 각각의 BALANCE값을
읽어오고
           // A에서 balnce만큼을 빼서 B에 더한다.

           long balanceA = accountA.getBalance();
           long balanceB = accountB.getBalance();

           // 현재 내용을 출력한다.
System.out.println("current balance A :"+balanceA);
System.out.println("current balance B :"+balanceB);
System.out.println("amount:"+amount);
           accountA.setBalance(balanceA-amount);
           accountB.setBalance(balanceB+amount);
           }catch(Exception e){
              System.out.println("Error occur during
AccountManager_EJB.transfer:"+e.toString());
              return e.toString();
           }
           /* end transaction */

           return null;
      }// transfer

      private AccountBankA getAccountBankA(long accountID){
           AccountBankA accountbank = null;

           try{
              Context ctx = getContext();
              Object obj =
ctx.lookup("com.bea.twopc.AccountBankA_Home");

              System.out.println("AccountBank_A lookup is :"+obj);


              AccountBankA_Home home =
```

```
                    (AccountBankA_Home)PortableRemoteObject.narrow(obj,
                                                    AccountBankA_Home.class);
               accountbank = home.findByPrimaryKey(new
Long(accountID));
          }catch(Exception e){
               System.out.println("error during
AccountManager_EJB:getAccountBankA:"
                                        +e.toString());
               return null;
          }

          return accountbank;
     }// getAccountBankA

     AccountBankB getAccountBankB(long accountID){
          AccountBankB accountbank = null;

          try{
               Context ctx = getContext();
               Object obj =
ctx.lookup("com.bea.twopc.AccountBankB_Home");

               System.out.println("AccountBank_B lookup is :"+obj);

               AccountBankB_Home home =

(AccountBankB_Home)PortableRemoteObject.narrow(obj,

AccountBankB_Home.class);

               accountbank = home.findByPrimaryKey(new
Long(accountID));
          }catch(Exception e){
               System.out.println("error during
AccountManager_EJB:getAccountBankB:"
                                        +e.toString());
               return null;
          }// try-catch

          return accountbank;
     }// getAccountBankB

     private Context getContext()
          throws NamingException
     {
          Context ctx = new InitialContext();
          return ctx;
     }// getContext

     // callback methods
     public void ejbRemove(){}
     public void ejbActivate(){}
     public void ejbPassivate(){}
     public void setSessionContext(SessionContext sc){}
     public void ejbCreate() throws CreateException {}

}// AccountManager_EJB
```

```
//AccountManagerHome.java
package com.bea.twopc;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface AccountManagerHome extends EJBHome{
     AccountManager create() throws CreateException,RemoteException;
}
```

여기까지 했으면 총 9개의 파일을 만들었을 것이다.

AccountBankA.java
AccountBankA_EJB.java
AccountBankA_Home.java
AccountBankB.java
AccountBankB_EJB.java
AccountBankB_Home.java
AccountManager.java
AccountManagerHome.java
AccountManager_EJB.java

이 9개의 파일을 컴파일 하자.

```
% javac -d . *.java
```

컴파일이 끝나면, ~/com/javastudy/twopc라는 디렉토리 아래 총 9개의 class 파일이 생성 되었을 것이다.

d) EJB 패키징

EJB들을 패키징하기 위해서 Deployment Descriptor를 작성하자.

```
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 2.0//EN' 'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>

<!-- Generated XML! -->


<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>AccountManager_EJB</ejb-name>
      <home>com.bea.twopc.AccountManagerHome</home>
      <remote>com.bea.twopc.AccountManager</remote>
      <ejb-class>com.bea.twopc.AccountManager_EJB</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
```

```
  <entity>
    <ejb-name>AccountBankA_EJB</ejb-name>
    <home>com.bea.twopc.AccountBankA_Home</home>
    <remote>com.bea.twopc.AccountBankA</remote>
    <ejb-class>com.bea.twopc.AccountBankA_EJB</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.Long</prim-key-class>
    <reentrant>False</reentrant>
    <cmp-field>
      <field-name>accountId</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>balance</field-name>
    </cmp-field>
    <primkey-field>accountId</primkey-field>
  </entity>

  <entity>
    <ejb-name>AccountBankB_EJB</ejb-name>
    <home>com.bea.twopc.AccountBankB_Home</home>
    <remote>com.bea.twopc.AccountBankB</remote>
    <ejb-class>com.bea.twopc.AccountBankB_EJB</ejb-class>
    <persistence-type>Container</persistence-type>
    <prim-key-class>java.lang.Long</prim-key-class>
    <reentrant>False</reentrant>
    <cmp-field>
      <field-name>accountId</field-name>
    </cmp-field>
    <cmp-field>
      <field-name>balance</field-name>
    </cmp-field>
    <primkey-field>accountId</primkey-field>
  </entity>
</enterprise-beans>

<assembly-descriptor>
  <container-transaction>
    <method>
      <ejb-name>AccountBankA_EJB</ejb-name>
      <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
  <container-transaction>
    <method>
      <ejb-name>AccountBankB_EJB</ejb-name>
      <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
  <container-transaction>
    <method>
      <ejb-name>AccountManager_EJB</ejb-name>
      <method-name>*</method-name>
    <method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
</assembly-descriptor>

</ejb-jar>
```

ejb-jar.xml에서 볼 수 있듯이. AccountManager EJB는 Stateless Session Bean이고, 모든 메소드의 트렌젝션 속성을 Required로 설정했다. (즉, AccountManager에 의해서 호출되는 AccountBankA와 AccountBankB EJB의 ejbStore와 ejbLoad 메소드는 AccountManager와 같은 트렌젝션으로 수행된다.)

Weblogic이나 상용 WAS에 이 EJB를 deploy하기 위해서는 별도의 descriptor가 필요하다. Weblogic의 경우에는 weblogic-ejb-jar.xml과 weblogic-cmp-rdbms-jar.xml이 필요한데, 지면 사정상 이 내용은 생략하기로 한다. 소스를 다운받아서 보면 알 수 있으리라 생각된다.
이 descriptor를 만드는 방법은
http://e-docs.bea.com/wls/docs61/ejb/index.html 를 참고하면 도움이 될 것이다.

여기서 WebLogic에서 EJB Deployment descriptor를 쉽게 만들 수 있는 방법을 잠깐 소개하고자 한다.
EJB가 있는 디렉토리 (여기서는 EJB가 ~/com/javastudy/twopc 구조를 가지고 있으므로 ~/ 디렉토리가 된다.) “ java weblogic.ant.taskdefs.ejb20.DDInit. “(WebLogic 6.1 기준이며, WebLogic 관련 클래스 패스를 설정해놓은 뒤에 실행해야 한다.)을 해 주면 자동으로 EJB Deploy에 필요한 3가지 descriptor 파일을 생성해준다. 물론 DataSource등의 세세한 내용은 개발자가 직접 수정해야 하지만, 그래도 상당 부분의 코딩을 덜어줄 수 있기 때문에, 매우 유용하게 사용할 수 있다.
이렇게 만들어진 deployment descriptor를 META-INF 디렉토리에 저장한다.
~/com/javastudy/twopc에는 9개의 컴파일된 EJB클래스가, ~/META-INF/ 에는 3개의 Deployment Descriptor 파일이 위치해야 한다.

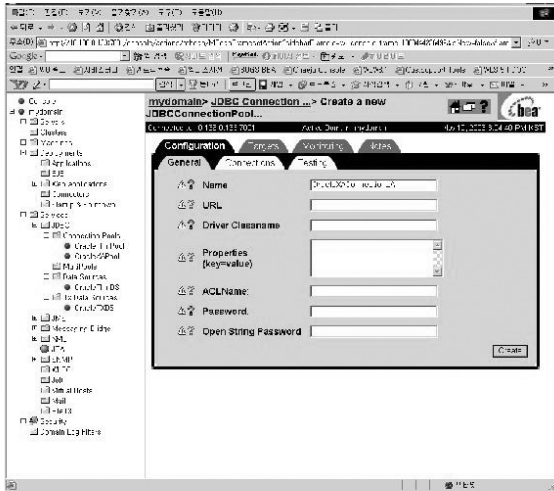
그러면 ~/ 디렉토리에서 하위 디렉토리를 jar 유틸을 이용해서 twopc.jar라는 이름으로 패키징 하자.

```
% jar -cvf twopc.jar ~/*
```

e) 웹로직 설정

EJB는 다 만들었으니까, 이제는 EJB를 Deploy할 EJB Container를 설정하자.
Entity Bean이 있기 때문에 DataSource를 설정해야 한다.

WebLogic Console을 연후, Services 〉 JDBC 〉 Connection Pools 에서 Create New JDBC Connection을 선택한다.



〈그림 3-1〉 웹로직 JDBC Connection Pool 설정

각각의 필드에, 다음과 같이 JDBC 연결 정보를 넣는다.

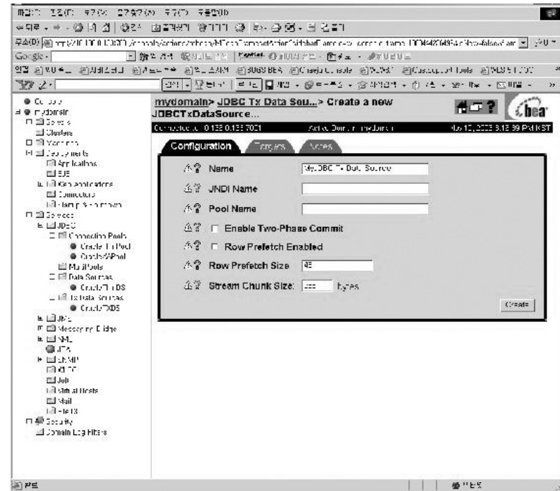
NAME	JDBC Connection Pool이름으로, OracleXAConnection_A와 OracleXAConnection_B를 사용하자.
URL	jdbc:oracle:thin:@IP Address:포트:오라클 SID
Driver ClassName	oracle.jdbc.xa.client.OracleXADataSource
Properties	user=오라클계정 password=오라클비밀번호

※ Driver Class는 XA를 사용하기 때문에, 일반 JDBC Driver 명을 넣는 것이 아니다. 위의 내용을 주목하자.

생성된 JDBC Connection Pool을 Target Tab을 선택하여, 해당 Server에 지정한다.

같은 방법으로 총 두 개의 Connection Pool을 만든다.

만들어진 두 개의 Connection Pool을 이용해서 DataSource를 만들어 보자.
Services>JDBC>TX DataSource 를 선택해서 Configure a new JDBC TxDataSource를 선택한다.



〈그림 3-1〉 웹로직 JDBC TXDataSource설정

각각의 필드를 아래와 같은 내용으로 입력한다.

NAME	OracleXADS_A
JNDI Name	OracleXADS_A
PoolName	위에서 설정한 Connection Pool 이름을 적는다. OracleXAConnection_A

같은 방법으로 OracleXADS_B도 설정한다.

f) EJB DEPLOY

이제 EJB를 Deploy하기 위한 Oracle RDBMS설정과 EJB Container WebLogic의 설정도 완료되었다. 실제로 EJB를 Deploy 해 보자.

웹로직 콘솔에서 Deployments 〉 EJB를 선택하고 Install a new EJB를 선택한다. 인스톨 화면이 나오면, [찾아보기] 버튼을 이용해서, 앞에서 만든 twopc.jar를 선택하고 deploy를 한다. Deploy가 끝난 후에, target 탭을 이용하여, 해당 server에 targetting이 되어 있는지 확인한다.

g) CLIENT 작성

이제 EJB를 성공적으로 Deploy를 했으니, TEST를 해볼 차례

다. TEST를 위해서는 하나의 CLIENT를 만들어야 한다.

```
// TwoPC.java

  import com.bea.twopc.*;
  import javax.naming.*;
  import javax.ejb.*;
  import javax.rmi.*;
  import java.util.*;

  public class TwoPC{
          public static void main(String args[]){
                    try{
                    // step 1. get AccountManager
                    Hashtable env = new Hashtable();
                    env.put(Context.INITIAL_CONTEXT_FACTORY,
  "weblogic.jndi.WLInitialContextFactory");
                    // 밑줄 친부분은 자기 환경에 맞추도록 하자.
                    env.put(Context.PROVIDER_URL,"t3://웹로직 IP:웹로직
  포트");
                    env.put(Context.SECURITY_PRINCIPAL,"system");
                    env.put(Context.SECURITY_CREDENTIALS,"weblogic");

                    Context ctx = new InitialContext(env);
                    System.out.println("ctx :"+ctx);
                    Object obj =
  ctx.lookup("com.bea.twopc.AccountManagerHome");
                    System.out.println
  ("com.bea.twopc.AccountManagerHome Jndi :"+obj);

                    AccountManagerHome home =
  (AccountManagerHome)
                         PortableRemoteObject.narrow
  (obj,AccountManagerHome.class);

                    AccountManager manager = home.create();


                    // step 2. run transfer method
                    String result =
  manager.transfer((long)1,(long)10,(long)100);

                    // step 3. display result
                    System.out.println("result is :"+result);
                    }catch(Exception e){
                              e.printStackTrace();
                    }
          }// main
  }
```

작성된 파일은 WebLogic Class Path를 적용하여, 컴파일 한다.

% javac -d. TwoPC.java

h) CLIENT 실행 및 결과 확인

완성된 Client를 실행해 보자,

% java TwoPC

처음 실행을 하고, DB 내용을 보면 TABLE_A에서 VALUE가 100이 빠져서, TABLE_B에 더해진다. TABLE_B가 들어있는 데이타베이스를 끊거나, 내리고 실행을 하면, 에러가 나면서, TABLE_A에서 100을 뺀 내용을 다시 ROLL BACK하는 것을 확인할 수 있을 것이다..

아주 간단하게, EJB를 이용한 분산 트렌젝션 처리에 대한 프로그래밍을 해봤다. 일일이 집어가면서 하나씩 자세히 설명하면 좋겠지만, 지면 사정상, 내용을 함축해서, EJB에 대한 개념이 없는 개발자에게는 다소 어려울 수 도 있다는 생각은 든다.

정리해보자면 EJB는 우리가 3회에 걸쳐서 살펴본 복잡한 Transaction 처리를 Component 모델이라는 이름 아래서, 매우 쉽고, 안정적으로 처리해 준다. EJB 자체의 개념이 어려우리라 생각하겠지만, 실제로, 이렇게 복잡한 트렌젝션 처리를 간단하고 안정적으로 처리해준다는건 많은 메리트가 있는 것이다. 거기다가 대부분의 EJB Container들은, 이런 EJB의 클러스터링 기능까지 지원하고 있기 때문에, 개발자가 개발해야 하는 많은 복잡한 부분을 대체해주고 있다..

EJB에서 제시하는 6가지 트렌젝션 모델은 대부분의 트렌젝션이 필요한 프로그래밍을 가능하게 해 준다. 일반적인 개발에서는 트렌젝션 관리를 Container에 맡기고, 개발자는 Logic에만 신경을 쓰는 것이 바람직하며, 복잡한 트렌젝션이나, 트렌젝션의 전이 과정, Bean Managed Transaction을 사용하는 경우에는 좀더 주의를 기울이도록 하자.