Tibero

애플리케이션 개발자 안내서

Tibero 7

대한민국 경기도 성남시 분당구 황새울로258번길 29, BS 타워 9층 우)13595

Website

http://www.tmaxtibero.com

기술서비스센터

Tel : +82-1544-8629

E-Mail : info@tmax.co.kr

Open Source Software Notice

Some modules or files of this product are subject to the terms of the following licenses. : OpenSSL, RSA Data Security, Inc., Apache Foundation, Jean-loup Gailly and Mark Adler, Paul Hsieh's hash

Detailed Information related to the license can be found in the following directory : ${INSTALL_PATH}/license/oss_licenses

본 제품의 일부 파일 또는 모듈은 다음의 라이선스를 준수합니다. : OpenSSL, RSA Data Security, Inc., Apache Foundation, Jean-loup Gailly and Mark Adler, Paul Hsieh's hash

관련 상세한 정보는 제품의 다음의 디렉터리에 기재된 사항을 참고해 주십시오. : ${INSTALL_PATH}/license/oss_licenses

안내서 정보

안내서 제목: Tibero 애플리케이션 개발자 안내서

발행일: 2024-08-22

소프트웨어 버전: Tibero 7.2.2

안내서 버전: v7.2.2

내용 목차

그림 목차

예 목차

안내서에 대하여

안내서의 대상

본 안내서는 Tibero®(이하 Tibero)에서 제공하는 각종 애플리케이션 라이브러리를 이용하여 프로그램을 개발하려는 애플리케이션 프로그램 개발자를 대상으로 기술한다.

안내서의 전제 조건

본 안내서를 원활히 이해하기 위해서는 다음과 같은 사항을 미리 알고 있어야 한다.

- 데이터베이스의 이해
- RDBMS의 이해
- Java 프로그래밍의 이해
- tbPSM 언어 및 패키지 사용의 이해

안내서의 제한 조건

본 안내서는 Tibero를 실무에 적용하거나 운용하는 데 필요한 모든 사항을 포함하고 있지 않다. 따라서 설치, 환경설정 등 운용 및 관리에 대해서는 각 제품 안내서를 참고하기 바란다.

참고

Tibero의 설치 및 환경설정에 관한 내용은 "Tibero 설치 안내서"를 참고한다.

안내서 구성

Tibero 애플리케이션 개발자 안내서는 총 7개의 장과 Appendix로 이루어져 있다.

각 장의 주요 내용은 다음과 같다.

- 제1장: 데이터 타입의 사용

  Tibero에서 제공하는 데이터 타입과 이를 사용하는 방법을 소개한다.

- 제2장: tbJDBC의 사용

  tbJDBC를 사용하는 방법을 기술한다.

- 제3장: 트리거의 사용

  트리거를 사용하는 방법을 기술한다.

- 제4장: XA의 사용

  분산 트랜잭션 처리에 필요한 XA를 사용하는 방법을 기술한다.

- 제5장: mod_tbPSM의 사용

  Apache HTTP 웹 서버를 통하여 tbPSM 프러시저 호출 및 HTML 페이지를 가져올 수 있는 mod_tbpsm를 사용하는 방법을 기술한다.

- 제6장: 객체 타입의 사용

  Tibero의 객체 타입과 구성요소 동작방식에 대해서 설명한다.

- 제7장: 컬렉션 타입의 사용

  Tibero의 컬렉션 타입과 사용 방법에 대해서 설명한다.

- Appendix A: tbJDBC 예제

  tbJDBC를 이용하여 작성한 기본 프로그램의 전체 소스 코드를 기술한다.

- Appendix B: Tibero와 Tuxedo 연동 예제

  Tibero와 Tuxedo 연동 예제 프로그램의 전체 소스 코드와 각종 스크립트를 기술한다.

안내서 규약

표기	의미
<<AaBbCc123>>	프로그램 소스 코드의 파일명
<Ctrl>+C	Ctrl과 C를 동시에 누름
[Button]	GUI의 버튼 또는 메뉴 이름
진하게	강조
" "(따옴표)	다른 관련 안내서 또는 안내서 내의 다른 장 및 절 언급
'입력항목'	화면 UI에서 입력 항목에 대한 설명
하이퍼링크	메일 계정, 웹 사이트
>	메뉴의 진행 순서
+----	하위 디렉터리 또는 파일 있음
|----	하위 디렉터리 또는 파일 없음
참고	참고 또는 주의사항
주의	주의할 사항
[그림 1.1]	그림 이름
[예 1.1]	예제 이름
`AaBbCc123`	Java 코드, XML 문서
[*command argument*]	옵션 파라미터
< xyz >	'<'와 '>' 사이의 내용이 실제 값으로 변경됨
|	선택 사항. 예) A|B: A나 B 중 하나
…	파라미터 등이 반복되어서 나옴
${ }	환경변수

시스템 사용 환경

	요구 사항
Platform	HP-UX 11i v3(11.31)
	Solaris (Solaris 11)
	AIX (AIX 7.1/AIX 7.2/AIX 7.3)
	GNU (X86, 64, IA64)
	Red Hat Enterprise Linux 7 kernel 3.10.0 이상
	Windows(x86) 64bit
Hardware	최소 2.5GB 하드디스크 공간
	1GB 이상 메모리 공간
Compiler	PSM (C99 지원 필요)
	tbESQL/C (C99 지원 필요)

관련 안내서

안내서	설명
Tibero 설치 안내서	설치 시 필요한 시스템 요구사항과 설치 및 제거 방법을 기술한 안내서이다.
Tibero tbCLI 안내서	Call Level Interface인 tbCLI의 개념과 구성요소, 프로그램 구조를 소개하고 tbCLI 프로그램을 작성하는 데 필요한 데이터 타입, 함수, 에러 메시지를 기술한 안내서이다.
Tibero External Procedure 안내서	External Procedure를 소개하고 이를 생성하고 사용하는 방법을 기술한 안내서이다.
Tibero JDBC 개발자 안내서	Tibero에서 제공하는 JDBC 기능을 이용하여 애플리케이션 프로그램을 개발하는 방법을 기술한 안내서이다.
Tibero tbESQL/C 안내서	C 프로그래밍 언어를 사용해 데이터베이스 작업을 수행하는 각종 애플리케이션 프로그램을 작성하는 방법을 기술한 안내서이다.
Tibero tbESQL/COBOL 안내서	COBOL 프로그래밍 언어를 사용해 데이터베이스 작업을 수행하는 각종 애플리케이션 프로그램을 작성하는 방법을 기술한 안내서이다.
Tibero tbPSM 안내서	저장 프러시저 모듈인 tbPSM의 개념과 문법, 구성요소를 소개하고, 프로그램을 작성하는 데 필요한 제어 구조, 복합 타입, 서브 프로그램, 패키지와 SQL 문장을 실행하고 에러를 처리하는 방법을 기술한 안내서이다.
Tibero tbPSM 참조 안내서	저장 프러시저 모듈인 tbPSM의 패키지를 소개하고, 이러한 패키지에 포함된 각 프러시저와 함수의 프로토타입, 파라미터, 예제 등을 기술한 참조 안내서이다.
Tibero 관리자 안내서	Tibero의 동작과 주요 기능의 원활한 수행을 보장하기 위해 DBA가 알아야 할 관리 방법을 논리적 또는 물리적 측면에서 설명하고, 관리를 지원하는 각종 도구를 기술한 안내서이다.
Tibero 유틸리티 안내서	데이터베이스와 관련된 작업을 수행하기 위해 필요한 유틸리티의 설치 및 환경설정, 사용 방법을 기술한 안내서이다.
Tibero TAS 안내서	Tibero Active Cluster (TAS)를 사용해서 Tibero의 파일을 관리하고자 하는 관리자를 대상으로 기술한 안내서이다.
Tibero	Tibero를 사용하는 도중에 발생할 수 있는 각종 에러의 원인과 해결 방법을 기술한 안내서이다.

안내서	설명
에러 참조 안내서	
Tibero 참조 안내서	Tibero의 동작과 사용에 필요한 초기화 파라미터와 데이터 사전, 정적 뷰, 동적 뷰를 기술한 참조 안내서이다.
Tibero SQL 참조 안내서	데이터베이스 작업을 수행하거나 애플리케이션 프로그램을 작성할 때 필요한 SQL 문장을 기술한 참조 안내서이다.
Tibero Spatial 참조 안내서	Tibero에서 Geometry 타입에 대한 설명과 Spatial 기능 관련 프러시저 함수 목록 및 사용 방법 등을 기술한 안내서이다.
Tibero TEXT 참조 안내서	Tibero의 제공하는 Text Index를 소개하고, Text Index를 생성 하고 사용하는 방법을 기술하는 안내서이다.
Tibero TDP.NET 안내서	Tibero Data Provider for .NET 기능을 기술하는 안내서이다.
Tibero IMCS 안내서	Tibero에서 제공하는 In-Memory Column Store(이하 IMCS) 기능을 기술하는 안내서이다.

제1장 데이터 타입의 사용

본 장에서는 각종 애플리케이션 라이브러리를 사용하기 앞서 Tibero가 기본적으로 제공하는 데이터 타입을 소개하고 사용하는 방법을 설명한다.

1.1. 개요

데이터 타입은 데이터베이스에 스키마 객체를 생성할 때 필요하며, 일반적인 클라이언트 프로그램에서도 모든 데이터 타입에 대응되는 변수를 사용할 수 있다.

Tibero에서 제공하는 데이터 타입은 다음과 같다.

구분	데이터 타입	설명
문자형	CHAR, VARCHAR, VARCHAR2, NCHAR, NVARCHAR, NVARCHAR2, RAW, LONG, LONG RAW	문자열을 표현하는 데이터 타입이다.
숫자형	NUMBER, BINARY_FLOAT, BINARY_DOUBLE	정수나 실수의 숫자를 저장하는 데이터 타입이다.
날짜형	DATE, TIME, TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE	시간이나 날짜를 저장하는 데이터 타입이다.
간격형	INTERVAL YEAR TO MONTH, INTERVAL DAY TO SECOND	시간 간격을 저장하는 데이터 타입이다.
대용량객체형	CLOB, NCLOB, BLOB, XMLTYPE, JSON	대용량의 객체를 저장하는 데이터 타입이다.
내재형	ROWID	사용자가 명시적으로 선언하지 않아도 Tibero가 자동으로 삽입되는 로우마다 포함하는 컬럼의 타입이다.

참고

데이터 타입에 대한 자세한 내용은 "Tibero SQL 참조 안내서"를 참고한다.

1.2. 문자형

본 절에서는 문자열을 표현하는 데이터 타입에 대해서 설명한다.

CHAR

CHAR 타입은 일반 문자열을 저장하는 데이터 타입으로, 고정 길이 문자열이다.

사용 방법은 다음과 같다.

```
CHAR[(size[BYTE|CHAR])]
```

옵션	설명
size	size는 선택적으로 사용할 수 있다. (최댓값: 2,000bytes) – size를 입력한 경우 : size만큼의 고정 길이를 갖는다. – size를 입력하지 않은 경우 : 기본값인 1byte의 고정 길이를 갖는다.

VARCHAR, VARCHAR2

VARCHAR 또는 VARCHAR2 타입은 일반 문자열을 저장하는 데이터 타입으로, 가변 길이 문자열이다.

사용 방법은 다음과 같다.

```
VARCHAR(size[BYTE|CHAR])
VARCHAR2(size[BYTE|CHAR])
```

항목	설명
size	size에 정의된 만큼의 길이를 갖는다. (최댓값: 65,532bytes)

NCHAR

NCHAR 타입은 다국어 문자집합을 사용하는 문자열을 저장하는 데이터 타입으로, 고정 길이 문자열이다.

사용 방법은 다음과 같다.

```
NCHAR(size)
```

항목	설명
size	size에 정의된 만큼의 고정 길이를 갖는다. (최댓값: 2,000bytes)

NVARCHAR, NVARCHAR2

NVARCHAR 또는 NVARCHAR2 타입은 다국어 문자집합을 사용하는 문자열을 저장하는 데이터 타입으로, 가변 길이 문자열이다.

사용 방법은 다음과 같다.

```
NVARCHAR(size)
```

항목	설명
size	size에 정의된 만큼의 길이를 갖는다. (최댓값: 65,532bytes)

LONG

LONG 타입은 대용량 데이터를 저장하기 위한 데이터 타입이다. 단, Oracle과의 호환성을 위해서만 사용한다. 그 외에는 CLOB 타입을 사용할 것을 권장한다.

사용 방법은 다음과 같다.

```
LONG
```

RAW

RAW 타입은 가변 길이를 갖는 바이너리 데이터를 저장하는 데이터 타입이다.

사용 방법은 다음과 같다.

```
RAW(size)
```

항목	설명
size	size에 정의된 만큼의 길이를 갖는다. (최댓값: 2,000bytes)

LONG RAW

LONG RAW 타입은 대용량의 바이너리 데이터를 저장하는 데이터 타입이다. 단, Oracle과의 호환성을 위해서만 사용한다. 그 외에는 BLOB 타입을 사용할 것을 권장한다.

사용 방법은 다음과 같다.

```
LONG RAW
```

1.3. 숫자형

본 절에는 정수나 실수의 숫자를 저장하는 데이터 타입에 대해서 설명한다.

NUMBER

NUMBER 타입은 정수 또는 실수를 저장하는 데이터 타입이다. 실제로 데이터베이스에 저장될 때는 숫자의 크기에 따라 가변 길이로 저장된다.

사용 방법은 다음과 같다.

```
NUMBER[(precision[, scale])]
```

옵션	설명
precision	정밀도이며, 유효숫자의 최대 자릿수를 설정한다.
scale	스케일이며, 소수점 아래 가장 오른쪽 유효숫자까지의 자릿수를 설정한다.

BINARY_FLOAT

BINARY_FLOAT 타입은 실수나 정수를 표현하고, 32비트로 저장하는 단일 정밀도 데이터 타입이다.

BINARY_DOUBLE

BINARY_DOUBLE 타입은 실수나 정수를 표현하고, 64비트로 저장하는 2배 정밀도 데이터 타입이다.

1.4. 날짜형

본 절에서는 시간이나 날짜를 저장하는 데이터 타입에 대해서 설명한다.

DATE

DATE 타입은 년, 월, 일의 날짜를 저장하는 데이터 타입이다. 실제로 데이터베이스에 저장될 때는 고정 길이로 저장된다.

사용 방법은 다음과 같다.

```
DATE
```

TIME

TIME 타입은 시, 분, 초의 시간을 저장하는 데이터 타입이다. 실제로 데이터베이스에 저장될 때는 고정 길이로 저장된다.

사용 방법은 다음과 같다.

```
TIME
```

TIMESTAMP

TIMESTAMP 타입은 DATE 타입을 확장한 것으로 시간까지 저장하는 데이터 타입이다. 실제로 데이터베이스에 저장될 때는 고정 길이로 저장된다.

사용 방법은 다음과 같다.

```
TIMESTAMP[(fractional_seconds_precision)]
```

옵션	설명
fractional_seconds_precision	소수점 초 단위 정밀도를 설정한다. – 0부터 9까지 사용할 수 있다. (기본값: 6)

TIMESTAMP WITH TIME ZONE

TIMESTAMP WITH TIME ZONE 타입은 TIMESTAMP 타입을 확장한 것으로 주어진 시간을 UTC 시간대로 변경하고, 시간대 정보까지 저장하는 데이터 타입이다. 실제로 데이터베이스에 저장될 때는 고정 길이로 저장된다.

사용 방법은 다음과 같다.

```
TIMESTAMP[(fractional_seconds_precision)] WITH TIME ZONE
```

옵션	설명
fractional_seconds_precision	소수점 초 단위 정밀도를 설정한다. – 0부터 9까지 사용할 수 있다. (기본값: 6)

TIMESTAMP WITH LOCAL TIME ZONE

TIMESTAMP WITH LOCAL TIME ZONE 타입은 TIMESTAMP WITH TIME ZONE 타입과 마찬가지로 UTC 시간대로 바꾸기는 하지만, 시간대 정보는 저장하지 않는 데이터 타입이다. 실제로 데이터베이스에 저장될 때는 고정 길이로 저장되고, TIMESTAMP 타입과 같은 길이를 가진다.

사용 방법은 다음과 같다.

```
TIMESTAMP[(fractional_seconds_precision)] WITH LOCAL TIME ZONE
```

옵션	설명
fractional_seconds_precision	소수점 초 단위 정밀도를 설정한다. – 0부터 9까지 사용할 수 있다. (기본값: 6)

1.5. 간격형

본 절에서는 시간 간격을 저장하는 데이터 타입인 간격형에 대해서 설명한다.

INTERVAL YEAR TO MONTH

INTERVAL YEAR TO MONTH 타입은 년과 월의 차이를 저장하는 데이터 타입이다. 실제로 데이터베이스에 저장될 때는 고정 길이로 저장된다.

사용 방법은 다음과 같다.

```
INTERVAL YEAR [(year_precision)] TO MONTH
```

옵션	설명
year_precision	연단위 정밀도를 설정한다. – 0부터 9까지 사용할 수 있다. (기본값: 2) – 연도의 자릿수를 제한할 수 있다.

INTERVAL DAY TO SECOND

INTERVAL DAY TO SECOND 타입은 날짜와 시간의 차이를 저장하는 데이터 타입이다. 실제로 데이터베이스에 저장될 때는 고정 길이로 저장된다.

사용 방법은 다음과 같다.

```
INTERVAL DAY [(day_precision)] TO SECOND [(fractional_seconds_precision)]
```

옵션	설명
day_precision	일단위 정밀도를 설정한다. – 0부터 9까지 사용할 수 있다. (기본값: 2)
fractional_seconds_precision	TIMESTMAP 타입에 정의된 옵션과 같다.

참고

이 타입은 day_precision 옵션과 fractional_seconds_precision 옵션을 사용하여 날짜의 자릿수, 초 이하의 자릿수를 제한할 수 있다.

1.6. 대용량 객체형

본 절에서는 대용량 객체를 저장하는 데이터 타입에 대해서 설명한다.

CLOB

CLOB 타입은 읽을 수 있는 문자열을 데이터베이스에 저장하는 데이터 타입으로, 대용량 데이터를 저장한다. 대용량 데이터는 4GB까지 저장할 수 있다.

사용 방법은 다음과 같다.

```
CLOB
```

NCLOB

NCLOB 타입은 읽을 수 있는 National Characterset 문자열을 데이터베이스에 저장하는 데이터 타입으로, 대용량 데이터를 저장한다. 대용량 데이터는 4GB까지 저장할 수 있다.

사용 방법은 다음과 같다.

```
NCLOB
```

BLOB

BLOB 타입은 대용량의 바이너리 데이터를 저장하는 데이터 타입이다.

사용 방법은 다음과 같다.

```
BLOB
```

대용량의 바이너리 데이터는 4GB까지 저장할 수 있다.

XMLTYPE

XML(Extensible Markup Language)은 구조화되거나 그렇지 않은 모든 데이터를 표현하기 위해 W3C(World Wide Web Consortium)에 의해 표준으로 제정된 형식이다. Tibero에서는 XML 데이터를 저장하기 위해 XMLTYPE 타입을 제공하며, 내부적으로 CLOB 형식으로 저장된다.

사용 방법은 다음과 같다.

```
XMLTYPE
```

JSON

JSON은 KEY/VALUE 쌍으로 이루어진 데이터 오브젝트를 전달하기 위해 사용하는 개방형 표준 포맷이다. Tibero에서는 JSON 데이터를 저장하기 위해 JSON 타입을 제공하며, 내부적으로 BLOB 형식으로 저장된다.

사용 방법은 다음과 같다.

```
JSON
```

1.7. 내재형

본 절에서는 사용자가 명시적으로 선언하지 않아도 Tibero가 자동으로 삽입되는 로우마다 포함하는 컬럼의 타입인 내재형에 대해서 설명한다.

ROWID

ROWID 타입은 데이터베이스 테이블의 컬럼 주소를 저장하는 데이터 타입이다. 실제로 데이터베이스에 저장될 때는 고정 길이로 저장된다.

사용 방법은 다음과 같다.

```
ROWID
```

제2장 tbJDBC의 사용

본 절에서는 tbJDBC의 사용법과 JDBC의 기본 기능, 프로그래밍에 대해서 기술한다.

2.1. 개요

Tibero에서는 Java 프로그램 안에서 SQL 문장을 실행하기 위해 데이터베이스를 연결해주는 애플리케이션 프로그램의 인터페이스를 제공한다. 이러한 인터페이스를 **tbJDBC**(Tibero의 Java Database Connectivity)라 한다.

tbJDBC는 JDK 버전이 1.4 이상의 환경에서 동작한다. JDBC 4.0을 기능을 모두 사용하기 위해서는 JDK 6 이상의 환경을 사용하여야 한다. JDK 6 이상에서 사용할 수 있는 드라이버 파일은 **tibero7-jdbc.jar** 파일의 형태로 제공되며, JDK 버전이 1.4인 경우에는 **tibero7-jdbc-14.jar** 파일의 형태로 제공된다.

참고

본 안내서에서는 JDBC 프로그래밍에 대해 간략히 설명한다. tbJDBC에 대한 자세한 내용은 "Tibero JDBC 개발자 안내서"를 참고한다.

2.2. JDK 설치

tbJDBC를 사용하기 위해서는 **JDK 1.4** 이상이 반드시 설치되어 있어야 한다.

다음 위치에서 JDK를 다운로드할 수 있다.

```
http://www.oracle.com/technetwork/java/javase/downloads/index.html
```

만약 시스템이 Oracle사의 JDK를 사용하지 않는다면 각각의 시스템에 적합한 JDK를 찾아 설치한다. 예를 들어 HP-UX는 HP, AIX는 IBM에서 JDK를 다운로드 받아 설치하면 된다.

각 시스템별 JDK를 설치하는 방법은 다음 위치에서 확인할 수 있다.

```
http://www.oracle.com/technetwork/java/index.html
```

참고

벤더별 JDK 설치 방법은 각 벤더에서 제공하는 설치 안내서를 참고한다.

2.3. JDBC의 표준 기능

tbJDBC는 JDBC 4.0 표준을 준수한다. 다만, 표준에 따르면 필수적으로 구현해야하는 부분과 벤더별로 구현여부를 선택 가능한 부분을 구분하여 명시하고 있으며, 이러한 추가 기능에 대해서는 지원하지 않을 수도 있다. 또한 JDBC 4.0은 아래의 JDBC 표준을 모두 포함한다.

- JDBC 3.0 API
- JDBC 2.0 API(JDBC 2.0 Optional Package API and JDBC 2.1 core API)
- JDBC 1.2 API
- JDBC 1.0 API

본 절에서는 tbJDBC에서 구현되는 표준 기능을 JDBC 버전 별로 간략히 설명한다. 보다 자세한 내용은 Sun사에서 발행한 JDBC 4.0 Specification 문서를 참고한다.

2.3.1. JDBC 1.0 및 JDBC 2.0

본 절에서는 JDBC 1.0 및 JDBC 2.0의 표준 기능을 지원 여부에 따라 각각 설명한다.

지원하는 기능

- SQL-99 데이터 타입

  tbJDBC는 SQL-99에서 새롭게 추가된 데이터 타입 중에 BLOB와 CLOB만을 지원한다. 따라서 이러한 데이터 타입을 Tibero에서 사용할 때는 다음과 같은 규칙을 따른다.

  – Blob, Clob 인터페이스는 해당 트랜잭션이 수행 중인 동안만 유효하다.

  – Blob, Clob 인터페이스는 위치 지시자(LOCATOR)를 통해 구현된다. 위치 지시자는 데이터베이스 서버에 존재하는 데이터의 논리적인 지시자(POINTER)이다.

- 인터페이스 메소드

  지원하는 인터페이스 메소드(Interface Method)는 다음과 같다.

  – java.sql.Array

  – java.sql.Blob

  – java.sql.CallableStatement

  – java.sql.Clob

  – java.sql.Connection

  – java.sql.DatabaseMetaData

  – java.sql.Driver

- java.sql.PreparedStatement
- java.sql.ResultSet
- java.sql.ResultSetMetaData
- java.sql.Statement
- java.sql.Struct

- 정적 초기화를 구현한 드라이버

  java.sql.Driver 인터페이스 메소드를 구현한 드라이버 클래스는 정적 초기화 블록(Static block)에서 Drivermanager.registerDriver 메소드를 호출함으로써, 새로 생성한 드라이버 인스턴스를 드라이버 관리자(Driver Manager)에 등록할 수 있다.

- SQL-92 Entry Level의 확장 기능

  DROP TABLE과 ESCAPE 문을 사용할 수 있다.

- 스칼라 함수(Scalar Functions)

  데이터베이스 서버에서 지원하는 모든 스칼라 함수를 사용할 수 있다. 스칼라 함수는 **DatabaseMetaData** 클래스를 통해 확인할 수 있다.

- 멀티스레딩(Multithreading)

  여러 개의 스레드가 java.sql과 javax.sql 패키지 내에 존재하는 객체에 모든 연산을 동시에 수행할 수 있다.

- 스크롤 가능한 결과 집합(Scrollable ResultSet)

  결과 집합을 전 방향 또는 후 방향으로 탐색할 수 있다. 또한 결과 집합 내에서 상대적이거나 절대적인 위치 이동도 할 수 있다.

- 수정 가능한 결과 집합(Updatable ResultSet)

  결과 집합을 수정할 수 있다.

- 위치 설정을 통한 수정 및 삭제(Positioned Updates and Deletes)

  결과 집합의 커서에서 설정한 현재 위치의 레코드를 수정하거나 삭제하는 기능이다.

- 민감한 결과 집합(Sensitive ResultSet)

  결과 집합의 커서가 열린 후에 발생하는 데이터의 변화를 반영할 수 있다.

- 일괄 수정 작업(Batch Updates)

  여러 번의 수정 작업을 한꺼번에 처리할 수 있다.

- RowSet 기술

  데이터를 편리하게 전달하고 처리할 수 있다.

- 분산 트랜잭션(Distrubuted Transactions)

  여러 데이터베이스 서버에 이르는 트랜잭션을 가능하게 한다.

지원하지 않는 기능

- SQL-99 데이터 타입

  tbJDBC는 SQL-99 데이터 타입 중에서 Array, Ref, Struct를 지원하지 않는다.

- 인터페이스 메소드

  지원하지 않는 인터페이스 메소드는 다음과 같다.

  – java.sql.Ref

  – java.sql.SQLData

  – java.sql.SQLInput

  – java.sql.SQLOutput

- 잘린(Truncated) 데이터 대한 예외 상황

  JDBC 드라이버가 예기치 않게 데이터 값을 잘라낸 경우 읽기를 할 때에는 Data Truncation 경고, 쓰기를 할 때에는 Data Truncation 예외(Exception)를 발생시킨다.

2.3.2. JDBC 3.0

본 절에서는 JDBC 3.0의 표준 기능을 지원 여부에 따라 각각 설명한다. JDBC 3.0 API는 J2SE platform, version 1.4 이상의 환경에서 사용할 수 있다.

지원하는 기능

- 인터페이스

  지원하는 인터페이스는 다음과 같다.

  – java.sql.ParameterMetaData

  – java.sql.Savepoints

  – javax.sql.ConnectionEventListener

  – javax.sql.ConnectionPoolDataSource

  – javax.sql.DataSource

  – javax.sql.PooledConnection

  - javax.sql.RowSet
  - javax.sql.RowSetInternal
  - javax.sql.RowSetListener
  - javax.sql.RowSetMetaData
  - javax.sql.RowSetReader
  - javax.sql.RowSetWriter
  - javax.sql.XAConnection
  - javax.sql.XADataSource

- 접속 풀링(Connection Pooling)의 설정

  최대 풀 크기, 최소 풀 크기, 초기 풀 크기 등과 같이 접속 풀링에 사용되는 다양한 파라미터를 설정할 수 있는 API를 제공한다.

- 문장의 풀링(Statement Pooling)

  접속 풀링과 관련된 문장의 풀링을 제공한다.

- 파라미터 메타데이터

  ParameterMetaData 인터페이스를 구현한 JDBC 클래스는 준비된 문장(Prepared Statement)에서 사용한 파라미터 개수의 정보를 제공한다. 단, 데이터 타입 및 속성(Property)에 대한 메타데이터(metadata)는 제공하지 않는다.

- 저장점

  Savepoint 인터페이스를 구현하여 트랜잭션에 대한 저장점 설정과 커밋 및 롤백 기능을 제공한다. 단, 특정한 저장점을 해제하는 **Connection.releaseSavepoint** java.sql 메소드는 제공하지 않는다.

- 자동 생성 키

  SQL 문장을 실행한 후 결과 집합으로부터 `getGeneratedKeys` 함수를 사용하여 결과 로우에 대한 키 또는 사동으로 생성된 컬럼의 값을 얻는다.

- 결과 집합의 유지성(ResultSet Holdability)

  결과 집합이 열려있는 동안에 커밋이 발생했을 때 결과 집합을 그대로 유지할지 아니면 닫을지를 설정한다. tbJDBC에서는 결과 집합을 유지하는 방법만을 지원한다.

- 여러 개의 결과 집합 반환

  한 SQL 문장이 여러 개의 결과 집합을 열 수 있도록 지원한다.

- BLOB와 CLOB 객체에 존재하는 데이터의 수정

  updateXXX API를 통해 BLOB와 CLOB 객체에 포함된 데이터를 수정하는 기능을 제공한다.

지원하지 않는 기능

- 추가된 데이터 타입

타입	설명
java.sql.Types.DATALINK	URL과 같은 외부 리소스의 접근을 제공한다.
java.sql.Types.BOOLEAN	BIT 타입과 논리적으로 동일하다.

- REF 객체에 의해 참조된 객체의 검색 및 수정

  REF 객체에 의해 참조된 객체를 검색하고, 수정할 수 있는 기능을 제공한다.

- 그룹 변환 및 데이터 타입의 매핑

  JDBC API를 통해 사용자 정의 데이터 타입(UDT: User Defined Types)과 Java 클래스 간의 매핑 관계를 설정할 수 있다. 그러나 Tibero에서는 사용자 정의 데이터 타입을 지원하지 않는다.

2.3.3. JDBC 4.0

본 절에서는 JDBC 4.0의 표준 기능을 지원 여부에 따라 각각 설명한다. JDBC 4.0 API는 Java SE platform, version 6 이상의 환경에서 사용할 수 있다.

지원하는 기능

- 인터페이스

  지원하는 인터페이스는 다음과 같다.

  – java.sql.NClob

  – java.sql.RowId

  – java.sql.SQLXML

  – java.sql.Wrapper

  – javax.sql.StatementEventListener

- SQL : 2003에 추가된 XML 데이터 타입 지원

  SQLXML 인터페이스를 통해 XML 데이터 타입을 사용할 수 있다.

- 자동 java.sql.Driver 검출(Auto java.sql.Driver discovery)

  Class.forName을 사용한 java.sql.Driver 클래스의 로드 없이도 드라이버 객체를 사용할 수 있다.

- 국가별 캐릭터 셋(National Character Set) 지원

  데이터베이스에서 별도로 지정해 사용하는 국가별 캐릭터 셋을 지원하기 위한 API가 추가되었다.

- 향상된 SQLException

  연쇄적으로 연결된 예외를 생성할 수 있게 되어 보다 자세한 원인을 전달해 줄 수 있으며, 새로운 종류의 예외 타입이 추가되었다.

- 향상된 Blob/Clob 기능

  Blob/Clob 객체를 생성/해제할 수 있는 API를 지원한다.

- SQL ROWID 데이터 타입의 지원

  RowId 인터페이스를 이용하여 SQL ROWID 타입을 사용할 수 있다.

- 실제 JDBC 객체에 대한 접근 허용

  Wrapper 인터페이스를 이용하여 애플리케이션 서버나 접속 풀링 환경에서도 실제 JDBC 객체에 접근해 사용할 수 있다.

- 접속 풀링 환경에서 실제 접속 상태에 대한 통지

  접속 풀링 환경에서 실제 접속이 닫히거나 유효하지 않게 되었을 때 그 상태를 풀링된 문장에 통지해 준다.

지원하지 않는 기능

- 사용자 정의 타입 조회 및 계층 구조 검색

  사용자 정의 타입의 속성 및 계층 구조를 조회할 수 있는 API가 추가되었다.

2.4. 기본 프로그래밍

본 절에서는 tbJDBC에서 제공하는 인터페이스 메소드를 통해 기본적인 Java 프로그램을 작성하는 방법을 설명한다. 본 절에서 설명하고 있는 전체 소스 코드는 “Appendix A. tbJDBC 예제”를 참고한다.

다음은 public class가 포함된 JdbcTest 클래스 파일의 일부이다.

[예 2.1] tbJDBC를 사용한 기본 프로그래밍

```
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;
```

```
public class JdbcTest
{
    Connection conn;

    public static void main(String[] args) throws Exception
   {
        JdbcTest test = new JdbcTest();

        test.connect();                      ... ⓐ ...
        test.executeStatement();             ... ⓑ ...
        test.executePreparedStatement();     ... ⓒ ...
        test.executeCallableStatement();     ... ⓓ ...
        test.disconnect();                   ... ⓔ ...
    }

    /* ... 기능별 멤버 함수를 구현한다. */
}
```

ⓐ JdbcTest라는 클래스를 통해 데이터베이스에 접속한다.

ⓑ ~ ⓓ 여러 종류의 멤버 함수를 수행한다.

ⓔ 데이터베이스 접속을 해제하는 작업을 수행한다.

다음 절부터는 위 예에서 ⓐ ~ ⓔ 사이에 구현된 멤버 함수를 각각 설명한다.

2.4.1. 접속

connect 멤버 함수는 기본 드라이버 관리자를 이용하여 데이터베이스에 **접속(connection)**하는 동작(ⓐ)을 수행한다. Java에서 기본 제공하는 **java.sql.DriverManager** 클래스를 통해 드라이버 관리자 기능을 사용할 수 있다.

[예 2.2] tbJDBC - 접속

```
private void connect() throws ClassNotFoundException
{
    Class.forName("com.tmax.tibero.jdbc.TbDriver");

    try {
        conn = DriverManager.getConnection(
                "jdbc:tibero:thin:@localhost:8629:tibero",
                "tibero", "tmax");
    } catch (SQLException e) {
        System.out.println("connection failure!");
        System.exit(-1);
    }
```

```
    System.out.println("Connection success!");
}
```

구현된 내용을 차례대로 설명하면 다음과 같다.

1. 사용할 드라이버에 해당하는 클래스를 지정하여 Class.forName 메소드를 호출한다.

   아래에서 드라이버 관리자를 사용하기 전에 사용할 드라이버를 미리 등록하는 과정이다. 사용할 드라이버에 해당하는 클래스의 이름을 지정하여, Java 가상머신이 드라이버 클래스를 로드하도록 한다.

2. DriverManager.getConnection 메소드를 호출하여 드라이버 관리자를 통해 데이터베이스 연결을 생성한다. 사용자는 DriverManager.getConnection 메소드의 파라미터로 접속할 데이터베이스의 URL과 DB 사용자이름 및 비밀번호를 입력한다. DriverManager 클래스는 URL에 맞는 드라이버를 찾아 접속에 성공할 경우 데이터베이스 연결을 java.sql.Connection 인터페이스의 객체로 돌려준다. 이를 필드 변수 conn에 할당한다.

3. 드라이버 관리자가 URL에 적합한 드라이버를 찾지 못하거나, 드라이버에서 데이터베이스로의 접속에 실패한 경우에는 SQLException이 throw된다.

2.4.2. 실행

tbJDBC에서는 다른 데이터베이스의 클라이언트에서처럼 **문장(statement)**에 대한 개념이 존재한다. **executeStatement 멤버 함수**는 이러한 문장을 실행(execution)하는 기능(ⓑ)을 수행한다.

[예 2.3] tbJDBC - 실행

```
private void executeStatement() throws SQLException
{
    String dropTable   = "drop table emp";
    String createTable = "create table emp (id number, "+
                         " name varchar(20), salary number)";
    String InsertTable = "insert into emp values(1000, 'Park', 5000)";

    Statement stmt = conn.createStatement();

    try {
        stmt.executeUpdate(dropTable);
    } catch(SQLException e) {
        // if there is not the table
    }

    stmt.executeUpdate(createTable);
    stmt.executeUpdate(InsertTable);

    stmt.close();
}
```

구현된 내용을 차례대로 설명하면 다음과 같다.

1. dropTable, createTable, InsertTable를 각각 String 객체 참조형 변수로 만든다. 객체 참조형 변수에 각각 특정 명령을 실행할 SQL 문장을 만든다.
2. 연결에서 `createStatement` 메소드를 호출하여 새로운 문장을 생성한다. java.sql.Statement 객체로 반환되며, 이를 이용해 SQL 문장을 실행할 수 있다.
3. `executeUpdate(str)` 메소드를 이용하여 지정한 SQL 문장을 실행한다.

또한 **준비된 문장(prepared statement)**을 이용하면 파라미터를 바인딩하여 원하는 작업을 수행할 수 있다. **executePreparedStatement 멤버 함수**는 파라미터가 바인딩이 된 질의를 실행하는 기능(ⓒ)을 수행한다.

[예 2.4] tbJDBC - 준비된 문장의 선언과 파라미터 바인딩

```
private void executePreparedStatement() throws SQLException
{
    PreparedStatement pstmt =
         conn.prepareStatement("select name from emp where id = ?");    ... ① ...

    pstmt.setString(1, "1000");

    ResultSet rs = pstmt.executeQuery();

    while (rs.next()) {
        System.out.println(rs.getString(1));
    }

    pstmt.close();
}
```

구현된 내용을 차례대로 설명하면 다음과 같다.

1. 연결에 prepareStatement 메소드를 호출하여 준비된 문장을 생성한다.

   java.sql.PreparedStatement 인터페이스 객체로 반환되며, execute(), executeUpdate(), executeQuery() 메소드를 호출하여 실행할 수 있다.
2. `setString(bind_no, bind_value)` 메소드를 이용하여 바인딩할 파라미터의 순서를 정하여 값을 설정한다. **pstmt.setString(1, "1000")**은 준비된 문장의 첫 번째 파라미터(①)에 대응되는 **"1000"**이라는 문자열을 바인딩한다.
3. 파라미터의 바인딩이 완료된 후 `executeQuery` 메소드를 실행하면 결과 집합(ResultSet)를 얻게 된다. 결과 집합은 개념적으로 tbCLI나 tbESQL의 커서와 동일하다. 결과 집합에 `getString(column_no)` 메소드를 사용하면 실행 결과를 문자열로 확인할 수 있다.

2.4.3. 호출

tbJDBC를 이용하여 작성한 프로그램에서 PL/SQL를 호출(call)할 수 있다. 이러한 문장을 **호출 가능 문장**(callable statement)이라고 한다. **executeCallableStatement 멤버 함수**는 호출 가능 문장을 실행하는 기능(ⓓ)을 수행한다.

[예 2.5] tbJDBC - 호출

```
private void executeCallableStatement() throws SQLException
{
    String callSQL =
        " CREATE PROCEDURE testProc "+
        " (ID_VAL IN NUMBER, SAL_VAL IN OUT NUMBER) as " +
        " BEGIN" +
        "   update emp" +
        "      set salary = SAL_VAL" +
        "    where id = ID_VAL;" +
        "   select salary into SAL_VAL" +
        "     from emp" +
        "    where id = ID_VAl;" +
        " END;";
    String dropProc = "DROP PROCEDURE testProc";

    Statement stmt = conn.createStatement();

    try {
        stmt.executeUpdate(dropProc);
    } catch(SQLException e) {
        // if there is not the procedure
    }

    stmt.executeUpdate(callSQL);

    CallableStatement cstmt = conn.prepareCall("{call testProc(?, ?)}");
    cstmt.setInt(1, 1000);
    cstmt.setInt(2, 7000);
    cstmt.registerOutParameter(2, Types.INTEGER);
    cstmt.executeUpdate();

    int salary = cstmt.getInt(2);
    System.out.println(salary);

    stmt.close();
    cstmt.close();
}
```

구현된 내용을 차례대로 설명하면 다음과 같다.

1. 사용할 프러시저를 생성한다. **java.sql.Statement** 객체를 이용하여 프러시저 생성 SQL 문장을 실행한다.

2. 연결에 `prepareCall(str)` 메소드를 호출하여 호출 가능 문장을 생성한다. 생성된 호출 가능 문장은 **java.sql.CallableStatement** 인터페이스 객체로 반환된다.

3. 입력 파라미터와 출력 파라미터를 바인딩한다.

   입력 파라미터를 바인딩하는 방법은 `setInt(bind_no, bind_value)` 같은 바인딩 메소드(binding method)를 사용하며, 준비된 문장에서와 동일하게 사용할 수 있다.

   출력 파라미터에 대해서는 `registerOutParameter(bind_no, type)` 메소드를 이용하여 등록한다. 예제에서 등록할 출력 파라미터는 1번에서 생성된 프러시저의 두 번째 파라미터(**SAL_VAL IN OUT NUMBER**)이다. 이 파라미터는 입출력용으로 선언되었으므로 출력 파라미터로 등록할 수 있다.

4. `executeUpdate` 메소드를 호출하여 문장을 실행한 후 출력 파라미터의 타입에 맞는 메소드로 실행 결과를 확인한다. 프러시저의 출력 파라미터는 integer 타입이므로, CallableStatement 클래스의 `getInt(bind_no)` 메소드를 이용하여 결과 값을 가져온다.

2.4.4. 커밋과 롤백

트랜잭션은 하나 이상의 SQL 문장으로 이루어져 있다. SQL 문장이 실행되면서 전체가 커밋(Commit)되든지 롤백(Rollback)된다. tbJDBC는 Connection 객체에서 트랜잭션을 설정할 수 있다. 초기에는 **auto-commit**으로 설정되어 있다.

트랜잭션의 처리 모드를 변경하려면 다음과 같이 소스 코드를 추가해야 한다.

```
conn.setAutoCommit(false);
conn.rollback();
conn.commit();
conn.setTransactionIsolation(Connection.TRANSACTION_READ_UNCOMMITED)
```

2.4.5. 접속 해제

disconnect 멤버 함수는 데이터베이스 접속을 해제하는 기능(ⓔ)을 수행한다.

[예 2.6] tbJDBC - 접속 해제

```
private void disconnect() throws SQLException
{
    if (conn != null)
        conn.close();
}
```

conn 객체가 존재하는 경우 close 멤버 함수를 사용하여 데이터베이스 접속을 해제할 수 있다.

제3장 트리거의 사용

본 장에서는 트리거의 기본 개념과 이를 생성하는 방법을 설명한다.

3.1. 개요

트리거(Trigger)는 스키마 객체의 일종으로, 데이터베이스가 미리 정해 놓은 특정 조건이 만족되거나 어떤 동작이 수행되면 자동으로 실행되도록 정의한 동작이다. 예를 들어 데이터베이스에 특정한 이벤트가 발생되거나 사용자가 설정한 DDL 문장이 수행될 때 트리거가 실행될 수 있다.

트리거의 내용은 PSM으로 이루어져 있다. 트리거는 이미 정의된 PSM 객체를 호출하여 사용하거나 트리거를 생성할 때 이름 없는 블록을 함께 선언해주는 방식을 모두 이용할 수 있다.

3.1.1. 트리거 구성요소

트리거는 다음과 같은 세 가지 구성요소를 갖춰야만 생성할 수 있다.

- 트리거가 실행될 조건이 되는 문장이나 이벤트
- 실행 조건의 제약
- 실행될 내용

다음은 **alarm_for_balance**라는 트리거를 생성하는 예이다.

```
CREATE OR REPLACE TRIGGER alarm_for_balance
BEFORE INSERT OR UPDATE ON balance_tab        ... ⓐ ...
FOR EACH ROW
WHEN (new.balance < 3000)                     ..  ⓑ ...
    CALL alarm_for_balance_fn()               ..  ⓒ ...
```

ⓐ 테이블 balance_tab의 balance 컬럼에 로우가 삽입되거나 수정이 발생했을 때 ⓑ로 이동한다.

ⓑ 해당 로우의 값이 3000 이하인지를 검사한다.

ⓒ 검사한 결과가 맞으면 `alarm_for_balance_fn` 함수를 호출하고, 함수에 정의한 동작이 실행된다.

3.1.2. 트리거 타입

트리거의 타입은 다음과 같이 나눌 수 있다.

- 로우(row) 및 문장(statement)

타입	설명
로우	테이블에 INSERT, UPDATE, DELETE가 발생하는 로우마다 트리거의 내용이 실행되는 타입이다. 이 타입의 트리거는 각 로우에 연산이 발생할 때마다 연산 직전 또는 직후에 트리거가 실행된다.
문장	로우의 개수에 상관없이 문장 단위로 한 번만 실행되는 타입이다.

- BEFORE 및 AFTER

타입	설명
BEFORE	조건 문장이 실행되기 전에 트리거의 내용이 실행되는 타입이다.
AFTER	조건 문장이 실행된 후 트리거의 내용이 실행되는 타입이다.

트리거는 두 종류의 타입 중에서 하나씩을 각각 가질 수 있다. 즉, BEFORE 로우(BEFORE row), BEFORE 문장(BEFORE statement), AFTER 로우(AFTER row), AFTER 문장(AFTER statement)의 타입으로 생성할 수 있다.

3.2. 트리거 생성

트리거를 생성하는 방법은 다음과 같다.

```
CREATE [OR REPLACE] TRIGGER 트리거_이름
      {BEFORE | AFTER} {INSERT | UPDATE | DELETE} ON 테이블_이름
      [FOR EACH ROW]
WHEN (조건_제약)
    {[선언부]
BEGIN
     ...
END;} |
CALL 함수_또는_프러시저_이름
```

참고

트리거의 문법에 대한 자세한 내용은 "Tibero SQL 참조 안내서"를 참고한다.

다음은 로우 타입의 트리거로, 테이블 DECK_TBL의 COUNT 컬럼에 로우 값이 1000을 초과할 때마다 로그를 기록하도록 작성한 예이다.

[예 3.1] 트리거의 생성

```
CREATE OR REPLACE TRIGGER Log_overflow
AFTER UPDATE ON Deck_tbl
```

```
FOR EACH ROW
WHEN (new.count > 1000)
BEGIN
    INSERT
    INTO Deck_log (Deck_id, Timestamp, New_count, Action)
    VALUES (:new.Deck_no, SYSTIMESTAMP, :new.count, 'overflow');
END;
```

CREATE 문장은 Tibero 내부에서는 BEGIN – END 사이의 부분을 PSM으로 인식한다. 이 문장이 컴파일 되면 PSM 스키마 객체가 생성되고 이를 데이터베이스에 저장한다. 만약 컴파일 에러가 발생한다면 정적 뷰를 통해 에러의 내용을 확인할 수 있다.

제4장 XA의 사용

본 장에서는 분산 트랜잭션(Distributed Transaction)를 처리하는 데 사용되는 XA를 설명한다.

4.1. 분산 트랜잭션

하나의 데이터베이스 인스턴스 내에서는 한 트랜잭션으로 묶인 여러 개의 SQL 문장이 모두 커밋되거나 롤백된다. 네트워크로 연결된 여러 개의 데이터베이스 인스턴스가 참여하는 트랜잭션에서도 마찬가지로 각각 다른 데이터베이스 인스턴스에서 수행한 SQL 문장이 모두 동시에 커밋되거나 롤백될 수 있는 방법이 필요하다. 이렇게 여러 개의 노드 또는 다른 종류의 데이터베이스가 참여하는 하나의 트랜잭션을 **분산 트랜잭션**(Distributed Transaction)이라고 한다.

참고

분산 트랜잭션에 대한 자세한 내용은 "Tibero 관리자 안내서"를 참고한다.

4.1.1. Two-phase commit

Tibero는 X/Open DTP(Distributed Transaction Processing) 규약의 XA를 지원한다. XA는 Two-phase commit를 이용하여 분산 트랜잭션을 처리한다. 분산 환경에서 트랜잭션의 무결성을 보장하기 위해서 사용하는 커밋 방법은 **Two-phase commit**이다.

보통 두 개 이상의 노드가 특정 트랜잭션을 함께 수행하고 있다면, 일반적으로 사용자의 요청을 받아 트랜잭션을 시작한 노드가 코디네이터가 된다. TP-Monitor가 있는 시스템일 경우 TP-Monitor가 그 역할을 한다.

Two-phase commit mechanism은 크게 두 단계로 작업이 이루어진다. 위 예제를 기준으로 Two-phase commit를 설명하면 다음과 같다.

1. **First Phase(또는 Prepare Phase)**

   커밋을 준비하는 단계로 다음의 세부 과정으로 실행된다.

단계	설명
send "prepare"	각 노드는 코디네이터 노드로부터 커밋을 준비하라는 메시지를 받는다.
reply "prepared"	각 노드는 메시지를 받은 후 복구를 위해 로그 등에서 커밋이 가능한지를 검사한다. 또는 필요에 따라 자체적으로 롤백을 한다. 만약 커밋이 가능하다면 최종으로 커밋한 로그를 제외한 모든 작업을 수행한 후 커밋이 준비되었다는 메시지를 코디네이터 노드로 전달한다.

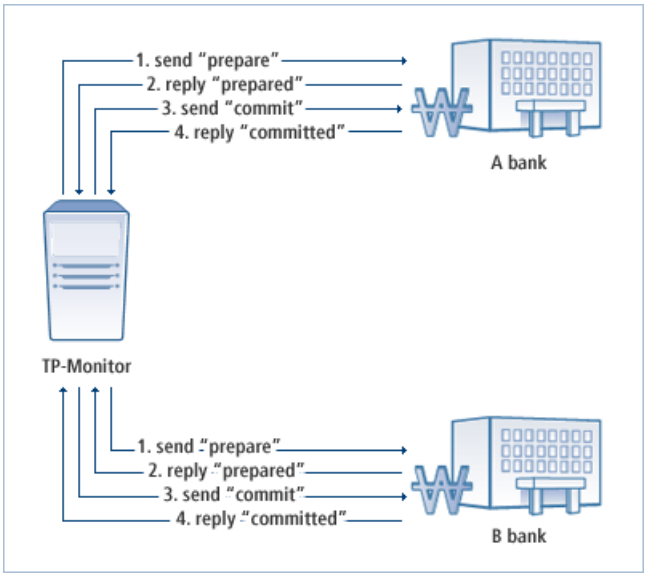
2. **Second Phase(또는 Commit Phase)**

실제로 커밋한 기록을 저장하는 단계로 Second Phase은 다음의 세부 과정으로 실행된다.

단계	설명
send "commit"	코디네이터 노드는 모든 노드에서 커밋이 되었다는 메시지를 전달받는다. 이를 확인한 후 실제로 커밋을 실행하라는 메시지를 각 노드에 보낸다.
reply "committed"	각 노드는 커밋을 기록한 후 커밋이 완료되었다는 메시지를 코디네이터 노드로 전달한다.

다음은 Two-phase commit의 일반적인 예를 나타내는 그림이다.

[그림 4.1] Two-phase commit의 일반적인 예



4.1.2. In-doubt 트랜잭션

Two-phase commit mechanism에 의해 첫 번째 prepare 메시지를 받으면 데이터베이스는 분산 트랜잭션에 해당하는 리소스를 잠금을 설정하거나 로그를 남김으로써 커밋할 준비를 한다. 그런데 **prepare**까지 마친 상태에서 네트워크의 이상으로 다음 메시지(커밋 또는 롤백)를 받지 못하는 경우가 발생할 수 있다.

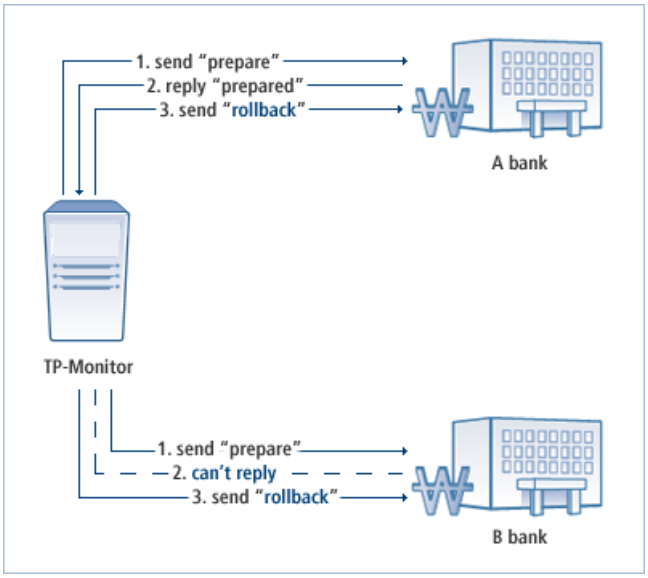
이 경우에 데이터베이스는 해당 트랜잭션을 커밋해야 할 지 롤백해야 할지 판단할 수 없다. 따라서 다음 메시지가 올 때까지 **prepare**된 리소스에 잠금 설정을 한 채로 기다리게 된다. 이렇게 prepare는 되었는데 그 다음 메시지를 받지 못한 채 리소스만 소유하고 기다리고 있는 트랜잭션을 **In-doubt 트랜잭션**이라 한다.

예를 들어 First Phase에서 코디네이터 노드가 커밋을 준비하라는 메시지를 보냈음에도 불구하고 어떤 특정 노드의 서버가 다운되었거나 네트워크 상태가 불안정하여 그 메시지를 못 받았거나 또는 메시지는 받았지만 커밋이 준비되었다는 답변을 받지 못했을 때 In-doubt 트랜잭션이 발생한다.

이러한 경우 코디네이터 노드는 다른 모든 노드의 응답을 받았어도 한 노드의 응답을 받지 못했으므로, 이 트랜잭션을 In-doubt 트랜잭션으로 표시하고, 모든 노드에 Second Phase의 커밋 명령을 실행하라는 메시지를 보내는 대신에 롤백하라는 메시지를 보낸다.

예를 들면 다음 그림과 같다.

[그림 4.2] In-doubt 트랜잭션이 발생하는 예



또한 모든 노드에서 커밋이 준비되었다는 메시지를 받아서 확인했더라도 그 이후에 코디네이터 노드가 보낸 Second Phase의 커밋 명령을 실행하라는 메시지를 특정 노드가 받지 못했거나, 그 노드가 커밋 명령은 잘 수행하였으나 커밋이 완료되었다는 응답을 코디네이터 노드에 전달하지 못했을 수도 있다. 이와 같은 경우도 마찬가지로 In-doubt 트랜잭션으로 분류된다.

트랜잭션은 커밋이나 롤백이 확정되지 않은 채로 데이터베이스 리소스에 대해 잠금(Lock)이 설정된 상태 즉 정체(Pending) 상태가 된다. 따라서 이를 해결하려면 첫 번째 경우처럼 코디네이터 노드가 자동으로 전체 트랜잭션을 롤백해 준다거나 DBA가 이러한 트랜잭션을 추출하여 자체적으로 판단하여 수동으로 처리할 수 있다.

이때 DBA가 In-doubt 트랜잭션을 추출하기 위해 정보를 볼 수 있는 테이블과 뷰로는 **VT_XA_BRANCH**와 **DBA_2PC_PENDING**이 있다.

참고

VT_XA_BRANCH는 현재 데이터베이스 서버에서 활동 중인 모든 XA 트랜잭션 브랜치(XA transaction branch)의 정보를 실시간으로 볼 수 있는 테이블이다. 그리고 DBA_2PC_PENDING은 현재 정체되고 있는 XA 트랜잭션 브랜치의 정보를 보여주는 뷰이다. 이에 대한 자세한 내용은 각각 "Tibero 참조 안내서"와 "Tibero 관리자 안내서"를 참고한다.

4.2. XA API

TP-Monitor를 이용하여 분산 트랜잭션을 수행하기 위해서는 XA API를 사용해야 한다.

Tibero는 X/Open DTP 규약의 XA를 지원하기 때문에 표준에 맞는 XA 애플리케이션 프로그램을 작성할 수 있다.

4.2.1. XA 함수

Tibero에서는 XA를 지원하기 위해 다음과 같은 XA 함수를 제공한다. 단, 이 함수는 **C 프로그래밍 언어용**으로만 제공된다.

함수	설명
xa_open	리소스 매니저(Resource Manager)에 접속한다. 자세한 설명은 "4.2.2. xa_open 함수의 속성"을 참고한다.
xa_close	리소스 매니저에서 데이터베이스 접속을 해제한다.
xa_start	XID에 새로운 트랜잭션을 시작하거나, 이미 존재하는 트랜잭션에 현재 프로세스를 연결한다.
xa_end	XID에서 현재 프로세스를 분리한다.
xa_rollback	XID의 트랜잭션을 롤백한다.
xa_prepare	XID에 커밋을 준비한다. Two-phase commit의 First Phase이다.
xa_commit	XID에 커밋을 완료한다. Two-phase commit의 Second Phase이다.
xa_recover	prepare 상태인 트랜잭션의 목록을 검사하여 커밋이나 롤백을 수행한다.
xa_forget	XID의 트랜잭션이 이미 처리된 경우 로그 기록을 삭제한다.

4.2.2. xa_open 함수의 속성

다음은 xa_open 함수를 호출하는 데 필요한 속성이다.

속성	필수	설명
user	O	접속할 사용자의 이름이다. (예: user=tibero)
pwd	O	접속할 사용자의 패스워드이다. (예: pwd=1234)
db	X	접속할 데이터베이스의 DSN 이름(tbdsn.tbr 파일 내의 DSN 이름)이다. (예: db=sample)
conn_id	X	XA 연결에 이름을 부여한다. 이 이름을 ESQL의 AT 구문에서 사용할 수 있다. (예: conn_id=db1)

속성	필수	설명
Loose_Coupling	O	다른 브랜치이지만 동일한 글로벌 트랜잭션끼리 같은 리소스를 사용하는지 여부를 설정한다. (true/false) loose coupling이면 (1,0)과 (1,2)는 서로 다른 내부 리소스를 사용한다. 예를 들어 TX가 해당된다. 이와는 반대로 tight coupling이면 두 xtb는 하나의 TX를 서로 잠금 처리를 설정해가며 공유하여 사용한다. (예: Loose_Coupling=false)
sestm	O	시스템에 의해 중단(abort)되기 전까지의 트랜잭션의 inactive time limit이다. 클라이언트로부터 한 요청에 대한 답변을 전달한 후 그 다음 요청이 오기 전까지 대기하는 시간이다. 그 이상의 시간이 지나면 클라이언트 측에서 문제가 있다고 판단하고 해당 xtb를 롤백한다. (예: sestm=10)
seswt	X	XA_RETRY가 반환되기 전까지 데이터베이스 서버가 트랜잭션을 기다리는 대기 시간이다. flag 내에 TMNOWAIT이 설정되어 있지 않을 경우에는 클라이언트의 요청을 데이터베이스가 곧바로 처리해줄 수 없을 때 해당 리소스를 사용할 수 있을 때까지 기다린 후에 클라이언트에 답변을 주게 된다. 이때 데이터베이스가 seswt로 설정한 시간까지 답변을 줄 수 없는 경우 XARETRY를 반환한다. (예: seswt=30)
logdir	X	XA의 로그를 저장할 디렉터리를 지정할 수 있다. (예: logdir=/home/test/log)

4.2.3. XA 애플리케이션 프로그래밍

일반적으로 C 프로그래밍 언어에서 XA 애플리케이션 프로그램을 작성할 때 ESQL을 이용하여 개발하는 예가 많다.

다음은 Tibero에서 제공하는 **tbESQL**를 이용하여 XA 애플리케이션 프로그램을 작성한 예이다.

```
#define XA_CONN_STR_TIGHT "TIBERO_XA:user=tibero, pwd=tmax," \
                          "db=sample, sestm=60, logdir=/home/path/to/xa_log"

/* 동일한 글로벌 트랜잭션에서 다른 브랜치로 xa_start했을 경우
* Tight-Coupling
*/
void test_xa_2branch_2pc_tight()
{
    int rc;
    XID xid1;
    XID xid2;
    struct xa_switch_t *tbxa = &XA_SWITCH_NAME;
    char *conn_str = XA_CONN_STR_TIGHT; /* tightly coupled */
```

```
long gtrid = _GTRID_BASE;
long bqual = 1;

EXEC SQL BEGIN DECLARE SECTION;
    int cnt;
EXEC SQL END DECLARE SECTION;

/* xa_open */
tbxa->xa_open_entry (conn_str, 0, TMNOFLAGS);
xid1.formatID = 1;
xid1.gtrid_length = sizeof(gtrid);
xid1.bqual_length = sizeof(bqual);
memcpy(&xid1.data[0], &gtrid, sizeof(gtrid));
memcpy(&xid1.data[sizeof(gtrid)], &bqual, sizeof(bqual));

bqual = 2;
xid2.formatID = 1;
xid2.gtrid_length = sizeof(gtrid);
xid2.bqual_length = sizeof(bqual);
memcpy(&xid2.data[0], &gtrid, sizeof(gtrid));
memcpy(&xid2.data[sizeof(gtrid)], &bqual,
sizeof(bqual));

EXEC SQL DELETE FROM PERSON;
EXEC SQL COMMIT WORK;

/* (1, 1) 시작 */
/* xa_start -- sql statements starts */
tbxa->xa_start_entry (&xid1, 0, TMNOFLAGS);
EXEC SQL INSERT INTO PERSON VALUES ('1', 'LEE');
EXEC SQL INSERT INTO PERSON VALUES ('2', 'KIM');

/* xa_end -- */
tbxa->xa_end_entry (&xid1, 0, TMSUCCESS);

/* (1, 2) 시작 */
/* xa_start -- sql statements starts */
tbxa->xa_start_entry (&xid2, 0, TMNOFLAGS);
EXEC SQL INSERT INTO PERSON VALUES ('2', 'PARK');
EXEC SQL INSERT INTO PERSON VALUES ('3', 'JAKE');
EXEC SQL INSERT INTO PERSON VALUES ('4', 'KID');
EXEC SQL INSERT INTO PERSON VALUES ('5', 'CHANHO');

/* xa_end -- */
tbxa->xa_end_entry (&xid2, 0, TMSUCCESS);

/* xa_prepare */
```

```
    tbxa->xa_prepare_entry (&xid1, 0, TMNOFLAGS);

    /* Tightly-Coupled 가정 */
    /* xa_prepare */
    tbxa->xa_prepare_entry (&xid2, 0, TMNOFLAGS);

    /* xa_commit */
    tbxa->xa_commit_entry (&xid1, 0, TMNOFLAGS);

    /* xa_commit */
    tbxa->xa_commit_entry (&xid2, 0, TMNOFLAGS);
    EXEC SQL SELECT COUNT(*) into :cnt FROM PERSON;
    CuAssertIntEq(tc, cnt, 6);

    /* xa_close */
    tbxa->xa_close_entry ("", 0, TMNOFLAGS);
    return;
}

/* 같은 Global Trasaction의 다른 Branch로 xa_start했을 경우
* Loose-Coupling
*/
void test_xa_2branch_2pc_loose(CuTest *tc)
{
    int rc;
    XID xid1;
    XID xid2;
    struct xa_switch_t *tbxa = &XA_SWITCH_NAME;
    char *conn_str = XA_CONN_STR_LOOSE;
    long gtrid = _GTRID_BASE;
    long bqual = 1;

    EXEC SQL BEGIN DECLARE SECTION;
        int cnt;
    EXEC SQL END DECLARE SECTION;

    /* xa_open */
    tbxa->xa_open_entry (conn_str, 0, TMNOFLAGS);
    xid1.formatID = 1;
    xid1.gtrid_length = sizeof(gtrid);
    xid1.bqual_length = sizeof(bqual);
    memcpy(&xid1.data[0], &gtrid, sizeof(gtrid));
    memcpy(&xid1.data[sizeof(gtrid)], &bqual, sizeof(bqual));

    bqual = 2;
    xid2.formatID = 1;
    xid2.gtrid_length = sizeof(gtrid);
```

```
    xid2.bqual_length = sizeof(bqual);
    memcpy(&xid2.data[0], &gtrid, sizeof(gtrid));
    memcpy(&xid2.data[sizeof(gtrid)], &bqual,
    sizeof(bqual));

    EXEC SQL DELETE FROM PERSON;
    EXEC SQL COMMIT WORK;

    /* (1, 1) 시작 */
    /* xa_start -- sql statements starts */
    tbxa->xa_start_entry (&xid1, 0, TMNOFLAGS);
    EXEC SQL INSERT INTO PERSON VALUES ('1', 'LEE');
    EXEC SQL INSERT INTO PERSON VALUES ('2', 'KIM');

    /* xa_end -- */
    tbxa->xa_end_entry (&xid1, 0, TMSUCCESS);

    /* (1, 2) 시작 */
    /* xa_start -- sql statements starts */
    tbxa->xa_start_entry (&xid2, 0, TMNOFLAGS);
    EXEC SQL INSERT INTO PERSON VALUES ('2', 'PARK');
    EXEC SQL INSERT INTO PERSON VALUES ('3', 'JAKE');
    EXEC SQL INSERT INTO PERSON VALUES ('4', 'KID');
    EXEC SQL INSERT INTO PERSON VALUES ('5', 'CHANHO');

    /* xa_end -- */
    tbxa->xa_end_entry (&xid2, 0, TMSUCCESS);

    /* xa_prepare */
    tbxa->xa_prepare_entry (&xid1, 0, TMNOFLAGS);

    /* Loosely-Coupled 가정 */
    /* xa_prepare */
    tbxa->xa_prepare_entry (&xid2, 0, TMNOFLAGS);

    /* xa_commit */
    tbxa->xa_commit_entry (&xid1, 0, TMNOFLAGS);

    /* xa_commit */
    tbxa->xa_commit_entry (&xid2, 0, TMNOFLAGS);
    EXEC SQL SELECT COUNT(*) into :cnt FROM PERSON;
    CuAssertIntEq(tc, cnt, 6);

    /* xa_close */
    tbxa->xa_close_entry ("", 0, TMNOFLAGS);
    return;
}
```

4.3. JDBC에서의 XA 지원

본 절에서는 XA(Extended Architecture)에서 지원하는 인터페이스와 이를 이용하여 작성한 프로그램에 대해 설명한다.

4.3.1. XA 인터페이스

Tibero에서 지원하는 JDBC의 XA 인터페이스는 다음과 같다.

- XA Datasource Interface
- XA Connection Interface
- XA Exception Interface
- XA XID Interface

다음은 Tibero에서 구현된 XA 인터페이스의 목록이다.

표준 XA 인터페이스(JDK 1.3)	**Tibero의 XA 인터페이스**
javax.sql.XADataSource	com.tmax.tibero.jdbc.ext.TbXADataSource
javax.sql.XAConnection	com.tmax.tibero.jdbc.ext.TbXAConnection
javax.transaction.xa.XAException	com.tmax.tibero.jdbc.ext.TbXAException
javax.transaction.xa.Xid	com.tmax.tibero.jdbc.ext.TbXid

4.3.2. XA 인터페이스 프로그래밍

다음은 tbJDBC 환경에서 XA 인터페이스를 이용하여 프로그래밍한 예이다.

```
package test.com.tmax.tibero.cases;
import com.tmax.tibero.jdbc.ext.*;
import test.com.tmax.tibero.AbstractBase;
import javax.sql.XAConnection;
import javax.transaction.xa.XAResource;
import java.sql.*;

public class TestXATwoBranch extends AbstractBase{
    int formatID = 1;
    int row_count = 0;
    int pre1 , pre2 = 0;

    byte[] gtid1 = new byte[1];
    byte[] bq1 = new byte[1];
    byte[] gtid2 = new byte[1];
```

```
byte[] bq2 = new byte[1];

public TestXATwoBranch (String name) {
super(name);
}

/* Tightly Coupled 일 때 */
public void test_xa_2branch_2pc_tight () throws Exception {
    debug("test_xa_2branch_2pc_tight " + this._getClassName());

    create_table_for_xa();

    gtid1[0] = (byte)1; bq1[0] = (byte)1;
    gtid2[0] = (byte)1; bq2[0] = (byte)2;

    TbXADataSource xads1 = new TbXADataSource();
    xads1.setUrl(getXAurl());
    xads1.setUser(getXAuser());
    xads1.setPassword(getXApasswd());

    TbXADataSource xads2 = new TbXADataSource();
    xads2.setUrl(getXAurl());
    xads2.setUser(getXAuser());
    xads2.setPassword(getXApasswd());

    XAConnection xacon1 = xads1.getXAConnection();
    XAConnection xacon2 = xads2.getXAConnection();

    Connection conn1 = xacon1.getConnection();
    Connection conn2 = xacon2.getConnection();

    XAResource xars1 = xacon1.getXAResource();
    XAResource xars2 = xacon2.getXAResource();

    /* XID (1.1) 생성 */
    TbXid xid1 = new TbXid(formatID, gtid1,bq1 );

    /* XID (1.2) 생성 */
    TbXid xid2 = new TbXid(formatID, gtid2, bq2);
    try {
        /* (1.1) 시작 */
        xars1.start(xid1, XAResource.TMNOFLAGS);

        PreparedStatement pstmt1;
        pstmt1 = conn1.prepareStatement("insert into author values (?,?)");

        pstmt1.setInt(1, 1);
```

```
pstmt1.setString(2, "FOSCHIA");
pstmt1.executeUpdate();

pstmt1.setInt(1,2);
pstmt1.setString(2, "AGNOS");
pstmt1.executeUpdate();

/* (1.1) 종료 */
xars1.end(xid1, XAResource.TMSUCCESS);

/* (1.2) 시작 */
xars2.start(xid2, XAResource.TMNOFLAGS);
PreparedStatement pstmt2;
pstmt2 = conn2.prepareStatement("insert into author values (?,?)");

pstmt2.setInt(1, 3);
pstmt2.setString(2, "JELLA");
pstmt2.executeUpdate();

pstmt2.setInt(1,4);
pstmt2.setString(2, "THIPHILO");
pstmt2.executeUpdate();

/* (1.2) 종료 */
xars2.end(xid2, XAResource.TMSUCCESS);

/* (1,1) prepare */
pre1= xars1.prepare(xid1);
assertEquals(pre1, XAResource.XA_RDONLY);

/* (1,2) prepare */
pre2 = xars2.prepare(xid2);
assertEquals(pre2, XAResource.XA_OK);

/* (1.1) commit */
try {
    xars1.commit(xid1, false);
} catch(TbXAException e) {}

/* (1.2) commit */
try {
    xars2.commit(xid2, false);
} catch(TbXAException e) {}

Statement stmt1 = conn1.createStatement();
ResultSet rs1 = stmt1.executeQuery("select * from author");
while (rs1.next())
```

```
            row_count ++;

        assertEquals(4, row_count);

        rs1.close(); rs1 = null;
        stmt1.close(); stmt1 = null;

        pstmt1.close(); conn1.close(); xacon1.close();
        pstmt2.close(); conn2.close(); xacon2.close();

        pstmt1= null; conn1= null; xacon1=null;
        pstmt2= null; conn2= null; xacon2=null;
    } catch (TbXAException e) {}
}

/* Loosely Coupled 일 때 */
public void test_xa_2branch_2pc_loose () throws Exception {
    debug("test_xa_2branch_2pc_loose " + this._getClassName());

    create_table_for_xa();

    gtid1[0] = (byte)1; bq1[0] = (byte)1;
    gtid2[0] = (byte)1; bq2[0] = (byte)2;

    TbXADataSource xads1 = new TbXADataSource();

    xads1.setUrl(getXAurl());
    xads1.setUser(getXAuser());
    xads1.setPassword(getXApasswd());

    TbXADataSource xads2 = new TbXADataSource();

    xads2.setUrl(getXAurl());
    xads2.setUser(getXAuser());
    xads2.setPassword(getXApasswd());

    XAConnection xacon1 = xads1.getXAConnection();
    XAConnection xacon2 = xads2.getXAConnection();

    Connection conn1 = xacon1.getConnection();
    Connection conn2 = xacon2.getConnection();

    XAResource xars1 = xacon1.getXAResource();
    XAResource xars2 = xacon2.getXAResource();

    TbXid xid1 = new TbXid(formatID, gtid1,bq1 );
    TbXid xid2 = new TbXid(formatID, gtid2, bq2);
```

```
        try {
            xars1.start(xid1, TbXAResource.TBRTRANSLOOSE);

            PreparedStatement pstmt1;
            pstmt1 = conn1.prepareStatement("insert into author values (?,?)");

            pstmt1.setInt(1, 1);
            pstmt1.setString(2, "FOSCHIA");
            pstmt1.executeUpdate();

            pstmt1.setInt(1,2);
            pstmt1.setString(2, "AGNOS");
            pstmt1.executeUpdate();

            xars1.end(xid1, XAResource.TMSUCCESS);
            xars2.start(xid2, TbXAResource.TBRTRANSLOOSE);

            PreparedStatement pstmt2;
            pstmt2 = conn2.prepareStatement("insert into author values (?,?)");

            pstmt2.setInt(1, 3);
            pstmt2.setString(2, "JELLA");
            pstmt2.executeUpdate();

            pstmt2.setInt(1,4);
            pstmt2.setString(2, "THIPHILO");
            pstmt2.executeUpdate();

            xars2.end(xid2, XAResource.TMSUCCESS);

            pre1= xars1.prepare(xid1);
            assertEquals(pre1, XAResource.XA_OK);

            pre2 = xars2.prepare(xid2);
            assertEquals(pre2, XAResource.XA_OK);

            xars1.commit(xid1, false);
            xars2.commit(xid2, false);

            Statement stmt1 = conn1.createStatement();
            ResultSet rs1 = stmt1.executeQuery("select * from author");

            while (rs1.next())
            row_count ++;

            assertEquals(4, row_count);
```

```
            rs1.close(); rs1 = null;
            stmt1.close(); stmt1 = null;

            pstmt1.close(); conn1.close(); xacon1.close();
            pstmt2.close(); conn2.close(); xacon2.close();

            pstmt1= null; conn1= null; xacon1=null;
            pstmt2= null; conn2= null; xacon2=null;
        } catch (TbXAException e) {}
    }
}
```

4.4. TP-Monitor와 Tibero 연동

TP-Monitor(Transaction Processing Monitor)는 각종 프로토콜에서 동작하는 세션과 시스템 및 데이터베이스 사이의 최소 처리 단위인 트랜잭션을 감시하여 일관성 있게 보관 및 유지하는 역할을 하는 트랜잭션 관리 미들웨어이다. 본 절에서는 대표적인 상용 TP-Monitor인 Tmax와 Tuxedo를 Tibero와 연동하는 예제를 설명한다.

4.4.1. Tmax와 Tibero 연동

Tmax는 Transaction Maximization의 약어로 트랜잭션 처리 극대화를 의미한다. Tmax는 시스템의 분산 환경에서 이기종 컴퓨터 간의 트랜잭션 처리를 완벽히 보장하면서 부하를 분산시키고 에러가 발생하는 경우 적절한 조치를 담당하는 TP-Monitor이다. 트랜잭션의 특성을 지원하면서 사용자에게는 최적의 개발 환경을 제공하고, 클라이언트/서버 환경에서 최적의 솔루션을 제공하며, 성능 개선은 물론 모든 장애에 완벽하게 대처한다.

Tmax는 분산 트랜잭션 프로세싱의 국제 표준인 X/Open DTP(Distributed Transaction Processing) 모델을 준수하고 국제 표준기구인 OSI(Open Systems Interconnection group)의 DTP 서비스에 대한 기능적 분산과 기능 구성 요소 간 API 및 시스템 인터페이스 정의에 따라 개발되었다. 또한 분산 환경에서 이기종 간의 투명한 업무 처리 및 OLTP(On-Line Transaction Processing)를 지원하고 트랜잭션 처리의 ACID(Atomic, Consistent, Isolated, Durable: transaction properties) 특성을 만족하게 한다.

아래에서 소개할 Tmax와 Tibero 연동 예제 프로그램을 테스트해보기 위해서는 Tmax와 Tibero가 정상적으로 설치되어 있어야 한다.

참고

1. 연동할 Tmax와 Tibero가 다른 머신에 설치된 경우 Tmax가 설치된 머신에서 Tibero클라이언트를 따로 설치하여 Tibero 서버에 정상적으로 접속할 수 있는 환경이 구축되어야 한다.

2. Tmax 설치 및 관리에 대한 자세한 내용은 "Tmax Installation Guide"나 "Tmax Administration Guide"를 참고한다. Tibero로 설치 및 관리에 관한 자세한 내용은 "Tibero 설치 안내서"와 "Tibero 관리자 안내서"를 참고한다.

여기서 소개하는 예제 프로그램은 Tmax를 설치할 때 인스톨러가 기본으로 제공하는 것이다. 클라이언트가 Tmax 서버를 통하여 Tibero DB를 접속하여 특정 테이블의 데이터를 조회, 추가, 변경, 삭제하는 작업을 한다.

테스트 환경과 프로그램에 사용된 각종 파일들은 다음과 같다.

- 테스트 환경

구분	설명
운영체제	Ubuntu Linux 2.6.32-24-server x86-64
셸	bash
$TMAXDIR	Tmax 설치 디렉터리

- 프로그램 파일

파일	설명
sample.m	Tmax 환경설정 파일($TMAXDIR/config)
tms_tbr.mk	Tibero용 TMS makefile($TMAXDIR/sample/server)
tbrtest.tbc	서버 프로그램 tbESQL/C 파일($TMAXDIR/sample/server)
tbrtest.h	서버 프로그램 헤더 파일($TMAXDIR/sample/server)
Makefile.tbr	서버 프로그램 makefile($TMAXDIR/sample/server)
compile	서버 프로그램 빌드 스크립트($TMAXDIR/sample/server)
tbr_main.c	클라이언트 프로그램 파일($TMAXDIR/sample/client)
Makefile.c	클라이언트 프로그램 makefile($TMAXDIR/sample/client)
compile	클라이언트 프로그램 빌드 스크립트($TMAXDIR/sample/client)

다음에 제시된 순서대로 따르면 Tmax와 Tibero가 연동하는 것을 확인할 수 있다.

1. Tmax 기본 환경설정
2. TMS 컴파일
3. TMS 컴파일
4. 서버 프로그램 컴파일
5. 클라이언트 프로그램 컴파일
6. DB 테이블 생성
7. 예제 프로그램 실행

Tmax 기본 환경설정

다음은 Tmax 기본 환경을 설정하는 방법이다.

- **Tmax 시스템 환경 파일 설정**

  $TMAXDIR/config 디렉터리에 있는 sample.m은 Tmax를 기동시킬 때 필요한 각종 정보들이 들어있는 Tmax 시스템 환경 파일이다. ASCII 파일 형태로 작성하며, cfl 유틸리티로 컴파일하여 이진 파일을 생성한다. 생성된 이진 파일은 Tmax 기동 및 종료할 때 참조된다.

  Tibero 서버와 연동하는 서비스를 활성화시키기 위해 sample.m 환경 파일의 SVRGROUP 절, SERVER 절, SERVICE 절 항목을 아래와 같이 수정한다.

```
### tms for Tibero ###
svg4            NODENAME = "integrity", DBNAME = TIBERO,
                OPENINFO = "TIBERO_XA:user=tibero,pwd=tmax,sestm=60,db=tibero",
                TMSNAME  = tms_tbr

### server for Tibero sample program ###
tbrtest         SVGNAME = svg4

### services for tbrtest ###
TBRINS          SVRNAME = tbrtest
TBRSEL          SVRNAME = tbrtest
TBRUPT          SVRNAME = tbrtest
TBRDEL          SVRNAME = tbrtest
```

  SVRGROUP 절의 NODENAME은 Tmax를 설치할 때에 hostname 값으로 자동 설정된다. DB 벤더를 구분하기 위한 DBNAME은 TIBERO로 설정되어 있다. OPENINFO는 XA 모드 설정을 위한 TIBERO_XA가 앞쪽에 쓰여 있고 그 뒤에 “4.2.2. xa_open 함수의 속성”이 나열되어 있다. TMSNAME에는 XA를 담당할 모듈 이름이 설정되어 있다.

  SERVER 절에는 예제 서버 프로그램의 이름인 tbrtest가 설정되고, 서버 프로그램이 제공하는 서비스 4가지가 설정되어 있다.

참고

Tmax 시스템 환경 파일 설정에 대한 자세한 내용은 "Tmax Administration Guide"를 참고한다.

- **Tmax 시스템 환경 파일 컴파일**

  수정한 sample.m 파일을 아래와 같은 명령으로 컴파일한다.

```
cfl -i sample.m
```

  성공적으로 컴파일 된 후에는 다음과 같은 메시지가 출력된다.

```
CFL is done successfully for node(<nodename>)
```

- **서비스 테이블 생성**

  서비스 테이블은 각각의 Tmax 시스템 내 서버 프로세스가 생성될 때 필요한 파일로서 각각의 프로세스들이 어떤 서비스를 처리하는지에 대한 정보가 담겨 있다. 아래와 같은 명령으로 서비스 테이블을 생성한다.

  ```
  gst
  ```

  성공적으로 처리되면 다음과 같은 메시지가 출력된다.

  ```
  SVC tables are successfully generated GST is successfully done
  ```

- **구조체 정의 이진 파일 및 필드 키 정의 이진 파일 생성**

  서버나 클라이언트 프로그램내에서 구조체나 필드 키를 쓰는 경우 이와 관련된 이진 파일을 생성해 주어야 한다. 하지만 이 예제 프로그램에서는 구조체나 필드 키를 사용하지 않으므로 그 과정을 생략한다.

TMS 컴파일

TMS(Transaction Management Server)는 Tmax 시스템의 구성요소로서 데이터베이스 관리 및 분산 트랜잭션 처리를 담당하는 프로세스이다. Tibero용 TMS를 컴파일하기 전에 Tibero 관련 환경변수 TB_HOME, TB_SID, LD_LIBRARY_PATH, PATH 등이 제대로 설정되었는지 확인한다.

확인한 다음 아래와 같이 $TMAXDIR/sample/server 디렉터리로 이동하여 Tibero용 TMS makefile을 이용하여 TMS를 컴파일한다.

```
cd $TMAXDIR/sample/server
make –f tms_tbr.mk all
```

서버 프로그램 컴파일

$TMAXDIR/sample/server 디렉터리로 이동하여 실제적으로 서비스를 제공하는 서버 프로그램을 빌드 스크립트를 이용하여 컴파일한다.

```
cd $TMAXDIR/sample/server
./compile tbc tbrtest
```

클라이언트 프로그램 컴파일

$TMAXDIR/sample/client 디렉터리로 이동하여 서비스를 요청하는 클라이언트 프로그램을 빌드 스크립트를 이용하여 컴파일한다.

```
cd $TMAXDIR/sample/client
./compile c tbr_main
```

DB 테이블 생성

Tibero 서버에 tibero/tmax 계정으로 접속하여 아래와 같은 emp 테이블을 생성한다.

```
tbsqltibero/tmax

create table emp (
    empno number,
    ename char(16),
    job char(16),
    hiredate char(16),
    sal number
);
```

예제 프로그램 실행

- **Tmax 시스템 기동**

  다음과 명령으로 Tmax를 기동한다.

  ```
  tmboot
  ```

  성공적으로 기동되면 다음과 같은 메시지가 출력된다.

  ```
  TMBOOT for node(<nodename>) is starting:
  Welcome to Tmax demo system: it will expire 2012/3/11
  Today: 2012/1/13
          TMBOOT: TMM is starting: Fri Jan 13 14:18:31 2012
          TMBOOT: CLL is starting: Fri Jan 13 14:18:31 2012
          TMBOOT: CLH is starting: Fri Jan 13 14:18:31 2012
  (I) CLH9991 Current Tmax Configuration: Number of client handler(MINCLH) = 1
                  Supported maximum user per node = 680
                  Supported maximum user per handler = 680 [CLH0125]
          TMBOOT: TLM(tlm) is starting: Fri Jan 13 14:18:31 2012
          TMBOOT: TMS(tms_tbr) is starting: Fri Jan 13 14:18:31 2012
  (I) TMS0211 General Infomation : transaction recovery will be started [TMS0221]
  (I) TMS0211 General Infomation : transaction recovery was completed [TMS0222]
          TMBOOT: TMS(tms_tbr) is starting: Fri Jan 13 14:18:31 2012
          TMBOOT: SVR(tbrtest) is starting: Fri Jan 13 14:18:31 2012
  ```

- **클라이언트 프로그램 실행**

  클라이언트 프로그램의 명령 옵션은 다음과 같다.

  ```
  Usage: ./tbr_main empno loop_cnt ins_flag upt_flag del_flag
  flag : 1|0
  ```

  원하는 옵션을 선택하여 수행시키면 아래와 같은 결과가 출력된다.

```
./tbr_main 12 3 1 0 0

 LOOP COUNT = 1
>> INSERT : COMMIT TEST
[./tbr_main] [[TBRINS] emp Insert Success]
[./tbr_main] [[TBRSEL] emp Select Success [1]]

 LOOP COUNT = 2
>> INSERT : COMMIT TEST
[./tbr_main] [[TBRINS] emp Insert Success]
[./tbr_main] [[TBRSEL] emp Select Success [2]]

 LOOP COUNT = 3
>> INSERT : COMMIT TEST
[./tbr_main] [[TBRINS] emp Insert Success]
[./tbr_main] [[TBRSEL] emp Select Success [3]]
```

4.4.2. Tuxedo와 Tibero 연동

Tuxedo는 분산 트랜잭션 처리를 위한 플랫폼으로서, C, C++ 및 COBOL로 작성된 소프트웨어를 위한 개방형의 분산 시스템을 토대로 Mainframe 확장성 및 성능을 제공한다. 또한 메인스트림 하드웨어를 토대로 메인프레임 애플리케이션을 “리호스팅” 할 수 있는 플랫폼이다.

Tuxedo는 비용 효과적인 신뢰성과 초당 수십만 건의 트랜잭션을 지원할 수 있는 탁월한 확장성을 제공하는 것은 물론, SOA와 같은 혁신적 아키텍처의 일부로서 기존 IT 자산의 수명을 연장함으로써 투자 보호의 이점을 제공한다. Oracle Tuxedo는 Oracle Fusion Middleware의 전략적 트랜잭션 처리 제품이다.

아래에서 소개할 Tuxedo와 Tibero 연동 예제 프로그램을 테스트하기 위해서는 Tuxedo와 Tibero가 정상적으로 설치되어 있어야 한다.

참고

1. 연동할 Tmax와 Tibero가 다른 머신에 설치된 경우 Tmax가 설치된 머신에서 Tibero 클라이언트를 따로 설치하여 Tibero 서버에 정상적으로 접속할 수 있는 환경이 구축되어야 한다.

2. Tuxedo 설치 및 관리에 대한 자세한 내용은 "Tuxedo Documenation"을 참고한다. Tibero로 설치 및 관리에 관한 자세한 내용은 "Tibero 설치 안내서"와 "Tibero 관리자 안내서"를 참고한다.

여기서 소개하는 예제 프로그램은 클라이언트가 Tuxedo 서버를 통하여 Tibero DB를 접속하여 특정 테이블의 데이터를 조회, 추가하는 작업을 한다. 테스트 환경과 프로그램에 사용된 각종 파일들은 다음과 같다. 편의상 테스트 서버는 tux_machine이라는 호스트 네임을 가지고 있으며, Tibero와 Tuxedo는 각각 path/to/tibero와 /path/to/tuxedo에 설치되어 있다고 가정한다. 그리고 예제 프로그램 파일들이 작업 디렉터리 /path/to/example에 있다고 가정한다.

테스트 환경과 프로그램에 사용된 각종 파일들은 다음과 같다.

- 테스트 환경

구분	설명
운영체제	AIX
호스트 네임	tux_machine
셸	bash
Tibero 설치 홈 디렉터리	/path/to/tibero
Tuxedo 설치 홈 디렉터리	/path/to/tuxedo
예제 프로그램 홈 디렉터리	/path/to/example

- 프로그램 파일

파일	설명
tb_tux.env	시스템 환경변수 설정 파일
tb_tux.conf.m	Tuxedo 환경설정 파일
tmax32.fld	필드 테이블 파일
trans_fml32.tbc	서버 프로그램 tbESQL/C 파일
builds.sh	서버 프로그램 빌드 스크립트
insert.c	INSERT 클라이언트 프로그램 파일
select.c	SELECT 클라이언트 프로그램 파일
buildc.sh	클라이언트 프로그램 빌드 스크립트
create_table.sql	테스트를 위한 DB 테이블 생성 파일
run.sh	Tuxedo 시스템 기동 스크립트

참고

본 절에서 설명하고 있는 기본 프로그램의 전체 소스 코드는 “Appendix B. Tibero와 Tuxedo 연동 예제”를 참고한다.

다음에 제시된 순서대로 따르면 Tuxedo와 Tibero가 연동하는 것을 확인할 수 있다. 운영체제나 시스템 환경에 따라서 예제 파일을 수정하여 테스트해야 한다.

1. 시스템 환경변수 설정
2. Tuxedo 기본 환경설정
3. TMS 컴파일
4. 서버 프로그램 컴파일

5. 클라이언트 프로그램 컴파일

6. DB 테이블 생성

7. 예제 프로그램 실행

시스템 환경변수 설정

Tibero와 Tuxedo 연동 테스트를 하기 위해 필요한 시스템 환경변수 설정은 아래와 같다.

- **Tibero를 위한 기본 환경변수 설정**

```
export TB_HOME=/path/to/tibero
export TB_SID=tibero
export PATH=$TB_HOME/bin:$TB_HOME/client/bin:$TB_HOME/scripts:$PATH
export LD_LIBRARY_PATH=$TB_HOME/client/lib:$TB_HOME/lib:$LD_LIBRARY_PATH
export LIBPATH=$TB_HOME/client/lib:$TB_HOME/lib:$LIBPATH
```

- **Tuxedo를 위한 기본 환경변수 설정**

```
export TUXDIR=/path/to/tuxedo
export JAVA_HOME=$TUXDIR/jre
export JVMLIBS=$JAVA_HOME/lib/amd64/server:$JAVA_HOME/jre/bin
export PATH=$TUXDIR/bin:$JAVA_HOME/bin:$PATH
export COBCPY=:$TUXDIR/cobinclude; export COBCPY
export COBOPT="-C ANS85 -C ALIGN=8 -C NOIBMCOMP -C TRUNC=ANSI -C OSEXT=cbl"
export SHLIB_PATH=$TUXDIR/lib:$JVMLIBS:$SHLIB_PATH
export LIBPATH=$TUXDIR/lib:$JVMLIBS:$LIBPATH
export LD_LIBRARY_PATH=$TUXDIR/lib:$JVMLIBS:$LD_LIBRARY_PATH
export WEBJAVADIR=$TUXDIR/udataobj/webgui/java
```

- **Tuxedo를 위한 추가 환경변수 설정**

  연동 테스트를 위해서 추가 설정해주어야 할 환경변수들로서 상황에 맞게 설정한다.

```
export TUXCONFIG=/path/to/tuxedo/tuxconf
export FLDTBLDIR32=/path/to/tuxedo
export FIELDTBLS32=tmax32.fld
export TLOGDEVICE=/path/to/tuxedo/TLOG
export ULOGPFX=/path/to/tuxedo/ULOG
```

Tuxedo 기본 환경설정

Tuxedo 기본 환경설정은 아래와 같다.

- **Tuxedo 시스템 환경 파일 설정**

tb_tux.conf.m은 Tuxedo를 기동시킬 때 필요한 각종 정보들이 들어있는 Tuxedo 시스템 환경 파일이다. ASCII 파일 형태로 작성하며, tmloadcf 유틸리티로 컴파일하여 이진 파일을 생성한다. 생성된 이진 파일은 Tuxedo 기동 및 종료할 때 참조된다.

Tibero 서버와 연동하는 서비스를 활성화시키기 위하여 tb_tux.conf.m 환경 파일의 RESOURCES 절, MACHINES 절, GROUPS 절, SERVERS 절, SERVICES 절 항목을 아래와 같이 설정한다.

```
*RESOURCES
IPCKEY          68300
DOMAINID        tbrdomain
MASTER          tbrtest
MAXACCESSERS    10
MAXSERVERS      5
MAXSERVICES     10
MODEL           SHM
LDBAL           N

*MACHINES
DEFAULT:
                TUXDIR="path/to/tuxedo"
                APPDIR="path/to/example"
                TUXCONFIG="path/to/example/tuxconf"

tux_machine     LMID=tbrtest

*GROUPS
TBXA        LMID=tbrtest GRPNO=1
                TMSNAME=tms_tibero
                OPENINFO="TIBERO_XA:TIBERO_XA:user=sys,pwd=tibero,
                          sestm=60,db=tibero"

*SERVERS
DEFAULT:
                CLOPT="-A -r"

trans_fml32     SRVGRP=TBXA  SRVID=1

*SERVICES
SELECT_FML32
INSERT_FML32
~
```

RESOURCES 절과 MACHINE 절은 일반적인 Tuxedo 시스템 환경 파일 설정처럼 한다. MACHINE 절의 tux_machine은 서버의 호스트 네임이므로 테스트 환경에 따라 다르게 설정해야 한다. LMID와 MASTER는 임의로 tbrtest라고 정하였고, DOMAINID는 임의로 tbrdomain이라고 정하였으로 원하는 대로 설정한다.

GROUPS 절도 일반적인 Tuxedo 시스템 환경 파일의 설정과 동일하게 한다. TMSNAME에는 Tibero 서버와 XA 통신을 담당할 모듈의 이름을 설정한다. OPENINFO는 XA 모드 설정을 위한 'TIBERO_XA:TIBERO_XA:'를 앞쪽에 쓰고 그 뒤에 “4.2.2. xa_open 함수의 속성”이 나열되어야 한다.

SERVERS 절에는 예제 서버 프로그램인 trans_fml32가 설정되어 있다. SERVICE 절에는 예제 서버 프로그램이 제공하는 서비스인 SELECT_FML32, INSERT_FML32가 설정되어 있다.

참고

Tuxedo 시스템 환경 파일 설정에 대한 자세한 설명은 "Tuxedo Documentation"를 참고한다.

- **Tuxedo 시스템 환경 파일 컴파일**

  Tuxedo 시스템 환경 파일을 아래와 같은 명령으로 컴파일한다.

```
tmloadcf -y tb_tux.conf.m
```

- **필드 테이블 파일을 헤더 파일로 변환**

  서버나 클라언트 프로그램간에 데이터를 주고받을 때 공용 자료 구조체 정의해서 쓴다.

  이번 연동 테스트에서는 FML(Field Manipulation Language)을 이용하여 아래 같은 필드 테이블 파일 tmax32.fld을 새로 정의해서 쓴다.

```
#name          number          type            flag    comment
OUTPUT         302             string          0       -
EMPNO          901             long            0       -
ENAME          902             string          0       -
JOB            903             string          0       -
MGR            904             long            0       -
SAL            905             float           0       -
COMM           906             float           0       -
DEPTNO         907             long            0       -
```

  이 필드 테이블 파일 tmax32.fld로부터 헤더 파일을 생성하는 방법은 다음과 같다.

```
mkfldhdr32 tmax32.fld
```

  이 결과 tmax32.fld.h이라는 헤더 파일이 생성되며, 예제 클라이언트 프로그램과 서버 프로그램에서 가져다 쓴다. 그 밖의 다른 구조체를 정의하여 파일을 만드는 방법은 "Tuxedo Documentation"을 참고한다.

TMS 컴파일

다음 아래와 같은 명령어를 이용하여 Tibero용 TMS를 컴파일한다.

```
buildtms -o $TUXDIR/bin/tms_tibero -v -r TIBERO_XA
```

서버 프로그램 컴파일

실제적으로 서비스를 제공하는 서버 프로그램을 아래와 같은 빌드 스크립트 builds.sh를 이용하여 컴파일한다.

```
#### transaction server precompile ####
PRECOMP=$TB_HOME/client/bin/tbpc
PRECOMPFLAGS="UNSAFE_NULL=YES"
LIB=$TB_HOME/client/lib
INC=$TB_HOME/client/include
CFLAGS="-ltbcli -ltbxa -lm -lrt -lpthread -ltbertl -g "
CC=gcc
rm -f trans_fml32.c
$PRECOMP INCLUDE=$TUXDIR/include UNSAFE_NULL=YES INCLUDE=$INC
$PRECOMPFLAGS ONAME=trans_fml32.c trans_fml32.tbc

#### transaction server build ####
buildserver -o trans_fml32 -v -f trans_fml32.c -s INSERT_FML32,
SELECT_FML32 -r TIBERO_XA
```

참고

프리컴파일 옵션에 대한 자세한 설명은 "Tibero tbESQL/C 안내서"를 참고한다.

클라이언트 프로그램 컴파일

insert와 select 서비스를 요청하는 클라이언트 프로그램을 아래와 같은 빌드 스크립트 buildc.sh를 이용하여 컴파일한다.

```
buildclient -o insert -v -f insert.c
buildclient -o select -v -f select.c
```

DB 테이블 생성

Tibero서버에 tibero/tmax 계정으로 접속하여 아래와 같은 emp 테이블을 생성한다.

```
tbsqltibero/tmax

drop table emp;
CREATE TABLE emp (
    empno           NUMBER,
    ename           VARCHAR2(10),
    job             VARCHAR2(9),
```

```
    mgr             NUMBER(4),
    hiredate        DATE,
    sal             NUMBER(7,2),
    comm            NUMBER(7,2),
    deptno          NUMBER(2)
);
```

예제 프로그램 실행

- **Tuxedo 시스템 기동**

  다음 명령으로 기동한다.

```
tmboot -y
```

  성공적으로 기동되면 다음과 같은 메시지가 출력된다.

```
Booting all admin and server processes in /path/to/example/tuxconf
INFO: Oracle Tuxedo, Version 10.3.0.0, 64-bit, Patch Level (none)

Booting admin processes ...

exec BBL -A :
        process id=3457104 ... Started.

Booting server processes ...

exec tms_tibero -A :
        process id=4046910 ... Started.
exec tms_tibero -A :
        process id=9265576 ... Started.
exec tms_tibero -A :
        process id=1863802 ... Started.
exec trans_fml32 -A -r :
        process id=3719284 ... Started.
5 processes started.
```

- **클라이언트 프로그램 실행**

  emp 테이블에 Empyee 정보를 추가하고 조회하는 클라이언트 프로그램은 다음과 같이 실행한다.

```
./insert
****************************************
| Employee Number : 1
| Employee Name   : Kim
| Employee Job    : Manager
****************************************
```

insert 프로그램을 실행시키고 Employee Number, Employ Name, Employ Job를 위와 같이 입력하면 Tuxedo 서버 프로그램을 통하여 Tibero 서버의 emp 테이블에 레코드가 추가된다.

```
./select
****************************************
| Employee Number : 1
****************************************

EMPNO: 1
ENAME: Kim
JOB: Manager
```

select 프로그램을 실행시키고 Employee Number를 위와 같이 입력하면 Tuxedo 서버 프로그램을 통하여 Tibero 서버의 emp 테이블로부터 해당 레코드 정보를 가져와서 출력한다.

제5장 mod_tbPSM의 사용

본 장에서는 Apache HTTP 웹 서버를 통하여 tbPSM 프러시저 호출 및 HTML 페이지를 가져올 수 있는 tbPSM를 사용하는 방법을 기술한다.

5.1. 개요

Tibero에서는 웹 브라우저 등을 이용하여 HTTP 페이지를 요청하는 경우 tbPSM으로 작성된 프러시저를 호출하여 전송할 수 있는 mod_tbPSM이라는 Apache 모듈을 제공한다. Apache HTTP 서버는 구동할 때 Apache 설정 파일에 지정된 모듈 라이브러리들을 동적으로 로딩한다. mod_tbPSM도 같은 원리로 Apache HTTP 서버로부터 로딩되어 기능을 수행한다.

다음은 mod_tbPSM 모듈을 사용하기 위한 과정이다.

1. Apache HTTP 서버 설치
2. mod_tbPSM 등록
3. 프러시저 작성 및 실행

각 과정에 대한 자세한 설명은 해당 절의 내용을 참고한다.

5.2. Apache HTTP 서버 설치

mod_tbPSM에 최적화된 Apache HTTP 서버 버전은 UNIX 계열의 경우 2.2.31, Windows 계열의 경우 2.4.27이다.

Apache HTTP 서버에 대한 내용은 아래 주소를 참고한다.

```
UNIX: https://httpd.apache.org/download.cgi#apache22
Windows 64bit: https://www.apachelounge.com/download
Windows 32bit: https://www.apachelounge.com/download
```

UNIX 계열은 소스 코드를 받아서 직접 빌드해야 하고, Windows 계열은 PC에 설치된 VC에 맞는 버전을 선택하여 다운로드 한 뒤, 포함된 ReadMe.txt 파일에서 Install을 참고하여 설치 및 실행한다.

참고

Tibero Windows 64bit를 사용할 경우 Apache HTTP도 64bit를 사용해야 한다.

5.3. mod_tbPSM 등록

Apache HTTP 설치가 완료되면 설정 파일을 수정하여 mod_tbPSM을 등록시켜야 한다. Apache 설정 파일은 $(Apache 서버 설치 경로)/conf/httpd.conf 이다.

- UNIX 계열

  httpd.conf 파일에 아래와 같이 추가한다.

```
LoadModule tbpsm_module       "${TB_HOME}/client/lib/libmod_tbpsm.so"
<Location /tbpsm>
    SetHandler  tbpsm_handler
    AuthType Basic
    AuthName "Tibero"
</Location>
```

- Windows 계열

  Windows는 TB_HOME 환경변수를 사용하지 않으므로, 절대 경로를 지정해주거나 Apache 설치 경로의 module 폴더에 dll 파일을 복사하고 아래와 같이 추가한다.

```
LoadModule tbpsm_module       modules/libmod_tbpsm.dll
<Location /tbpsm>
    SetHandler  tbpsm_handler
    AuthType Basic
    AuthName "Tibero"
</Location>
```

[참고]

경로를 입력하는 경우 Windows 계열은 원래 역슬래시(\)로 디렉터리를 지정하지만, Apache에서는 슬래시(/)로 입력받는다.

```
LoadModule tbpsm_module       c:/tibero/client/lib/libmod_tbpsm.dll
<Location /tbpsm>
    SetHandler  tbpsm_handler
    AuthType Basic
    AuthName "Tibero"
</Location>
```

참고

Location 부분은 위의 예제와 똑같이 설정해야 한다.

5.4. 프러시저 작성 및 실행

본 절에서는 HTP 패키지를 이용하여 HTML 페이지를 리턴하는 프러시저를 작성하고 실행하는 방법을 설명한다.

참고

HTP 패키지 사용에 대한 자세한 내용은 "Tibero tbPSM 참조 안내서"를 참고한다.

5.4.1. 프러시저 생성

tbPSM 문법에 기반하여 HTP 패키지의 멤버 프러시저를 호출함으로써 HTML 소스 코드를 생성한다.

[예 5.1] HTP 패키지를 사용한 tbPSM 프러시저 작성 예제

```
CREATE OR REPLACE PROCEDURE hello (title in varchar2, text in varchar2)
AS
BEGIN
    HTP.HTMLOPEN;

    HTP.HEADOPEN;
    HTP.TITLE(title);
    HTP.HEADCLOSE;

    HTP.BODYOPEN;
    HTP.PRINT(text);
    HTP.BODYCLOSE;

    HTP.HTMLCLOSE;
END;
/
```

다음은 예제에 내용에 대한 간략한 설명이다.

- HTML의 HEAD 영역에서는 웹 브라우저 제목 표시줄에 title 파라미터를 통해 받은 문자열이 표시된다. 예를 들어 hello 프러시저의 첫 번째 인자로 'abc'를 지정하면 제목 표시줄에는 abc가 표시된다.
- HTML의 BODY 영역에서는 웹 브라우저 메인 화면에 text 파라미터를 통해 받은 문자열이 표시된다. 예를 들어 hello 프러시저의 두 번째 인자로 'def'를 지정하면 메인 화면에 def가 표시된다.

5.4.2. 실행

mod_tbPSM 모듈을 통해 프러시저를 실행하기 위한 URL 주소 형식은 다음과 같다.

```
http://{http_server_ip}:{http_server_port}/tbpsm/{tibero_dsn}/
{package_name}.{procedure_name}?{parameter1}={value1}&{parameter2}={value2)&...
```

항목	설명
http_server_ip	HTTP 서버의 IP 주소이다.
http_server_port	HTTP 서버의 포트 번호이다. 설정되지 않으면, 기본값 80이 적용된다.
tbpsm	httpd.conf 파일에 지정한 경로처럼 mod_tbPSM와 연동되기 위해서 반드시 넣어줘야하는 상대 경로이다.
tibero_dsn	tbdsn.tbr 파일에 명시된 DSN 이름이다. 프러시저가 저장되어있는 Tibero DBMS 서버를 가리킨다.
package_name	호출할 프러시저가 패키지의 멤버일 경우 패키지 이름 및 점(.)을 명시해야 한다.
procedure_name	호출할 프러시저의 이름이다.
parameter1, parameter2, ... , parametern	프러시저의 파라미터 이름이다. URL 주소 형식에서는 질의에 해당한다.
value1. value2, ... , valuen	프러시저 해당 파라미터에 들어가는 IN값이다.

예를 들어 "12.34.56.78" 주소를 가진 HTTP 서버에 접속하여 "test"라는 DSN을 가진 Tibero DBMS 서버에 있는 hello 프러시저(위에서 작성한 프러시저 예제 기반)을 호출하고 싶다면, 아래와 같이 URL 주소를 작성한다.

```
http://12.34.56.78/tbpsm/test/hello?title=abc&text=def
```

웹 브라우저에서 위와 같이 입력하고 요청하면 로그인 화면이 나타난다. 로그인 화면에 입력하는 아이디 및 비밀번호는 hello 프러시저 수행을 위해 로그인해야 하는 Tibero DBMS 계정의 아이디와 비밀번호와 동일하다.

5.5. 자동 로그인 기능 설정 방법

본 절에서는 URL 주소에 따라 자동 로그인할 수 있는 기능의 설정에 대해 설명한다.

일반적으로 mod_tbPSM와 연동되는 HTTP 서버에 웹 페이지 요청을 하면 로그인 화면이 나타나서 아이디 및 비밀번호를 직접 입력해야 한다. 자동 로그인 기능을 설정하면 아이디와 비밀번호을 저장해서 자동 로그인한다. 다만 모든 사용자가 해당 URL 주소로 접근이 가능해지기 때문에 보안상 주의가 필요하다.

mod_tbPSM에서는 mod.tbr 파일을 읽어들여 자동 로그인 기능을 지원한다. mod.tbr 파일은 TB_HOME/client/config 경로에 아래와 같은 형식으로 작성한다.

[예 5.2] mod.tbr 작성 예제

```
(LOCATION="/test")=(
    (USERNAME="tibero")
```

```
    (PASSWORD="tmax")
)

(LOCATION="/test2")=(
    (USERNAME="tibero")
    (PASSWORD="tmax")
)

(LOCATION="/test3")=(
    (USERNAME="sys")
    (PASSWORD="tibero")
)
```

다음은 mod.tbr 예제의 내용에 대한 간략한 설명이다.

항목	설명
LOCATION	URL 주소의 상대 경로이다.
USERNAME	접속할 DB 사용자 계정의 이름이다.
PASSWORD	접속할 DB 사용자 계정의 비밀번호이다.

'http://12.34.56.78/tbpsm/test/hello?title=abc&text=def'와 같이 접속한다고 했을 때, 'http://12.34.56.78/tbpsm 기준으로 상대 경로를 찾고, mod.tbr 파일에서 일치하는 LOCATION을 찾으면 설정된 'USERNAME'과 'PASSWORD'로 자동으로 로그인한다.

위의 예제처럼 여러 개의 접속 정보를 사용할 경우 tbdsn.tbr 파일도 DSN 접속 정보를 수정해야 한다.

[예 5.3] tbdsn.tbr 수정 예제

```
test=(
    (INSTANCE=(HOST=localhost)
              (PORT=8629)
              (DB_NAME=tibero)
    )
)

test2=(
    (INSTANCE=(HOST=localhost)
              (PORT=8629)
              (DB_NAME=tibero)
    )
)

test3=(
    (INSTANCE=(HOST=localhost)
              (PORT=8629)
```

```
            (DB_NAME=tibero)
    )
)
```

제6장 객체 타입의 사용

본 장에서는 Tibero의 객체 타입과 구성요소 동작방식에 대해서 설명한다.

6.1. 개요

Tibero 객체 타입은 사용자 정의 타입으로서 주문 또는 고객과 같은 실세계에 존재하는 개체를 표현하는데 사용된다. NUMBER, VARCHAR와 같은 기존 타입들을 사용하여 새로운 객체 타입을 정의할 수 있으며, 이전에 정의했던 객체 타입 혹은 컬렉션 타입도 타입을 정의하는데 이용할 수 있다.

또한 테이블을 생성할 때 컬럼의 타입으로 객체 타입을 명시하거나 해당 객체 타입 값만을 저장할 수 있도록 테이블을 생성할 수가 있다. 이러한 테이블을 **객체 테이블**이라고 한다.

6.2. 주요 개념

Tibero는 객체를 관계형 모델의 확장으로써 구현한다. 이를 위해 SQL 문법이 확장되었으며, PSM 역시 확장되었다.

6.2.1. 객체 타입

객체 타입은 데이터 타입의 한 종류이기 때문에 데이터 타입을 명시할 수 있는 대부분의 경우에 객체 타입 역시 사용이 가능하다. 테이블을 생성하는 경우 컬럼의 타입을 명시하는데 사용하거나 PSM을 정의하는 경우 변수의 타입을 명시하는 데에서도 사용할 수 있다. 타입을 정의하기 위해서 CREATE TYPE 구문 그리고 타입의 메소드가 존재할 경우 CREATE TYPE BODY 구문을 사용한다.

다음은 고객 타입을 정의하는 예이다.

[예 6.1] 객체 타입

```
CREATE TYPE customer_type AS OBJECT (
  custno NUMBER,
  name   VARCHAR2(40),
  phone  VARCHAR2(20),
  MEMBER FUNCTION tostring RETURN VARCHAR);
/

CREATE TYPE BODY customer_type AS
  MEMBER FUNCTION tostring RETURN VARCHAR IS
  BEGIN
    RETURN custno || ':' || name || ':' || phone;
```

```
  END;
END;
/
```

객체 타입을 구성하는 요소는 다음의 2가지로 구성된다.

타입	설명
속성	위의 예에서 custno, name, phone에 해당된다. 실제 데이터를 저장하기 위한 항목이다.
메소드	위의 예에서 tostring에 해당된다. 해당 타입 값에 대해 정해진 연산을 수행하기 위해 설정한다.

CREATE TYPE 문의 속성에 대해 제약 조건을 걸 수는 없지만, CREATE TABLE 문에서 타입을 사용할 경우에는 속성 값에 대한 제약 조건을 명시할 수 있다.

6.2.2. 객체 타입 값(객체)

객체 타입을 통해 실제 데이터를 가지는 객체가 존재할 수 있는데, 이것을 **객체 타입 값** 또는 간단히 **객체**라고 한다. 하나의 객체 타입을 통해 만들어질 수 있는 객체는 실제 들어가는 데이터에 따라 여러 개가 존재할 수 있다.

다음은 테이블을 생성할 때 컬럼의 타입을 객체 타입으로 지정하고, 이러한 테이블에 실제 객체(객체 타입 값)를 저장하는 예제이다.

[예 6.2] 객체 타입 값

```
CREATE TABLE customer (
  cust_obj customer_type,
  status   VARCHAR(1));

INSERT INTO customer VALUES (customer_type(1, 'James', '111-222-3333'), 'V');
```

위 예에서 customer 테이블에 이름이 'James'인 customer_type 객체 타입의 객체가 하나 저장되었다. customer_type(...) 표현식은 디폴트 생성자라 부르는 것으로, 해당 객체 타입의 객체를 하나 만들 때 이용할 수 있는 문법 중 하나이다.

6.2.3. 테이블 생성에 객체 타입 사용

테이블에서 객체를 저장하는 형태는 테이블을 생성할 때 결정되며, 다음의 두 가지로 나눌 수 있다.

- 일반 테이블의 컬럼 형태

  보통의 CREATE TABLE 문법에서 컬럼의 타입을 지정하는 부분에 객체 타입을 명시하면 테이블의 컬럼 형태로 객체가 저장된다. 이러한 객체를 컬럼 객체라고 한다.

- 객체 테이블 형태

  하나의 객체 타입의 객체들만을 저장할 수 있는 테이블을 만들 수도 있는데, 이러한 테이블을 객체 테이블이라고 한다. 이 때 생성되는 객체를 행 객체라고 한다.

다음은 객체 테이블을 생성하는 CREATE TABLE 문의 예제이다.

[예 6.3] 객체 테이블 생성

```
CREATE TABLE customer_type_tab of customer_type;
```

객체 테이블을 생성하면 객체 타입의 속성에 해당하는 컬럼이 해당 테이블의 컬럼으로 생성된다. 그러므로 위의 예에서 customer_type_tab 테이블은 사용자 입장에서 3개의 컬럼을 가지는 테이블로 조회된다.

[예 6.4] 객체 테이블 조회 (1)

```
SQL> desc customer_type_tab;

COLUMN_NAME                                TYPE               CONSTRAINT
------------------------------------------ ------------------ --------------------
CUSTNO                                     NUMBER
NAME                                       VARCHAR(40)
PHONE                                      VARCHAR(20)
```

객체 테이블은 객체를 저장하고 있으므로, 실제 객체를 테이블에서 꺼내올 수도 있게 되는데, 이때 사용하는 문법이 VALUE(x) 표현식이다.

다음과 같이 객체 테이블 별칭(x)을 인자로 표현식에 주어서 객체 테이블에 저장되어 있는 객체에 접근할 수가 있다.

[예 6.5] 객체 테이블 조회 (2)

```
INSERT INTO customer_type_tab VALUES(customer_type(1, 'James', '111-222-3333'));
INSERT INTO customer_type_tab VALUES(2, 'Bob', '444-555-6666');

DECLARE
  c1 customer_type;
BEGIN
  SELECT VALUE(x) into c1 FROM customer_type_tab x where custno = 1;
  dbms_output.put_line(c1.tostring());
END;
/
```

위의 예에서 보는 것처럼 객체 테이블에 Insert를 수행할 때는 객체 표현식(위에서는 디폴트 생성자를 사용했다) 하나만을 명시해도 Insert가 가능하고, 각각의 속성 값을 일일이 지정하는 형태로도 Insert가 가능함을 알 수가 있다.

6.2.4. 컬렉션 사용

같은 타입의 데이터를 모아서 하나의 값으로 만들어 처리하는 경우 사용하는 것이 컬렉션 타입이다. Tibero에서 컬렉션 타입은 객체 타입과 마찬가지로 사용자가 직접 정의하는 타입이며, 가변 배열 혹은 중첩 테이블의 두 가지 형태로 제공된다. 컬렉션 타입 역시 사용자가 새로운 객체 타입을 생성할 때 속성의 타입으로서 명시될 수 있다. 자세한 내용은 "제7장 컬렉션 타입의 사용"을 참고한다.

다음의 예는 중첩 테이블 타입을 하나 정의하고 이러한 타입 속성을 가지는 객체 타입을 정의하는 CREATE TYPE 문법의 예를 보여준다.

```
CREATE TYPE customer_ntab_type AS TABLE OF customer_type;
/

CREATE TYPE int_group_type AS OBJECT (
  group_no NUMBER,
  members  customer_ntab_type);
/
```

6.3. 객체 구성요소

본 절에서는 Tibero 객체를 구성하는 여러 가지 요소에 해서 설명한다.

6.3.1. SQL 내에서의 객체 사용

본 절에서는 SQL 내에서 사용되는 객체의 구성요소에 대해서 설명한다.

6.3.1.1. NULL 객체와 객체의 NULL 속성

객체의 값 자체가 NULL일 경우 쪼갤 수 없는 객체 혹은 NULL 객체라고 한다. 쪼갤 수 없는 객체는 모든 속성값이 NULL인 객체와는 다른 객체이다(즉, 이 경우는 NULL이 아닌 객체이다). NULL이 아닌 객체의 경우 특정 속성의 값은 NULL을 가질 수도 있으며 이 속성값은 프러시저 혹은 함수를 통해 NULL이 아닌 값으로 변경될 수 있다. 그러나 NULL 객체인 경우 객체의 속성값을 바꾸는 것은 불가능하며, 객체의 메소드를 호출하는 것도 불가능하다.

다음은 컬럼 객체를 가지는 테이블의 객체 컬럼에 NULL 객체와 NULL 속성값을 가지는 NULL이 아닌 객체를 Insert하는 예제이다.

[예 6.6] NULL 객체와 객체의 NULL 속성

```
CREATE TYPE customer_type2 AS OBJECT (
  custno NUMBER,
  name   VARCHAR2(40),
  phone  VARCHAR2(20),
  MEMBER FUNCTION tostring RETURN VARCHAR DETERMINISTIC);
/
```

```
CREATE TYPE BODY customer_type2 AS
  MEMBER FUNCTION tostring RETURN VARCHAR DETERMINISTIC IS
  BEGIN
    RETURN custno || ':' || name || ':' || phone;
  END;
END;
/

CREATE TABLE customer2 (
  cust_obj customer_type2,
  status   VARCHAR(1));

INSERT INTO customer2 VALUES (Null, 'I');
INSERT INTO customer2 VALUES (customer_type2 (NULL, NULL, NULL), 'V');
```

주의

객체 테이블의 행 객체는 NULL 객체일 수가 없다. 또한 컬렉션의 경우 NULL 컬렉션과 구성 요소가 하나도 들어있지 않은 빈 컬렉션은 NULL 컬렉션이 아니며 서로 다른 컬렉션이다.

6.3.1.2. 테이블 제약 조건

일반 테이블에 제약 조건을 정의할 수 있는 것처럼 객체 타입을 사용한 테이블에도 제약 조건을 정의할 수 있다. 다만 제약 조건은 객체 타입의 포함 관계 트리에서 맨 아래에 존재하는 속성에 대해서만 정의할 수 있다. 맨 아래의 속성은 객체 타입 속성일 수 없으므로, 이러한 속성은 내장 타입을 가지게 된다.

다음은 객체 테이블을 생성할 때 객체 타입의 속성 하나를 객체 테이블의 기본 키로 지정하는 예이다.

[예 6.7] 객체 테이블 생성에 객체 테이블을 기본 키로 지정

```
CREATE TABLE customer_type_tab2 of customer_type2 (
  custno PRIMARY KEY);
```

테이블이 객체 컬럼을 가질 때에도 해당 객체의 속성에 제약 조건을 정의할 수 있는데, 이 때는 점(.) 표기법으로 정의할 수 있다.

[예 6.8] 객체 컬럼의 제약 조건 설정

```
CREATE TABLE customer3 (
  cust_obj customer_type2,
  status   VARCHAR(1),
  CONSTRAINT cust_check_cons
  CHECK (cust_obj.custno IS NOT NULL));
```

6.3.1.3. 인덱스

인덱스 컬럼이 될 수 있는 조건은 제약 조건을 정의할 수 있는 컬럼의 조건과 동일하다. 객체 타입의 포함 관계 트리에서 맨 아래에 존재하는 속성만을 인덱스 컬럼으로 지정할 수 있다. 객체 컬럼의 경우 점(.) 표기법으로 정의할 수 있다.

[예 6.9] 객체 컬럼의 인텍스 생성

```
CREATE INDEX cust_idx_1 ON customer3 (cust_obj.custno);
```

함수 기반 인덱스를 생성하는 경우 함수의 반환 타입이 객체 타입일 경우는 생성이 허용되지 않는다. 만일 객체 테이블에 대해 함수 기반 인덱스를 정의하면서 기반 객체 타입의 함수 멤버 메소드를 호출하는 경우 다음과 같이 테이블 별칭 다음 점(.) 표기법으로 정의할 수 있다.

[예 6.10] 객체 테이블에 함수 기반 인덱스 생성

```
CREATE INDEX cust_idx_2 ON customer_type_tab2 x (x.tostring());
```

6.3.1.4. 속성 이름의 대상 찾기

테이블의 컬럼이 객체 타입인 경우 이 객체의 속성 값을 Select하는 경우 점(.) 표기법을 사용한다. SELECT 문에서 점(.) 표기법을 사용할 때 한 가지 주의할 점은 객체 컬럼에 대해 점(.) 표기법을 사용할 때 항상 테이블 별칭을 점(.) 표기법과 앞에 표기해 주어야 한다.

[예 6.11] 테이블의 컬럼이 객체 타입인 경우 Select

```
SELECT cust_obj.phone FROM customer3; -- 구문 에러
SELECT x.cust_obj.phone FROM customer3 x; -- 올바른 문장
```

6.3.1.5. VALUE(x) 표현식

VALUE(x) 표현식은 SQL에서 객체 테이블의 테이블 별칭을 인자로 받아 현재 행에 해당하는 행 객체를 반환하는 표현식이다.

[예 6.12] VALUE(x) 표현식

```
INSERT INTO customer_type_tab2 values (1, 'David', '555-666-7777');
DECLARE
  c1 customer_type2;
BEGIN
  SELECT VALUE(x) into c1 FROM customer_type_tab2 x where custno = 1;
  dbms_output.put_line(c1.tostring());
END;
/
```

위의 예처럼 객체 테이블의 속성값을 조회하는 것이 아니고 객체의 멤버 메소드를 호출하는 경우 경우에는 VALUE(x) 표현식이 필요하다. VALUE(x) 표현식은 다음과 같이 UPDATE 문을 수행하는 경우 사용자 정의 생성자(아래에 설명)가 생성한 객체로 객체 테이블의 현재 객체를 덮어쓰는 경우에도 사용할 수 있다.

[예 6.13] VALUE(x) 표현식을 사용한 update

```
CREATE OR REPLACE TYPE customer_type3 AS OBJECT (
  custno NUMBER,
  name   VARCHAR2(40),
  phone  VARCHAR2(20),
  CONSTRUCTOR FUNCTION customer_type3 RETURN SELF AS RESULT);
/

CREATE OR REPLACE TYPE BODY customer_type3 AS
  CONSTRUCTOR FUNCTION customer_type3 RETURN SELF AS RESULT IS
  BEGIN
    custno := -1;
    name := 'Untitled';
    phone := 'N/A';
    RETURN;
  END;
END;
/

CREATE TABLE customer_type_tab3 OF customer_type3;

INSERT INTO customer_type_tab3 values (1, 'David', '555-666-7777');

UPDATE customer_type_tab3 x SET VALUE(x) = customer_type3()
WHERE custno = 1;
```

6.3.2. 객체 타입 메소드

객체 타입의 메소드는 PSM 함수 혹은 프러시저일 수 있으며 해당 객체에 대해 수행할 연산을 명시한다.

- **멤버 메소드**

  해당 객체에 대한 연산을 정의할 때 일반적으로 사용하는 메소드 형태이다. 연산을 적용할 객체를 명시할 때 점(.)을 사용한 특별한 문법을 이용해서 멤버 메소드를 호출할 수 있다.

- **정적 메소드**

  PSM 패키지 내의 함수 혹은 프러시저를 정의하는 것과 유사하게 객체 타입을 정의할 때 타입 내부에 정의하는 함수 혹은 프러시저이다. 정적 메소드는 객체 타입 내부에 존재한다는 것 빼고는 일반 패키지 함수 혹은 프러시저와 다른 점은 없다.

- **생성자 메소드**

  특정 연산을 수행한 결과를 통해서 해당 타입의 객체를 생성하고 싶을 때 이용할 수 있다. 사용자는 이러한 생성자 메소드를 이용해서 새로운 객체를 생성할 수도 있고, 따로 생성자를 정의하지 않았을 경우에도 디폴트 생성자 메소드 문법을 통해 객체를 생성할 수가 있다.

다음은 객체의 멤버 메소드를 호출하는 예이다.

```
SELECT x.cust_obj.tostring() FROM customer x;
```

6.3.2.1. 멤버 메소드

해당 함수 혹은 프러시저가 현재 주어진 객체의 값에 따른 연산을 수행할 때 멤버 메소드를 통해 이러한 연산을 정의한다. CREATE TYPE 문 내에서 멤버 메소드를 정의할 때는 MEMBER FUNCTION 또는 MEMBER PROCEDURE와 같이 앞에 MEMBER를 명시한다. 이렇게 정의된 멤버 메소드는 다음과 같이 객체 표현식 다음에 점(.) 표기법을 사용하여 호출할 수 있다.

[예 6.14] 멤버 메소드 호출 (1)

```
dbms_output.put_line(my_obj_var.obj_memfunc1());
```

점 앞에 오는 것은 해당 객체의 값을 가질 수 있는 객체 표현식이고, 점 다음에는 그 객체의 객체 타입에서 정의하고 있는 멤버 메소드의 이름을 명시한다 SQL에서 멤버 메소드를 호출할 때는 메소드에 건네줄 매개변수가 없는 경우에도 다음과 같이 항상 괄호를 붙여 주어야 한다.

[예 6.15] 멤버 메소드 호출 (2)

```
SELECT x.cust_obj.tostring() FROM customer2 x;
```

멤버 메소드와 SELF 매개변수

멤버 메소드에는 첫 번째 매개변수로서 정해진 하나의 매개변수가 항상 전달되는 것으로 간주되는데, 이 매개변수를 SELF고 한다. 이 SELF 매개변수에는 멤버 메소드를 호출할 때 명시한 대상 객체의 값이 매개변수의 값으로 전달된다.

멤버 메소드를 정의할 때 첫 번째 매개변수로 이 SELF 매개변수를 직접 명시할 수 있으나, 명시하지 않아도 명시한 것으로 간주하며 문법 에러는 발생하지 않는다. 타입의 BODY에서 대상 객체의 속성 값을 명시할 때 SELF 및 점(.) 표기법을 이용해 속성 값을 명시할 수도 있고, SELF와 점을 생략해도 명시가 가능하다. (매개 변수와 이름이 겹치지 않을 경우)

[예 6.16] SELF 매개변수 생성

```
CREATE OR REPLACE TYPE rectangle_type AS OBJECT (
  x NUMBER,
  y NUMBER,
  MEMBER FUNCTION around RETURN NUMBER,
  MEMBER FUNCTION area(SELF IN OUT NOCOPY rectangle_type) RETURN NUMBER);
/

CREATE OR REPLACE TYPE BODY rectangle_type AS
  MEMBER FUNCTION around RETURN NUMBER IS
  BEGIN
    RETURN 2 * (x + y);
```

```
  END;
  MEMBER FUNCTION area RETURN NUMBER IS
  BEGIN
    RETURN SELF.x * SELF.y;
  END;
END;
/
```

객체 테이블에 저장되어 있는 객체에 대한 멤버 메소드를 호출하는 경우 테이블 별칭과 점(.) 표기법을 이용하여 다음과 같이 호출할 수가 있다. 또한 행 객체를 나타내는 VALUE(x) 표현식으로도 가능하다.

[예 6.17] 멤버 메소드 호출

```
CREATE TABLE rect_tab of rectangle_type;
INSERT INTO rect_tab VALUES(2, 4);
INSERT INTO rect_tab VALUES(rectangle_type(8, 3));
SELECT x.around(), x.area() FROM rect_tab x;
SELECT VALUE(x).around(), VALUE(x).area() FROM rect_tab x;
```

멤버 메소드와 객체 비교

객체를 비교하거나 또는 정렬하려면 일단 두 객체의 기반이 되는 객체 타입이 같아야 하며, 정해진 형태의 멤버 메소드를 통해서 비교 연산이 수행된다. 만일 이러한 멤버 메소드가 존재하지 않을 경우 에러가 발생한다(단, 동등 비교(equal) 및 비동등 비교(inequal)에 대해서는 특수한 규칙을 통해 허용된다).

비교를 수행하는 멤버 메소드는 두 가지(map 메소드/order 메소드) 형태로 존재하는데, 하나의 객체 타입에 대해서 이 둘 중 하나만을 정의할 수 있다.

- **map 메소드**

  하나의 내장 타입 값을 반환하는 메소드이다. 이 값을 기준으로 객체를 비교하거나, 정렬하는 데 이용할 수 있다. 객체 표현식에 대해 SQL에서 ORDER BY를 구사하면, 자동적으로 이 map 메소드가 각 객체에 대해 호출된다. map 메소드가 정의되어 있을 때 다음의 SQL에서 컬럼 c1과 컬럼 c2가 해당 객체 타입일 경우 위 SQL 은 실제로 다음과 같이 수행된다.

  ```
  SELECT 1 FROM t1 x WHERE x.c1 > x.c2;
  ```

  ```
  SELECT 1 FROM t1 x WHERE x.c1.map() > x.c2.map();
  ```

  다음은 비교 연산을 위해 해당 객체 타입에 map 메소드를 정의하는 예이다.

  [예 6.18] 객체 타입에 map 메소드 정의

  ```
  CREATE OR REPLACE TYPE rectangle_type AS OBJECT (
    x NUMBER,
    y NUMBER,
    MAP MEMBER FUNCTION area RETURN NUMBER);
  ```

```
/

CREATE OR REPLACE TYPE BODY rectangle_type AS
  MAP MEMBER FUNCTION area RETURN NUMBER IS
  BEGIN
    RETURN SELF.x * SELF.y;
  END;
END;
/
```

- **order 메소드**

  비교할 두 개의 객체를 매개변수로 받아서 비교 결과를 반환하는 메소드이다. 이 중 첫 번째 객체는 SELF를 통해 전달되므로, SELF와 비교할 나머지 객체만을 추가 매개변수로 전달하면 된다. order 메소드는 비교 연산자를 통한 객체의 비교에서는 자동적으로 호출되나, ORDER BY에서는 호출되지 않는다.

  order 메소드의 반환값은 음수, 0, 양수의 숫자형 값을 반환하여야 하며 이 값들은 각각 SELF 객체가 비교하는 객체보다 작거나 또는 같거나 또는 더 큼을 의미한다.

  [예 6.19] order 메소드

```
CREATE OR REPLACE TYPE address_type AS OBJECT (
  name  VARCHAR(20),
  addr  VARCHAR(50),
  city  VARCHAR(20),
  state VARCHAR(2),
  zip   VARCHAR(5),
  ORDER MEMBER FUNCTION cmp(v address_type) RETURN NUMBER);
/

CREATE OR REPLACE TYPE BODY address_type AS
  ORDER MEMBER FUNCTION cmp(v address_type) RETURN NUMBER IS
  BEGIN
    IF SELF.zip < v.zip THEN RETURN -1;
    ELSIF SELF.zip > v.zip THEN RETURN 1;
    ELSIF state < v.state THEN RETURN -1;
    ELSIF state > v.state THEN RETURN 1;
    ELSIF city < v.city THEN RETURN -1;
    ELSIF city > v.city THEN RETURN 1;
    ELSIF addr < v.addr THEN RETURN -1;
    ELSIF addr > v.addr THEN RETURN 1;
    ELSE RETURN 0;
    END IF;
  END;
END;
/
```

만일 map 메소드도 order 메소드도 존재하지 않는데 동등 비교(equal) 또는 비동등 비교(inequal) 연산을 수행할 경우 이 비교는 객체 타입의 포함 관계 트리에서 맨 아래에 존재하는 속성(즉, 내장 타입 속성) 각각에 대해 비교를 수행하고, 그 결과들을 AND 연산(비동등 비교일 때 OR 연산)으로 묶은 것과 같은 것으로 간주되어 정상 수행된다. 즉, 만일 위의 address_type에서 order 메소드가 정의되지 않았다고 했을 때 다음의 SQL이 수행되면 이 SQL은 실제로 다음과 같이 수행된다.

[예 6.20] 메소드 비교

```
SELECT 1 FROM t1 x WHERE x.c1 = x.c2;
```

```
SELECT 1 FROM t1 x
WHERE x.c1.name = x.c2.name AND
      x.c1.addr = x.c2.addr AND
      x.c1.city = x.c2.city AND
      x.c1.state = x.c2.state AND
      x.c1.zip = x.c2.zip;
```

6.3.2.2. 정적 메소드

정적 메소드는 특정 객체에 대해 수행할 동작을 정의하는 것이 아닌 단순히 객체 타입 내에 정의되는 함수 또는 프러시저이다. 그러므로 정적 메소드에서는 미리 정의된 SELF 매개변수가 존재하지 않는다. 정적 메소드를 호출하기 위해서는 메소드 이름 앞에 점(.)을 사용하여 객체 타입 이름을 선행한다.

[예 6.21] 정적 메소드

```
my_obj_type.my_static_method1();
```

6.3.2.3. 생성자 메소드

생성자 메소드는 수행의 결과로 해당 타입의 객체를 반환하는 함수이다. 이것은 미리 정의되어 있는 디폴트 생성자와 사용자가 정의하는 사용자 정의 생성자로 나눌 수 있다.

- **디폴트 생성자**

  디폴트 생성자는 생성자의 매개변수로 해당 객체 타입의 속성 값을 순서대로 명시하여 호출하는 생성자이다. 예를 들어 위의 address_type에 대한 디폴트 생성자를 호출하는 예는 다음과 같다.

  [예 6.22] 디폴트 생성자

  ```
  address_type("Bob", "123 Oak Street", "Los Angeles", "CA", "90143")
  ```

- **사용자 정의 생성자 메소드**

  사용자 정의 생성자 메소드는 사용자가 해당 타입의 객체 생성을 위해 객체 타입 내에 직접 정의하는 메소드 함수이다. 사용자 정의 생성자를 정의할 때 어떤 매개변수를 전달할 지는 사용자가 자유롭게 정할 수 있다. 만일 생성자 내에서 특정 속성에 대해 아무런 값도 지정하지 않았다면, 해당 속성의 값은 NULL이 된다.

사용자 정의 생성자 메소드의 정의는 다음의 예와 같이 CONSTRUCTOR FUNCTION으로 시작하여 RETURN SELF AS RESULT로 끝난다. 생성자의 이름은 객체 타입의 이름과 일치해야 한다.

[예 6.23] 사용자 정의 생성자 메소드

```
CREATE OR REPLACE TYPE rectangle_type AS OBJECT (
  x NUMBER,
  y NUMBER,
  CONSTRUCTOR FUNCTION rectangle_type (
    SELF IN OUT NOCOPY rectangle_type, n NUMBER)
    RETURN SELF AS RESULT);
/

CREATE OR REPLACE TYPE BODY rectangle_type AS
  CONSTRUCTOR FUNCTION rectangle_type(
    SELF IN OUT NOCOPY rectangle_type, n NUMBER)
    RETURN SELF AS RESULT IS
  BEGIN
    SELF.x := n;
    y := n;
    RETURN;
  END;
END;
/
```

사용자 정의 생성자 메소드의 첫 번째 매개변수는 멤버 메소드와 마찬가지로 SELF 를 명시할 수 있으며 명시하지 않아도 명시한 것으로 간주된다. 메소드의 BODY 부분에서는 이 SELF 객체 변수의 값을 결정하고, 함수가 종료될 경우에는 단일 RETURN 문(RETURN 문 뒤에 함수가 반환할 값이 오지 않는)으로 메소드를 종료하여야만 한다. 사용자 정의 생성자가 생성하는 객체의 값은 이 SELF 객체 변수의 값이 된다.

6.4. 객체의 동작

본 절에서는 Tibero 객체의 실제 동작과 관련한 몇 가지 내용을 설명한다.

6.4.1. 객체의 저장

객체는 속성들의 집합으로 구성되며, 각각의 속성 역시 객체가 될 수 있기 때문에(단, 순환 참조는 허용하지 않는다) 객체 타입을 정의하고 나면 이것은 객체 간의 포함 관계를 나타내는 하나의 Tree로 표현이 가능해진다. Tree에서 단말에 위치하는 속성은 객체 타입 속성일 수 없으며 내장 타입을 가지는 속성이 된다.

객체가 테이블에 저장될 때는 한 객체가 하나의 값으로 합쳐져서 저장되지 않고 Tree에서 단말에 위치하는 내장 타입 속성들의 각 값으로 찢어져서 저장된다. 다시 말해, 객체가 저장되어 있는 테이블의 하나의

컬럼이 객체 자체에 대한 컬럼이 될 수는 없으며 NUMBER, VARCHAR2와 같은 내장 타입 컬럼과 가변 배열 컬럼들로 구성된 여러 개의 컬럼들을 사용해 해당 객체를 저장하게 된다.

테이블을 생성하는 경우 사용자가 명시한 컬럼의 타입이 객체 타입일 경우 Tibero는 내부적으로 이런 내장 타입 컬럼들을 숨겨진 컬럼의 형태로 만들고 추가로 컬럼을 하나 더 추가하게 되는데, 이 컬럼(NULL 지시자 컬럼)에는 해당 객체를 구성하는 객체들 중(객체의 속성 값 또한 임의의 객체 타입의 객체가 될 수 있으므로) 누가 NULL이고 누가 NULL이 아닌지에 대한 정보를 담게 된다. 그러나 단말 속성(내장 타입 속성)이 NULL 여부에 대한 정보는 NULL 지시자 컬럼에 들어가지 않고, 나머지 (내장 타입에 대한) 숨겨진 컬럼을 사용해 이 정보를 저장한다.

객체 테이블의 경우 객체 테이블이 기반하는 객체 타입을 구성하는 각각의 속성에 대해서 객체 테이블의 컬럼을 생성하며 컬럼의 이름은 속성의 이름을 따라간다. 기반 객체 타입의 특정 속성이 객체 타입 속성일 경우 위에서 기술한 대로(즉, 일반 테이블에 사용자가 객체 타입의 컬럼을 명시했을 때와 같이) 숨겨진 컬럼들이 내부적으로 추가된다.

객체 테이블에서 VALUE(x) 표현식을 사용하여 객체 테이블의 객체에 접근할 경우 이 객체는 객체의 포함 관계 Tree 맨 아래에 있는 객체부터 디폴트 생성자를 사용하여 객체들을 생성하고, 마지막으로 가장 상위 단계의 객체들 그리고 내장 타입 속성 값들을 모아 최종 객체 타입의 디폴트 생성자를 호출해서 객체 테이블의 객체를 생성한다.

주의

이때 디폴트 생성자가 사용자에 의해 가려졌을 경우(매개변수의 타입과 갯수, 순서가 디폴트 생성자와 동일한 생성자)라 하더라도 여전히 시스템에서 정의한 디폴트 생성자로 객체를 생성한다.

6.4.1.1. 객체 테이블과 객체 식별자(Object Identifier; OID)

객체 식별자를 이용해 객체 테이블에 저장되는 객체를 유일하게 식별할 수 있다. 객체 식별자는 객체 테이블을 생성하는 경우 **자동생성형**과 **기본키 기반형** 중 하나를 선택할 수가 있다.

- **자동생성형**

  따로 지정하지 않으면 자동생성형 객체 식별자가 객체 테이블의 숨겨진 컬럼(OID 컬럼)으로서 할당이 된다. 객체 테이블에 객체를 하나 Insert할 때마다, 분산 환경까지 고려해서 전역적으로 유일한 16Bytes 값이 생성되며 이것이 OID 컬럼의 값으로서 저장이 된다. 이것은 CREATE TABLE을 수행할 때 OID 컬럼의 DEFAULT 값이 SYS_GUID() 내장 함수로 선언된 것과 동일한 동작을 한다.

- **기본키 기반형**

  객체 테이블을 만들 때 기본키를 지정한다면 해당 객체 테이블의 객체 식별자로서 사용할 수가 있다. CREATE TABLE 문에 OBJECT IDENTIFIER IS PRIMARY KEY 절을 추가하면서 생성하는 객체 테이블은 자동생성형 OID 컬럼을 만들지 않고 기본키의 값을 객체 식별자의 값으로 사용한다. 기본키 기반형 객체 식별자는 해당 테이블의 기본키 값을 기반으로 하고 있으므로, 유일성은 해당 객체 테이블 내에서만 보장된다. 만일 기본키의 길이가 16Bytes보다 작은 경우 이것은 공간의 절약을 의미한다.

6.4.1.2. 가변 배열(VARRAY)의 저장

VARRAY가 테이블의 컬럼으로 저장될 때는 실제로도 하나의 컬럼으로 저장된다. VARRAY 타입 값의 최대 길이는 요소 타입의 최대 길이를 VARRAY 타입의 최대 요소 갯수로 곱한 값에 약간의 오버헤드를 추가한 값이다.

해당 VARRAY 타입 값의 가능한 최대 길이가 VARCHAR2의 최대 길이를 넘을 경우 이 VARRAY 컬럼을 가지는 테이블을 생성할 때 긴 길이의 VARRAY 값을 저장할 수 있도록 하는 LOB 세그먼트도 같이 생성이 된다. 하지만 실제 값을 저장할 때는 LOB 세그먼트의 존재 유무와는 관계 없이 각각의 값 별로 VARCHAR2의 최대 길이보다 작을 경우는 그대로 테이블에 저장되고, 그렇지 않을 경우는 LOB 세그먼트에 저장된다.

6.4.2. 객체 생성자

다음은 객체 생성자의 주요 동작방식에 대한 설명이다.

- **생성자 가리기**

  만일 사용자가 디폴트 생성자와 똑같이 생긴(매개변수의 타입과 갯수, 순서가 디폴트 생성자와 똑같은) 생성자를 정의했을 경우 디폴트 생성자는 사용자에 의해 호출될 수 없으며 사용자가 정의한 생성자에 의해 가려진다. 사용자 정의 생성자는 서브타입으로 상속되지 않기 때문에 서브타입에서 정의하는 생성자로 가려지지 않는다.

- **생성자 오버로딩**

  생성자도 일반 타입 메소드와 마찬가지로 오버로딩(메소드 이름이 같지만 매개변수의 타입과 갯수, 순서가 다른 메소드)할 수 있다. 이것은 특정 객체 타입의 객체를 생성하기 위한 생성자가 동시에 여러 개 존재할 수 있다는 의미이다.

- **생성자의 사용**

  생성자는 표현식의 일종으로 일반 컬럼 및 함수가 사용될 수 있는 곳이라면 생성자도 사용이 가능하다. SQL에서 생성자를 호출하기 위해서는 생성자의 인자가 없더라도 항상 괄호를 붙여 주어야 한다. 생성자를 정의할 때는 SELF를 명시할 수 있으나, 호출할 때는 SELF에 해당하는 매개변수를 명시할 수 없다. CREATE TABLE 또는 ALTER TABLE의 DEFAULT 절 그리고 CHECK 제약 조건에는 사용자 정의 생성자가 올 수 없으며 디폴트 생성자만 올 수 있다.

제7장 컬렉션 타입의 사용

본 장에서는 Tibero의 컬렉션 타입과 사용방법에 대해서 설명한다.

7.1. 개요

컬렉션 타입은 같은 타입의 값들을 모아 두기 위해 정의할 수 있는 사용자 정의 타입의 한 종류이다.

다음의 두 가지 형태 중 하나로 존재한다.

- **가변 배열**은 순서가 있는 같은 타입의 요소들의 모음이다.
- **중첩 테이블**은 갯수의 제한이 없는 순서가 정해지지 않은 같은 타입의 요소들의 모음이다.

참고

현재 중첩 테이블은 PSM 내에서만 사용이 가능하다.

7.1.1. 컬렉션 타입 생성

CREATE TYPE 문을 사용해 컬렉션 타입을 생성한다.

가변 배열 타입을 생성하기 위해서는 다음과 같이 AS VARRAY를 명시한다. VARRAY 다음에 오는 괄호 안에 이 가변 배열이 담을 수 있는 요소 갯수의 최대값을 입력한다. OF 뒤에 오는 요소 타입으로는 내장 타입 또는 사용자 정의 타입의 이름을 명시할 수 있다.

[예 7.1] 컬렉션 타입 생성

```
CREATE TYPE str_varr_type AS VARRAY(10000) OF VARCHAR(100);
/
```

컬렉션 타입 생성은 객체 타입 생성과 마찬가지로 실제 컬렉션을 저장하기 위한 공간을 할당하는 것이 아니며 컬렉션의 모양만을 기술할 뿐이다. LOB 타입 그리고 XMLType들에 대한 가변 배열은 생성할 수 없다. 컬렉션 타입은 테이블의 컬럼 타입, 객체 타입의 속성 타입, PSM 내에서의 변수 및 매개변수 그리고 반환값의 타입으로써 사용될 수 있다. 객체 테이블의 타입으로는 사용이 할 수 없다.

7.1.2. 컬렉션(컬렉션 값) 생성

컬렉션(컬렉션 값)을 생성하기 위해서는 컬렉션 타입의 이름 다음에 괄호 안에 해당 컬렉션 타입의 요소들을 콤마(,)와 함께 나열하여 생성할 수 있다.

[예 7.2] 컬렉션(컬렉션 값) 생성

```
str_varr_type('ABC', 'DEFG', 'HH')
str_varr_type()
```

괄호 안에 어떤 요소 값도 주지 않은 채로 컬렉션을 생성할 때 이것을 **빈 컬렉션**이라고 한다. 빈 컬렉션은 컬렉션에 참여하는 요소만 없을 뿐 NULL은 아니다.

7.1.3. 다층 컬렉션 타입

컬렉션 타입의 요소의 타입이 또 다른 컬렉션 타입이거나, 요소의 타입이 객체 타입이고 이 객체 타입의 요소 중 하나가 컬렉션 타입일 경우 이것을 다층 컬렉션 타입이라고 한다. **현재 SQL 쿼리에서는 가변 배열로만 다층 컬렉션 타입의 구성이 가능하며 중첩 테이블과 가변 배열을 섞어서 다층 컬렉션 타입을 구성할 수는 없다.**

컬렉션 타입을 사용할 수 있는 곳이라면 다층 컬렉션 타입도 사용이 가능하다. 다층 컬렉션(컬렉션 값)을 생성할 때도 일반 컬렉션과 마찬가지 방식으로 요소들을 명시하는데, 결국 이것은 다음과 같이 컬렉션 타입 이름을 중첩해서 명시한다.

[예 7.3] 다층 컬렉션 타입

```
CREATE OR REPLACE TYPE str_varr_coll_type AS VARRAY(1000) OF str_varr_type;
/

CREATE TABLE nested_coll_tbl (id number, coll_val str_varr_coll_type);

INSERT INTO nested_coll_tbl VALUES (1,
  str_varr_coll_type(str_varr_type('AB', 'CD'), str_varr_type()));
INSERT INTO nested_coll_tbl VALUES (2,
  str_varr_coll_type(str_varr_type(), str_varr_type('EF', 'GH')));
```

7.2. 사용 예제

본 절에서는 컬렉션 타입을 사용하는 경우에 따른 사용법을 설명한다.

7.2.1. 쿼리에서 사용

현재 SELECT의 결과 컬럼을 컬렉션으로 하여 컬렉션 내용을 클라이언트로 전달하는 기능은 구현되어 있지 않다. 그러나 TABLE() 표현식을 사용해서 컬렉션을 풀어헤치면 SELECT 쿼리에서도 컬렉션의 내용을 조회할 수가 있다. TABLE() 표현식은 FROM 절에 사용하여 컬렉션의 각 요소를 행으로 바꾼 테이블처럼 사용할 수 있도록 하는 표현식이다.

[예 7.4] 쿼리에서의 컬렉션 타입 사용

```
CREATE TABLE coll_tbl (id number, coll_val str_varr_type);

INSERT INTO coll_tbl VALUES (1, str_varr_type('AB', 'CD'));
INSERT INTO coll_tbl VALUES (2, str_varr_type('EF', 'GH'));
INSERT INTO coll_tbl VALUES (3, str_varr_type());

SQL> SELECT id, d.* FROM coll_tbl c, TABLE(c.coll_val) d;

        ID COLUMN_VALUE
---------- ------------
         1 AB
         1 CD
         2 EF
         2 GH

4 rows selected.
```

TABLE() 표현식의 인자로 공급되는 컬럼은 FROM 절에서 자신의 왼쪽에 있는 테이블의 컬럼 중 컬렉션 타입의 컬럼을 명시할 수 있다(이를 위해 일반적으로 위와 같이 테이블 별칭을 사용한다). 위에서 보는 것처럼 내장 타입 혹은 컬렉션에 대한 컬렉션일 경우 TABLE() 표현식으로 인해 만들어지는 테이블은 COLUMN_VALUE라고 하는 하나의 컬럼만을 가진다.

TABLE() 표현식에 대해서는 다음과 같이 외부 조인 연산자인 (+)로 외부 조인을 수행시킬 수 있다.

[예 7.5] 컬렉션 타입의 외부 조인 Select

```
SQL> SELECT id, d.* FROM coll_tbl c, TABLE(c.coll_val)(+) d;

        ID COLUMN_VALUE
---------- ------------
         1 AB
         1 CD
         2 EF
         2 GH
         3

5 rows selected.
```

외부 조인을 수행하지 않았을 경우에는 빈 컬렉션에 해당하는 로우가 Select되지 않지만, 외부 조인을 수행하였을 경우에는 빈 컬렉션에 해당하는 로우도 출력되는 것을 볼 수가 있다.

서브 쿼리의 결과가 하나의 컬렉션 값을 반환하는 스칼라 서브 쿼리일 경우 이 서브 쿼리를 TABLE() 표현식의 인자로 주어서 컬렉션 내용을 풀어헤쳐 조회해 볼 수 있다. 즉, 서브 쿼리는 일반적인 값 표현식에 올 수 있는 스칼라 서브 쿼리만 허용된다.

[예 7.6] 서브 쿼리 결과가 컬렉션인 경우

```
SQL> SELECT * FROM TABLE(SELECT coll_val FROM coll_tbl WHERE id = 1);

COLUMN_VALUE
------------
AB
CD

2 rows selected.
```

위 예제의 서브쿼리에서 WHERE 절이 존재하지 않고 서브쿼리의 결과 로우의 갯수가 2개 이상일 경우는, 위 TABLE 쿼리는 정상 수행되지 않고 런타임 에러를 발생시킨다.

다층 컬렉션의 경우 TABLE() 표현식을 반복해서 사용함으로써 다층 컬렉션의 임의의 계층에 들어있는 요소들도 추출이 가능하다. 이 때 상위 계층의 TABLE() 표현식이 만들어 내는 컬렉션 컬럼을 그 다음 계층의 TABLE() 표현식에 명시하기 위해서는 테이블 별칭이 필요하다.

[예 7.7] 다층 컬렉션

```
SQL> SELECT id, z.* FROM nested_coll_tbl x, TABLE(x.coll_val) y,
                         TABLE(y.COLUMN_VALUE) z;

        ID COLUMN_VALUE
---------- ------------
         1 AB
         1 CD
         2 EF
         2 GH

4 rows selected.
```

컬렉션이 객체에 대한 컬렉션일 경우 TABLE() 표현식이 반환하는 테이블은 객체 테이블이 되며 객체 테이블에서 구사할 수 있는 각종 표현식을 똑같이 구사할 수 있다. 또한 이 테이블은 COLUMN_VALUE 컬럼을 가지지 않으며 객체 테이블의 기반 객체 타입의 속성 이름과 같은 이름의 컬럼들을 가진다.

[예 7.8] 컬렉션이 객체에 대한 컬렉션

```
CREATE TYPE customer_type2 AS OBJECT (
  custno NUMBER,
  name   VARCHAR2(40),
  phone  VARCHAR2(20),
  MEMBER FUNCTION tostring RETURN VARCHAR);
/

CREATE TYPE BODY customer_type2 AS
```

```
  MEMBER FUNCTION tostring RETURN VARCHAR IS
  BEGIN
    RETURN custno || ':' || name || ':' || phone;
  END;
END;
/

CREATE OR REPLACE TYPE cust_coll_type AS VARRAY(400) OF customer_type2;
/

CREATE TABLE cust_coll_tab (
  id NUMBER,
  coll_val cust_coll_type);

INSERT INTO cust_coll_tab VALUES (1,
  cust_coll_type(customer_type2(1, 'Bob', '222-333-4444'),
                 customer_type2(2, 'Alice', '444-555-6666')));

SQL> SELECT id, d.*, VALUE(d).tostring() str FROM
       cust_coll_tab c, TABLE(c.coll_val) d;

        ID     CUSTNO NAME  PHONE        STR
---------- ---------- ----- ------------ ------------------------------
         1          1 Bob   222-333-4444 1:Bob:222-333-4444
         1          2 Alice 444-555-6666 2:Alice:444-555-6666

2 rows selected.
```

7.2.2. DML에서 사용

현재 중첩 테이블에 대해서는 DML이 지원되지 않고 가변 배열에 대해서는 가변 배열값을 통으로 Insert하거나 Update하는 것은 지원된다. 쿼리 내에서 가변 배열의 요소를 조작하는 것은 지원되지 않는다.

Appendix A. tbJDBC 예제

본 장에서는 tbJDBC를 이용하여 작성한 기본 프로그램의 전체 소스 코드를 설명한다.

A.1. JdbcTest.class

다음은 tbJDBC를 이용하여 **JdbcTest 클래스 파일**을 작성한 프로그램 소스 코드이다.

```
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Types;

public class JdbcTest
{
    Connection conn;

    public static void main(String[] args) throws Exception
    {
        JdbcTest test = new JdbcTest();

        test.connect();
        test.executeStatement();
        test.executePreparedStatement();
        test.executeCallableStatement();
        test.disconnect();
    }

    private void connect() throws ClassNotFoundException
    {
        Class.forName("com.tmax.tibero.jdbc.TbDriver");

        try {
            conn = DriverManager.getConnection(
                    "jdbc:tibero:thin:@localhost:8629:tibero", "tibero", "tmax");
        } catch (SQLException e) {
            System.out.println("connection failure!");
            System.exit(-1);
```

```
    }
    System.out.println("Connection success!");
}

private void executeStatement() throws SQLException
{
    String dropTable   = "drop table emp";
    String createTable = "create table emp (id   number, "+
                         " name varchar(20), salary number)";
    String InsertTable = "insert into emp values(1000, 'Park', 5000)";

    Statement stmt = conn.createStatement();

    try {
        stmt.executeUpdate(dropTable);
     } catch(SQLException e) {
       // if there is not the table
     }

     stmt.executeUpdate(createTable);
     stmt.executeUpdate(InsertTable);

     stmt.close();
}

private void executePreparedStatement() throws SQLException
{
    PreparedStatement pstmt = conn
    .prepareStatement("select name from emp where id = ?");

    pstmt.setString(1, "1000");

    ResultSet rs = pstmt.executeQuery();

    while (rs.next()) {
        System.out.println(rs.getString(1));
    }
    pstmt.close();
}

private void executeCallableStatement() throws SQLException
{
    String callSQL = " CREATE PROCEDURE testProc "+
                     " (ID_VAL IN NUMBER, SAL_VAL IN OUT NUMBER) as " +
                     " BEGIN" +
                     "   update emp" +
                     "      set salary = SAL_VAL" +
```

```
                        "    where id = ID_VAL;" +
                        "   select salary into SAL_VAL" +
                        "     from emp" +
                        "    where id = ID_VAl;" +
                        " END;";

        String dropProc = "DROP PROCEDURE testProc";

        Statement stmt = conn.createStatement();

        try {
            stmt.executeUpdate(dropProc);
        } catch(SQLException e) {
            // if there is not the procedure
        }

        stmt.executeUpdate(callSQL);

        CallableStatement cstmt = conn.prepareCall("{call testProc(?, ?)}");
        cstmt.setInt(1, 1000);
        cstmt.setInt(2, 7000);
        cstmt.registerOutParameter(2, Types.INTEGER);
        cstmt.executeUpdate();

        int salary = cstmt.getInt(2);
        System.out.println(salary);

        stmt.close();
        cstmt.close();
    }

    private void disconnect() throws SQLException
    {
        if (conn != null)
            conn.close();
    }
}
```

Appendix B. Tibero와 Tuxedo 연동 예제

본 장에서는 Tibero와 Tuxedo 연동 예제 프로그램의 전체 소스 코드와 각종 스크립트를 설명한다.

B.1. tb_tux.env

다음은 시스템 환경변수 설정 파일이다.

```
# for tibero
export TB_HOME=/path/to/tibero
export TB_SID=tibero
export PATH=$TB_HOME/bin:$TB_HOME/client/bin:$TB_HOME/scripts:$PATH
export LD_LIBRARY_PATH=$TB_HOME/client/lib:$TB_HOME/lib:$LD_LIBRARY_PATH
export LIBPATH=$TB_HOME/client/lib:$TB_HOME/lib:$LIBPATH

# for tuxedo
export TUXDIR=/path/to/tuxedo
export JAVA_HOME=$TUXDIR/jre
export JVMLIBS=$JAVA_HOME/lib/amd64/server:$JAVA_HOME/jre/bin
export PATH=$TUXDIR/bin:$JAVA_HOME/bin:$PATH
export COBCPY=:$TUXDIR/cobinclude; export COBCPY
export COBOPT="-C ANS85 -C ALIGN=8 -C NOIBMCOMP -C TRUNC=ANSI -C OSEXT=cbl"
export SHLIB_PATH=$TUXDIR/lib:$JVMLIBS:$SHLIB_PATH
export LIBPATH=$TUXDIR/lib:$JVMLIBS:$LIBPATH
export LD_LIBRARY_PATH=$TUXDIR/lib:$JVMLIBS:$LD_LIBRARY_PATH
export WEBJAVADIR=$TUXDIR/udataobj/webgui/java

export TUXCONFIG=/path/to/tuxedo/tuxconf
export FLDTBLDIR32=/path/to/tuxedo
export FIELDTBLS32=tmax32.fld
export TLOGDEVICE=/path/to/tuxedo/TLOG
export ULOGPFX=/path/to/tuxedo/ULOG
```

B.2. tb_tux.conf.m

다음은 Tuxedo 환경설정 파일이다.

```
*RESOURCES
IPCKEY          68300

DOMAINID        tbrdomain
```

```
MASTER          tbrtest
MAXACCESSERS    10
MAXSERVERS      5
MAXSERVICES     20
MODEL           SHM
LDBAL           N

*MACHINES
DEFAULT:
                TUXDIR="/data1/apmqam/oracle/tuxedo/tuxedo10gR3"
                APPDIR="/data1/apmqam/tibero_tuxedo_test"
                TUXCONFIG="/data1/apmqam/tibero_tuxedo_test/tuxconf"
                TLOGDEVICE="/data1/apmqam/tibero_tuxedo_test/TLOG"

tmaxi4          LMID=tbrtest

*GROUPS
TBXA            LMID=tbrtest GRPNO=1
                TMSNAME=tms_tibero
                OPENINFO="TIBERO_XA:TIBERO_XA:user=sys,pwd=tibero,
                          sestm=60,db=tibero"

*SERVERS
DEFAULT:
                CLOPT="-A -r"

trans_fml32     SRVGRP=TBXA  SRVID=1

*SERVICES
SELECT_FML32
INSERT_FML32
```

B.3. tmax32.fld

다음은 필드 테이블 파일이다.

```
#name          number          type            flag    comment
OUTPUT         302             string          0       -
EMPNO          901             long            0       -
ENAME          902             string          0       -
JOB            903             string          0       -
MGR            904             long            0       -
SAL            905             float           0       -
COMM           906             float           0       -
DEPTNO         907             long            0       -
```

B.4. trans_fml32.tbc

다음은 서버 프로그램 tbESQL/C 파일이다.

```
#include <stdio.h>
#include <atmi.h>
#include <userlog.h>
#include <Uunix.h>
#include <fml32.h>
#include "fml32.fld.h"
#include <tx.h>
#include "sqlca.h"

EXEC SQL include SQLCA.H;

EXEC SQL begin declare section;
    int h_empno;
    char h_ename[10];
    char h_job[10];
EXEC SQL end declare section;

void INSERT_FML32(rqst)
TPSVCINFO *rqst;
{
    FBFR32 *sndbuf;
    char msgbuf[256];
    FLDLEN32 flen;
    XID *xid;
    TXINFO info;
    char xidstring[1000];
    char str[100];

    int i;

    sndbuf = (FBFR32 *)rqst->data;

    tx_info(&info);
    xid = &(info.xid);

    memset( xidstring, 0x00, 1000);

    for( i = 0 ; i < xid->gtrid_length+xid->bqual_length ; i++ )
    {
        sprintf(xidstring+strlen(xidstring),"%0x\0",xid->data[i]);
    }

    Fprint32(sndbuf);
```

```
    memset( &h_empno, 0x00, sizeof ( h_empno ) );
    memset( h_ename, 0x00, sizeof ( h_ename ) );
    memset( h_job, 0x00, sizeof ( h_job ) );

    Fget32(sndbuf, EMPNO, 0, (char *)&h_empno, &flen);
    Fget32(sndbuf, ENAME, 0, (char *)h_ename, &flen);
    Fget32(sndbuf, JOB, 0, (char *)h_job, &flen);

    printf("SVR: EMPNO %d ENAME %s JOB %s\n", h_empno, h_ename, h_job);
    printf("SVR: INSERT_FML32 XID:%d.%d. %0x.%s - %s\n\n",
            xid->gtrid_length, xid->bqual_length,
            xid->formatID, xidstring, h_ename);

    EXEC SQL INSERT
    INTO emp( empno, ename, job )
    VALUES ( :h_empno, :h_ename, :h_job );

    if ( sqlca.sqlcode != 0 ){
        sprintf(msgbuf, "insert fail: sqlcode = %d(%s)\n",
                        sqlca.sqlcode, sqlca.sqlerrm.sqlerrmc);
        Fchg32(sndbuf, OUTPUT, 0, msgbuf, 0);
        tpreturn(TPFAIL, -1, (char *)sndbuf, 0, 0);
    }

    strcpy(msgbuf, "insert success!" );
    Fchg32(sndbuf, OUTPUT, 0, msgbuf, 0);
    tpreturn(TPSUCCESS, 0, rqst->data, strlen(rqst->data), 0);
}

void SELECT_FML32(rqst)
TPSVCINFO *rqst;
{
    FBFR32 *sndbuf;
    char msgbuf[256];
    FLDLEN32 flen;

    sndbuf = (FBFR32 *)rqst->data;
    Fprint32(sndbuf);
    Fget32(sndbuf, EMPNO, 0, (char *)&h_empno, &flen);

    EXEC SQL SELECT NVL(ename,' '), NVL(job,' ')
    INTO :h_ename,:h_job
    FROM emp
    WHERE empno = :h_empno;

    if ( sqlca.sqlcode != 0 ){
        sprintf(msgbuf, "select failed sqlcode = %d(%s)\n",
```

```
                        sqlca.sqlcode, sqlca.sqlerrm.sqlerrmc);
        Fchg32(sndbuf, OUTPUT, 0, msgbuf, 0);
        tpreturn(TPFAIL, -1, (char *)sndbuf, 0, 0);
    }

    printf("#### %d \n", h_empno);
    Fchg32(sndbuf, ENAME, 0,(char *)h_ename, 0);
    Fchg32(sndbuf, JOB, 0, (char *)h_job, 0);
    Fchg32(sndbuf, EMPNO, 0, (char *)&h_empno, 0);

    strcpy(msgbuf, "select success!" );
    Fchg32(sndbuf, OUTPUT, 0, msgbuf, 0);

    Fprint32(sndbuf);
    tpreturn(TPSUCCESS, 0,(char *)sndbuf, sizeof(sndbuf), 0);
}
```

B.5. builds.sh

다음은 서버 프로그램 빌드 스크립트이다.

```
#### transaction server precompile ####
PRECOMP=$TB_HOME/client/bin/tbpc
PRECOMPFLAGS="UNSAFE_NULL=YES"
LIB=$TB_HOME/client/lib
INC=$TB_HOME/client/include
CFLAGS="-ltbcli -ltbxa -lm -lrt -lpthread -ltbertl -g "
CC=gcc
rm -f trans_fml32.c
$PRECOMP INCLUDE=$TUXDIR/include UNSAFE_NULL=YES INCLUDE=$INC
$PRECOMPFLAGS ONAME=trans_fml32.c trans_fml32.tbc

#### transaction server build ####
buildserver -o trans_fml32 -v -f trans_fml32.c -s INSERT_FML32,SELECT_FML32 -r
TIBERO_XA
```

B.6. insert.c

다음은 INSERT 클라이언트 프로그램 파일이다.

```
#include <stdio.h>
#include <atmi.h>
#include <fml32.h>
#include <string.h>
#include "fml32.fld.h"
```

```
struct temp_str{
        int   empno;
        char  ename[11];
        char  job[10];
};
typedef struct temp_str *str;

main(argc, argv)
char *argv[];
{
    FBFR32 *sendbuf;
    FBFR32 *recvbuf;
    long recvlen = 0;
    int ret;
    str tmpbuf;
    char msgbuf[30];
    FLDLEN32 flen;

    tmpbuf = malloc(sizeof(struct temp_str));

    if (tpinit((TPINIT *) NULL) == -1) {
        fprintf(stderr, "Tpinit failed\n");
        exit(1);
    }

    if((sendbuf = (FBFR32 *) tpalloc("FML32", NULL, 0)) == NULL) {
        fprintf(stderr,"Error allocating send buffer\n");
        tpterm();
        exit(1);
    }

    if((recvbuf = (FBFR32 *) tpalloc("FML32", NULL, 0)) == NULL) {
        fprintf(stderr,"Error allocating receive buffer\n");
        tpfree((char *)sendbuf);
        tpterm();
        exit(1);
    }

    tmpbuf = malloc(sizeof(struct temp_str));

    printf("\n****************************************\n");
    printf( "|  Employee Number : " ); scanf ( "%d", &tmpbuf->empno );
    printf( "|  Employee Name   : " ); scanf ( "%s", tmpbuf->ename );
    printf( "|  Employee Job    : " ); scanf ( "%s", tmpbuf->job );
    printf("****************************************\n\n");
```

```
    tpbegin(10, 0);
    Fchg32(sendbuf,EMPNO,0,(char *)&tmpbuf->empno,0);
    Fchg32(sendbuf,ENAME,0,(char *)tmpbuf->ename,0);
    Fchg32(sendbuf,JOB,0,(char *)tmpbuf->job,0);

    ret = tpcall("INSERT_FML32", (char *)sendbuf, 0, (char **)&recvbuf,
                &recvlen, (long)0);
    if(ret == -1) {
        fprintf(stderr, "tperrno = %d (%s) \n", tperrno, tpstrerror(tperrno));
        tpfree((char *)sendbuf);
        tpfree((char *)recvbuf);
        tpterm();
        exit(1);
    }

    flen = sizeof( msgbuf );
    Fget32(recvbuf, OUTPUT, 0, (char *)msgbuf, &flen);
    printf("%s\n", msgbuf);

    tpcommit(0);

    free(tmpbuf);
    tpfree((char *)sendbuf);
    tpterm();
}
```

B.7. select.c

다음은 SELECT 클라이언트 프로그램 파일이다.

```
#include <stdio.h>
#include <atmi.h>
#include <fml32.h>
#include <string.h>
#include "fml32.fld.h"


main(argc, argv)
char *argv[];
{
    FBFR32 *sendbuf;
    FBFR32 *recvbuf;
    long recvlen;
    int ret;
    int empno;
    FLDLEN32 flen;
```

```
int h_empno;
char h_ename[10];
char h_job[10];
char msgbuf[256];

if (tpinit((TPINIT *) NULL) == -1) {
    fprintf(stderr, "Tpinit failed\n");
    exit(1);
}

if((sendbuf = (FBFR32 *) tpalloc("FML32", NULL, 0)) == NULL) {
    fprintf(stderr,"Error allocating send buffer\n");
    tpterm();
    exit(1);
}

if((recvbuf = (FBFR32 *) tpalloc("FML32", NULL, 0)) == NULL) {
    fprintf(stderr,"Error allocating receive buffer\n");
    tpfree((char *)sendbuf);
    tpterm();
    exit(1);
}

printf("\n*****************************************\n");
printf( "|  Employee Number : " ); scanf ( "%d", &empno );
printf("*****************************************\n\n");

Fchg32(sendbuf,EMPNO,0,(char *)&empno,0);

ret = tpcall("SELECT_FML32", (char *)sendbuf, 0, (char **)&recvbuf,
             &recvlen, (long)0);
if(ret == -1) {
    fprintf(stderr, "tperrno = %d (%s)\n", tperrno, tpstrerror(tperrno));
    flen = sizeof( msgbuf );
    Fget32(recvbuf, OUTPUT, 0, (char *)msgbuf, &flen);
    printf("error msg: %s\n", msgbuf);
    tpfree((char *)sendbuf);
    tpfree((char *)recvbuf);
    tpterm();
    exit(1);
}

flen = sizeof(msgbuf);
Fget32(recvbuf, OUTPUT, 0, (char *)msgbuf, &flen);
printf("%s\n", msgbuf);

flen = sizeof(&h_empno);
```

```
    Fget32(recvbuf, EMPNO, 0, (char *)&h_empno, &flen);
    printf("EMPNO; %d \n", h_empno);
    flen = sizeof(h_ename);
    Fget32(recvbuf, ENAME, 0, (char *)h_ename, &flen);
    printf("ENAME: %s \n", h_ename);
    flen = sizeof(h_job);
    Fget32(recvbuf, JOB, 0, (char *)h_job, &flen);
    printf("JOB: %s \n", h_job);

    tpfree((char *)recvbuf);
    tpfree((char *)sendbuf);
    tpterm();
}
~
```

B.8. buildc.sh

다음은 클라이언트 프로그램 빌드 스크립트이다.

```
buildclient -o insert -v -f insert.c
buildclient -o select -v -f select.c
```

B.9. create_table.sql

다음은 SELECT 클라이언트 프로그램 파일이다.

```
drop table emp;
CREATE TABLE emp (
    empno           NUMBER,
    ename           VARCHAR2(10),
    job             VARCHAR2(9),
    mgr             NUMBER(4),
    hiredate        DATE,
    sal             NUMBER(7,2),
    comm            NUMBER(7,2),
    deptno          NUMBER(2)
);
~
```

B.10. run.sh

다음은 Tuxedo 시스템 기동 스크립트이다.

```
#!/bin/sh

tmshutdown -y
rm ULOG* > /dev/null 2>&1
rm xa* > /dev/null 2>&1

tmloadcf -y tb_tux.conf.m
rm /data1/apmqam/tibero_tuxedo_test/TLOG* > /dev/null 2>&1
rm /data1/apmqam/tibero_tuxedo_test/ULOG* > /dev/null 2>&1

tmadmin -c << EOF
crdl -z /data1/apmqam/tibero_tuxedo_test/TLOG -b 1000
q
EOF

tmboot -y
```

색인

ㅇ

ㅈ

ㅋ

ㅌ

ㅎ