DCL Database Connectivity

Version 2.0

2005.04.27

김대중<daejung@sysdeveloper.net>
http://www.sysdeveloper.net/daejung

요약

DDBC(DCL Database Connectivity)는 Microsoft Windows와 GNU/Linux를 비롯한 UNIX 운영체제의 DBMS들 사이에서 이식성과 객체지향 방법론을 통한 개발의 생산성 및 데이터베이스 서버 연결에 관한 성능의 극대화를 목표로 개발된 데이터베이스 드라이버 시스템으로 데이터베이스 클라이언트 어플리케이션을 서로 다른 DBMS 접속 환경으로부터 분리해 준다.

ANSI SQL2(SQL92)에서 정의한 대부분의 RDBMS(관계형 데이터베이스) 관련 데이터 타입을 지원하며, 이들 데이터 타입에 유연하게 접근할 수 있도록 해 준다.

본 문서의 1장은 DDBC의 구성요소에 대하여, 2장은 DDBC에서 사용하고 있는 에러코드와 지원하는 SQL 데이터타입, 그리고 인터페이스에 대한 개괄적인 내용을 설명하고 있고, 3장은 드라이버의 설치와 관련한 드라이버 메니저에 대한 내용을 요약하고 있다. 4장은 어플리케이션 개발을 위한 DCL 클래스들에 대한 자세한 참조 매뉴얼을 기술하고 있다. 마지막으로 5장에서는 DDBC 2.0 개발과 함께 개발된 Informix, InterBase, MySQL, Oracle 드라이버에 대한 주요내용을 요약하고 있다.

어플리케이션 개발자는 본 문서만을 통해서도 충분히 개발할 수 있으나 드라이버 개발자를 위한 더 자세한 내용은 소스 코드를 참고 하여야 한다.

- 목 차 -

제1장 DDBC 개요 4
1.1 역사 4
1.2 개발 배경 및 목적 5
1.3 구성요소 5
1.4 객체기반 인터페이스 모델 5
1.5 스레드 모델 7
제2장 인터페이스 레퍼런스 7
2.1 SQL::Error 7
2.2 SQL::DataType 7
2.3 SQL::Field 12
2.4 SQL::Param 13
2.5 SQL::Query 14
2.6 SQL::Connection 15
2.7 SQL::DRIVER_MODULE 15
제3장 드라이버 메니저 16
3.1 SQLDriver 16
3.2 SQLDriverPool 16
3.3 드라이버의 설치 17
제4장 어플리케이션의 개발 19
4.1 SQLConnection 19
SQLConnection::SQLConnection 19
SQLConnection::open 20
SQLConnection::close 20
SQLConnection::execute 20
SQLConnection::startTrans 20
SQLConnection::commitTrans 20
SQLConnection::rollbackTrans 21
SQLConnection::serverInfo 21
SQLConnection::canTransact 21
SQLConnection::driver 21
SQLConnection::connected 21
SQLConnection::inTransaction 21
4.2 SQLConnectionPool 21
SQLConnectionPool::SQLConnectionPool 22
SQLConnectionPool::getConnection 22
SQLConnectionPool::release 23
SQLConnectionPool::setMaxCount 23

SQLConnectionPool::count 23
SQLConnectionPool::idleCount 23
SQLConnectionPool::clear 23
4.3 SQLQuery 23
SQLQuery::SQLQuery 23
SQLQuery::prepare 24
SQLQuery::execute 24
SQLQuery::fetch 25
SQLQuery::eof 25
SQLQuery::affectedRows 25
SQLQuery::fields 25
SQLQuery::params 26
4.4 SQLField 26
SQLField::name 26
SQLField::dataType 26
SQLField::dataTypeName 26
SQLField::serverDataTypeName 26
SQLField::isNull 26
SQLField::dataSize 27
SQLField::dataSizeMax 27
SQLField::getData 27
SQLField::getValue 27
SQLField::asXXXX 28
4.5 SQLParam 29
SQLParam::setNull 29
SQLParam::setData 29
SQLParam::setDataType 29
SQLParam::setData 29
4.6 SQLFields 31
SQLFields::operator[] 31
SQLFields::byName 31
SQLFields::count 31
4.7 SQLParams 31
SQLParams:operator[] 31
SQLParams::byName 31
SQLParams::count 32
제5장 Informix 32
5.1 개요 32
5.2 연결문자열 32

5.3 데이터 타입 32
5.4 트랜잭션 33
5.5 EXECUTE PROCEDURE 33
제6장 InterBase 34
6.1 개요 34
6.2 연결문자열 34
6.3 데이터 타입 35
6.4 트랜잭션 35
6.5 CREATE DATABASE 35
6.6 EXECUTE PROCEDURE 36
제7장 MySQL 36
7.1 개요 36
7.2 연결문자열 37
7.3 데이터 타입 37
7.4 트랜잭션 38
7.5 파라미터의 구현 38
제8장 Oracle 38
8.1 개요 38
8.2 연결문자열 39
8.3 데이터 타입 39
8.4 트랜잭션 39
8.5 PLSQL과 RETURNING 절 39

제1장 DDBC 개요

1.1 역사

DCL[1]에 데이터베이스 연결을 지원하기 위해 개발된 DDBC는 C/C++ 프로그래밍 언어를 기반으로 하여 2001년 말에 처음 시작 되었다. 초기의 인터페이스는 InterBase, Oracle의 클라이언트 API를 분석하면서 시작 되었고 2002년 4월에 IDBC[2] 1.0으로 릴리즈 했다. 2003년 8월의 3.0 버전에는 MySQL과 Informix 드라이버 추가 되었으며 2004년 2월 DCL의 전체 라이브러리 시스템을 재조직 하면서 DDBC 1.2으로 정리하였다.

DDBC 2.0버전은 2004년 12월에 기존 버전이 다양한 데이터 타입을 명확히 구분하여 지원하는데 부족한 점이 있고 필드 및 파라미터의 데이터에 접근하는데 있어서 유연성이 떨어지는 것으로 판단되어 이들을 재설계하여 관련 코드를 재작성 하였다.

---

[1] Daejung Class Library, Microsoft Windows와 UNIX 사이에서의 소스레벨 이식성을 목표로 개발된 C++ 클래스 라이브러리

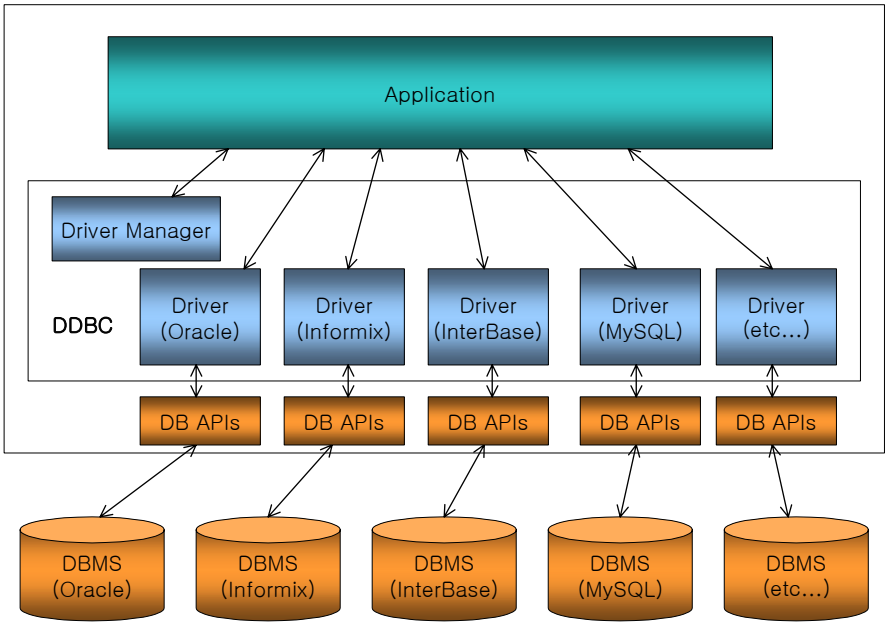
[2] DCL의 초기 프로젝트 명칭은 IFC(ITOS Foundation Class Library)이었다.

1.2 개발 배경 및 목적

DDBC 개발의 주요 목적은 이식성과 성능, 그리고 응용프로그램개발의 생산성 향상에 있다. 일반적으로 데이터베이스 드라이버 시스템은 운영체제 의존적인 경우가 많으며 특히 Microsoft Windows에 그 개발 도구들이 집중되어 있는 현실이다. 이러한 이유 때문에 동일한 동작을 하는 데이터베이스 응용프로그램은 운영체제 바뀌게 되면 처음부터 코드를 재작성 해야만 한다. DCL의 목적이 그러하듯 DDBC도 가장 중요한 목적은 이식성이 있다.

1.3 구성요소

DDBC는 DDBC 인터페이스, 드라이버, 그리고 DBAPI로 구성된다. DDBC 인터페이스는 C++의 추상 클래스로 명세 되어 있으며 응용프로그램이 특정 DBAPI에 의존되는 것을 제거한다. DDBC 드라이버는 이들 인터페이스를 구현하기 위해 DBMS의 DBAPI 의존적 코드를 포함하는 동적공유객체(DSO: Dynamic Shared Object)[3]이며 드라이버의 로드는 응용프로그램의 실행시간에 이루어 진다. DBAPI는 Oracle, Informix, MySQL등 DBMS 벤더에 의해 제공되는 클라이언트 프로그래밍을 위한 라이브러리 시스템을 말한다.



<그림 1> DDBC의 구성요소 및 구조

1.4 객체기반 인터페이스 모델

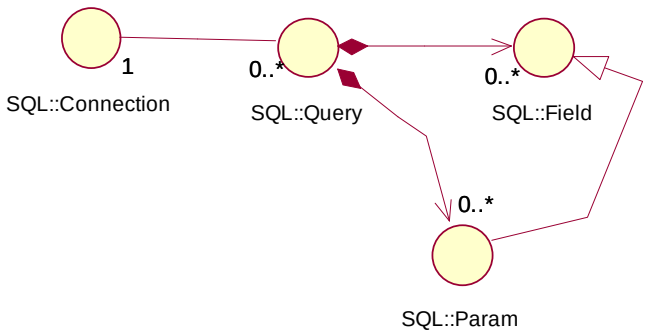
데이터베이스 서버에 접근하도록 하는 DDBC의 인터페이스는 객체기반 모델을 취하고 있다. 이들 인터페이스는 C++의 순수가상함수(pure virtual method)를 포함하는 추상(abstract)클래스로 정의되어 있

[3] 윈도우에서 DLL(Dynamic Linked Library)로 불려진다.

고 드라이버에서 이들 인터페이스들의 객체가 생성된다.

객체기반 인터페이스 모델은 어플리케이션 개발자와 드라이버 개발자 모두에게 유용하다. 어플리케이션 개발자는 객체의 구체적인 구현을 알지 못하더라도 인터페이스 명세를 통하여 객체에 접근할 수 있다. 객체의 접근하기 위해서는 인터페이스의 멤버 메소드를 통하여 이루어 지며 인터페이스 명세에 의하여 객체의 멤버 접근이 제한되기 때문에 어플리케이션 개발자의 적절하지 않은 객체접근은 컴파일 타임에서부터 금지된다.

객체기반 인터페이스 모델은 드라이버 개발자로 하여금 드라이버 개발에 있어서 별도의 명세서 없이 최소한의 구현이 가능하도록 유도할 수 있으며, 드라이버 구현에 있어서 기본 설계 및 기초 코드 작성에 관하여 묵시적인 지침을 제공한다.



<그림 2> DDBC 인터페이스

DDBC의 인터페이스는 SQL::Connection, SQL::Query, SQL::Field, SQL::Param로 이루어져 있으며 자세한 내용은 2장에서 다루고 있고 여기에서는 인터페이스들 간의 관계에 대하여 간략히 살펴본다.

SQL::Connection 객체는 데이터베이스 서버와 단일의 연결을 확립하고 트랜잭션 상태와 연결을 사용한 오퍼레이션들의 에러상태 등과 같은 정보를 유지한다. 이 객체는 CREATE TABLE, DROP TABLE과 같은 DDL(Data Definition Language)과 DELETE, UPDATE, INSERT와 같이 리턴 레코드셋이 없는 간단한 DML(Date Manipulation Language)을 실행할 수 있다.

보다 복잡한 DML을 위해서는 Connection::createQueryInstance()의 호출을 통하여 SQL::Query 객체를 생성할 수 있다. SQL::Query는 동적인 파라미터가 있는 SQL을 위하여 SQL::Param 객체를 생성하고 SELECT문과 같은 리턴 레코드셋이 있는 SQL을 실행한 후에는 SQL::Field 객체를 생성한다. SQL::Field 객체와 SQL::Param 객체는 모두 SQL::Query 내부에서 생성되고 파괴되어지기 때문에 어플리케이션에서는 이들에 대하여 참조만 할 수 있다. 한번 생성된 SQL::Query객체는 한번 이상의 서로 다른 SQL을 실행할 수 있으며 Query::prepare() 호출을 통하여 준비된 SQL은 Query::execute()를 한번 이상 호출하여 실행될 수 있다.

SQL::Query, SQL::Field, SQL::Param 객체들의 메소드 호출 후 에러가 발생하면 마지막 에러상태가

관련된 SQL::Connection 객체에 보관된다.

1.5 스레드 모델

다중 스레드 어플리케이션에 있어서 SQL::Connection객체는 원칙적으로 2개 이상의 스레드에서 공유될 수 없다. 따라서 SQL::Query, SQL::Field, SQL::Param 역시 하나의 스레드를 통해서만 접근되어야 한다. 이것은 이들 인터페이스들에 대하여 동시 접근을 허용하지 않는다는 의미이다. DDBC를 사용하여 다중 스레드 데이터베이스 어플리케이션을 개발하려면 스레드마다 별도의 SQL::Connection 객체를 두어야 한다.

제2장 인터페이스 레퍼런스

DDBC 인터페이스는 dcl/core/SQLCore.h에 단일 파일로 명세화 되어 있다. 이들 인터페이스와 관련 상수(constants)들은 class SQL의 멤버이며 이렇게 함으로써 DCL의 다른 클래스 및 전역 상수들과의 열거형(enum)관련 이름 충돌을 피하고 있다.

클래스 SQL은 객체로 인스턴스화 하고자 하는 것이 아니라 DCL 전체 클래스들에서 DDBC 인터페이스들에 이름영역(namespace)을 지정하고자 사용된다.

2.1 SQL::Error

DDBC의 모든 인터페이스들의 멤버 메소드들은 실행의 성공 여부만 true/false로 리턴한다. 만약 메소드 실행 과정에 에러가 발생하여 실패한 경우 Connection 인터페이스의 객체에 이를 보관하고 있으며 에러코드는 Connection::errorCode()를 사용하여 알아 낼 수 있다.

에러코드가 SQL::eServerError인 경우는 DBAPI 호출 실패를 의미하며 그 외에는 드라이버가 만들어 주는 값들이다. SQL::eServerError의 경우 Connection::getErrorMessage()를 사용하여 DBMS가 만들어 주는 메시지를 얻을 수 있다.

2.2 SQL::DataType

SQL::DataType은 응용프로그램과 드라이버간의 데이터 타입들을 정의하고 있으며 문맥에 따라 세 가지의 용도로 사용된다.

첫 번째는, 필드 혹은 파라미터 객체에 지정되어 있는 데이터 타입을 의미한다. Query::excute()를 호출한 후에는 서버로부터 리턴 되는 레코드들의 필드에 대한 정보들이 정해진다. 이들 필드의 데이터 타입은 Field::dataType()을 통해서 알 수 있다.

두 번째는, 필드 객체로부터 데이터를 얻기 위해 사용되는 버퍼의 데이터 타입이나 파라미터에 데이터를 설정할 때 사용될 버퍼의 데이터 타입을 지정한다. 가령 필드의 데이터 타입이 SQL::typeNumeric 일 경우 필드의 데이터를 얻기 위해 short, int, float, double과 같은 데이터 타입이 사용될 수 있다.

세 번째는, 파라미터의 데이터 타입을 지정할 때 사용되며 2.4에서 자세히 언급한다.

■ typeInteger, typeUInteger

필드의 데이터 타입이 TINY, SMALLINT, INTEGER과 같은 정수형 자료형과 BIT, BOOLEAN과 같은 불리언 값을 의미한다.

<표 1> 필드 및 파라미터 데이터 타입

SQL::DataType	SQL TYPES
typeInteger	TINY, SMALLINT, INTEGER
typeUIngeger	BIT, UNSIGNED
typeFloat	REAL, FLOAT
typeNumeric	DECIMAL, NUMERIC, NUMBER
typeDate	DATE
typeTime	TIME
typeTimeStamp	TIMESTAMP
typeTimeStampTz	TIMESTAMP WITH TIMEZONE
typeIntervalYm	INTERVAL YEAR TO MONTH
typeIntervalDs	INTERVAL, INTERVAL DAY TO SECOND
typeText	CHAR, VARCHAR, NCHAR, NVARCHAR
typeBinary	RAW
typeLongText	LONG, TEXT, sequential access CLOB
typeLongBinary	LONG RAW, BYTE, sequential access CLOB
typeClob	CLOB
typeBlob	BLOB

■ typeFloat

REAL, FLOAT, DOUBLE PRECISION과 같은 부동소수점 자료형임을 나타낸다. C언어에서 부동소수점 자료형은 float, double이 사용될 수 있으며 long double은 지원하지 않는다. long double은 컴파일러 의존적이다.

■ typeNumeric

서버의 데이터타입이 NUMERIC, DECIMAL이나 Oracle의 NUMBER 등과 같을 때 사용된다. 이 데이터 타입들은 서버의 구현마다 정밀도(precision)나 스케일(scale)을 다루는 방법, 그리고 문자열로 변환하는 방법들이 모두 다르다.

버퍼의 데이터 타입은 typeInteger, typeUInteger, typeFloat, typeText 모두 가능하다. typeInteger, typeUInteger, typeFloat을 사용하여 데이터에 접근 할 경우 SQL::eOutOfRange나 SQL::eServerError

가 발생할 수 있다. typeText는 서버의 DBAPI를 사용하여 문자열로 변환한 결과를 얻는다. 만약 버퍼의 크기가 작으면 SQL::eInvalidBufferSize가 발생한다.

■ typeDate

서버의 데이터 타입이 년, 월, 일을 갖는 DATE인 경우이다. 버퍼는 SQL::Date* 이어야 한다.

```
struct Date
{
	INT16	nYear;		// -9999 ~ 9999, DBMS dependent
	UINT8	nMonth;		// 1 ~ 12
	UINT8	nDay;		// 1 ~ 31
};
```

SQL::Date의 멤버들은 원칙적으로 양의 정수 이어야 하지만 서버마다 값의 범위가 모두 다르다. 가령 MySQL 같은 경우는 이들 멤버의 값이 모두 0 일 수도 있고 Oracle에서는 nYear가 음수인 경우도 있다.

<표 2> 버퍼 데이터 타입

SQL::DataType	Buffer	Size
typeInteger	INT8*, INT16*, INT32*, INT64*	sizeof(TYPE), 1, 2, 4, 8
typeUInteger	UINT8*, UINT16*, UINT32*, UINT64*	sizeof(TYPE), 1, 2, 4, 8
typeFloat	float*, double*, long double*	sizeof(TYPE), 4, 8, 12
typeDate	SQL::Date*	sizeof(SQL::Date)
typeTime	SQL::Time*	sizeof(SQL::Time)
typeTimeStamp	SQL::TimeStamp*	sizeof(SQL::TimeStamp)
typeTimeStampTz	SQL::TimeStamp*	sizeof(SQL::TimeStamp)
typeIntervalYm	SQL::Interval*	sizeof(SQL::Interval)
typeIntervalDs	SQL::Interval*	sizeof(SQL::Interval)
typeText	char*	n
typeBinary	unsigned char*	n
typeInputStream	InputStream*	n or EOF((size_t)-1)
typeOutputStream	OutputStream*	n or EOF(size_t)-1)

■ typeTime

서버의 데이터 타입이 시, 분, 초를 가지는 TIME인 경우이고 버퍼는 SQL::Time* 이어야 한다.

```
struct Time
{
	UINT8	nHour;		// 0 ~ 23
	UINT8	nMin;		// 0 ~ 59
```

```
    UINT8  nSec;          // 0 ~ 59
    UINT32 nFrac;         // 0 ~ 999999999, 1/1,000,000,000 fractional seconds
};
```

여기서 nFrac는 1/1,000,000,000초 값을 가진다. 가령 0.123초 이면 이 값은 123000000이다.

■ typeTimeStamp

서버의 데이터 타입이 날짜와 시간 정보를 가지는 데이터 타입이다. TIMESTAMP 외에 날짜와 시간 정보를 가지는 데이터 타입으로 Oracle의 DATE와 같은 경우를 포함할 수 있다. 버퍼는 SQL::TimeStamp* 이어야 한다.

```
struct TimeStamp
{
    INT16  nYear;         // 1 ~ 9999, -4712 ~ 9999, DBMS dependent
    UINT8  nMonth;        // 1 ~ 12
    UINT8  nDay;          // 1 ~ 31
    UINT8  nHour;         // 0 ~ 23
    UINT8  nMin;          // 0 ~ 59
    UINT8  nSec;          // 0 ~ 59
    UINT32 nFrac;         // 0 ~ 999999999, 1/1,000,000,000 fractional seconds
    INT16  nTzMin;        // -720~840, GMT-12 ~ GMT+ 14, Minutes east of UTC
};
```

버퍼의 데이터 타입이 SQL::typeTimeStamp로 지정 되면 nTzMin 멤버는 무시된다.

■ typeTimeStampTz

TIMESTAMP WITH TIME ZONE과 같은 타임존 정보를 가지는 타임스템프로 버퍼는 SQL::TimeStamp* 이어야 하고 버퍼의 타입을 typeTimeStampTz로 지정해야 한다.

SQL::TimeStamp의 멤버 중 nTzMin은 타임존(time-zone)값을 분(minute)로 표시한 것이다.

■ typeIntervalYm

서버의 데이터 타입이 INTERVAL YEAR TO MONTH이다. 버퍼는 SQL::Interval* 이어야 하고 버퍼의 타입을 typeIntervalYm로 지정해야 한다.

버퍼의 타입이 SQL::typeIntervalYm으로 지정되면 nYears, nMonths만 사용되고 나머지는 무시된다.

■ typeIntervalDs

서버의 데이터 타입이 INTERVAL DAY TO SECOND나 또는 이와 유사한 경우이다.. 버퍼는 SQL::Interval* 이어야 한다.

버퍼의 타입이 SQL::typeIntervalDs로 지정되면 nYear, nMonths는 무시된다.

```
struct Interval
{
	INT32	nYears;		// -999999999 ~ 999999999
	INT8	nMonths;	// -11 ~ 11
	INT32	nDays;		// -999999999 ~ 999999999
	INT8	nHours;		// -23 ~ 23
	INT8	nMins;		// -59 ~ 59
	INT8	nSecs;		// -59 ~ 59
	INT32	nFracs;		// -999999999 ~ 999999999, 1/1,000,000,000 fractional seconds
};
```

■ typeText

일반적으로 CHAR, VARCHAR과 같은 데이터 타입을 나타내지만 드라이버의 구현에 따라 다를 수 있으며 64KBytes 이하의 데이터 타입 중 텍스트 형식은 이 데이터 타입이 사용된다.

버퍼의 타입을 지정할 때에는 char* 를 의미하고 데이터 보다 버퍼의 크기가 작으면 버퍼의 크기만 복사한다. 버퍼의 크기가 데이터 보다 크면 복사된 데이터의 크기를 리턴한다.

파라미터 데이터 타입을 지정할 경우 typeText, typeLongText, typeClob로 구별할 수 있으며 드라이버에 따라 이것의 구현은 다를 수 있다. 가령 Informix의 TEXT 컬럼을 사용할 경우 반드시 typeLongText를 지정해야 한다.

■ typeBinary

Oracle의 RAW과 같은 경우이다.

버퍼의 타입을 지정할 때에는 unsigned char* 를 의미하고 데이터 보다 버퍼의 크기가 작으면 버퍼의 크기만 복사한다. 버퍼의 크기가 데이터 보다 크면 복사된 데이터의 크기를 리턴한다.

파라미터 데이터 타입을 지정할 경우 typeBinary, typeLongBinary로 구별할 수 있으며 드라이버에 따라 이것의 구현은 다를 수 있다.

■ typeLongText

Oracle의 LONG과 같이 크기를 지정할 수 없는 데이터 타입과 순차적으로 접근해야 하는 CLOB 데이터 타입을 의미한다. 순차적으로 접근하는 CLOB는 Informix의 TEXT, InterBase의 텍스트 타입의 BLOB, MySQL의 MIDIUMTEXT나 LONGTEXT와 같은 데이터 타입이 있다.

■ typeLongBinary

Oracle의 LONG RAW나 순차적으로 접근하는 BLOB 데이터 타입이다.

■ typeClob

직접 접근(direct access)이 가능한 CLOB를 의미한다. Oracle의 CLOB, Informix의 CLOB가 여기에 해당된다.

■ typeBlob

직접 접근(direct access)이 가능한 BLOB를 의미한다. Oracle의 BLOB, Informix의 BLOB가 여기에 해당된다.

■ typeInputStream

서버의 데이터 타입이 typeLongText, typeLongBinary, typeClob, typeBlob를 사용할 때 파라미터의 데이터를 설정할 때 사용된다.

크기를 (size_t)-1로 하면 스트림의 EOF까지 사용된다.

■ typeOutputStream

서버의 데이터 타입이 typeLongText, typeLongBinary, typeClob, typeBlob일 경우 필드나 파라미터로부터 데이터를 스트림으로 출력한다.

크기를 (size_t)-1로 하면 데이터 전체를 의미한다.

2.3 SQL::Field

질의의 결과로부터 리턴되는 레코드의 각각의 필드들의 값에 접근하는 방법을 제공하는 인터페이스이다. 이 인터페이스의 객체는 SQL::Query에 의하여 생성되고 응용프로그램에서 이 인터페이스를 생성하는 부분은 없고 참조만 가능하다.

필드는 그것의 이름과 데이터타입 프로퍼티를 가지며 데이터타입과 관련한 데이터의 최대 크기와 리턴 된 레코드 내에서의 각 필드의 크기를 알아낼 수 있으며 NULL의 여부 그리고 필드의 서버 의존적인 데이터 타입의 문자열 이름을 알아낼 수 있다.

필드 객체는 Query::execute()가 호출된 이후 초기화 된 것을 보장한다. SELECT문과 같이 하나 이상의 레코드들을 리턴하는 경우 Query::fetch() 호출 이후 데이터에 접근할 수 있으며 EXECUTE PROCEDURE와 같은 경우 Query::execute() 호출 이후 접근 가능하다.

데이터를 얻기 위한 버퍼의 데이터 타입은 typeNumeric, typeLongText, typeLongBinary, typeClob, typeBlob외에는 반드시 동일해야 하며 이 관계를 표3에서 표시하고 있다.

정수형과 실수형인 typeInterger, typeUInteger 그리고 typeFloat의 버퍼 크기는 최대 크기와 같거나

커야 한다. 가령 서버의 데이터 타입이 SMALLINT일 경우 버퍼 타입은 typeInteger이어야 하고 버퍼의 크기는 sizeof(short) 보다 같거나 커야 한다. 또 다른 예로, 서버의 데이터 타입이 REAL(single precision)이면 버퍼는 float*, double* 모두를 사용할 수 있다. 만약 FLOAT(double precision)이면 double*를 사용해야 한다.

<표 3> 필드 데이터 접근

Field DataType / Buffer DataType	typeInteger	typeUInteger	typeFloat	typeNumeric	typeDate	typeTime	typeTimeStamp	typeTimeStampTz	typeIntervalYm	typeIntervalDs	typeText	typeBinary	typeLongText	typeLongBinary	typeClob	typeBlob
typeInteger	●			●												
typeUInteger		●		●												
typeFloat			●	●												
typeDate					●											
typeTime						●										
typeTimeStamp							●									
typeTimeStampTz								●								
typeIntervalYm									●							
typeIntervalDs										●						
typeText				●							●		●		●	
typeBinary												●		●		●
typeOutputStream													●	●	●	●

typeText, typeBinary, typeOutputStream의 경우 크기가 원래 데이터 크기보다 작으면 리턴되는 데이터는 잘려진다.

각 데이터의 최대크기는 Field::getDataSize(size_t* pnSize, bool bMaxSize) 메소드의 bMaxSize에 true를 패스하여 알 수 있다.

2.4 SQL::Param

이 인터페이스는 SQL 질의에서 ':' 문자로 시작하는 단일의 문자열(placeholder라 불림)을 구별하여 이들을 파라미터로 처리할 수 있도록 하는 것이 주요 역할이다. 즉, 질의의 실행에 있어서 SQL::Query의 prepare, 파라미터의 값 설정, execute의 메커니즘을 사용이 가능하도록 하는 것 이다. 파라미터 객체 이름에서 ':' 문자는 포함되어 있지 않다.

예를 들어 "INSERT INTO ATABLE(COL1, COL2) VALUES(:COL1, :COL2)"과 같은 문장이 있을 경

우 “:COL1”, “:COL2”가 파라미터에 해당된다. 이 질의는 단 한번 준비(prepare)되어지고 파라미터값의 변경과 실행(execute)을 반복적으로 할 수 있도록 하여 성능을 향상 시킬 수 있다.

파라미터 객체는 Query::prepare()가 호출된 이후 접근 가능하며 질의가 실행(execute)된 이후의 인터페이스 객체의 상태는 NULL이 된다. 이 인터페이스에 값을 설정할 때에는 값의 데이터 타입과 파라미터 객체의 데이터 타입을 모두 지정해 주어야 한다. 가령 인포믹스의 경우 TEXT나 BYTE 컬럼에 값을 삽입하기 위해 버퍼의 타입이 SQL::typeText를 사용하게 되면 파라미터의 데이터 타입을 지정하기 위해 SQL::typeText나 SQL::typeLongText를 지정하여야 한다.

또한, 인터페이스는 SQL::Field 인터페이스로부터 상속되어 확장된 인터페이스 이기 때문에 SQL::Field의 기능을 수행하며 저장 프로시저나 함수의 실행 이후 리턴되는 값을 접근을 가능하도록 한다. 읽기 가능한 파라미터 인터페이스는 드라이버 의존적이기 때문에 모든 서버에서 가능하지는 않다.

파라미터 객체에 값을 설정하기 위해서는 Param::setData() 메소드를 호출 하여야 하며 호출이 성공적으로 이루어 지면 파라미터 객체의 데이터 타입도 결정된다. Oracle의 PL/SQL 블록의 리턴값과 같은 경우를 위해 Param::setDataType()을 사용한다.

```
bool Param::setData(
        _CONST void*    pv,             // data, IN
        size_t          nSize,          // sizeof *pv
        DataType        dataType,       // typeof *pv
        DataType        assignType      // assign to server type
        );
```

dataType	assignType
typeText	typeNumeric, typeText, typeLongText, typeClob
typeBinary	typeBinary, typeLongBinary, typeBlob
typeInputStream	typeText, typeLongText, typeClob, typeBinary, typeLongBinary, typeBlob

pv는 데이터의 버퍼를, nSize는 *pv의 크기, 그리고 dataType은 *pv의 데이터 타입을 의미한다. assignType은 파라미터 객체의 데이터 타입을 결정하며 SQL::typeText, SQL::typeBinary, SQL::typeInputStream을 제외한 모든 assignType은 dataType과 동일해야 한다. Param::setData의 assignType과 Param::setDataType의 dataType은 Field::dataType을 결정한다.

파라미터 객체는 typeText, typeBinary, typeInputStream과 함께 주어진 문자열이나 메모리 블록 및 객체를 복사하지 않고 참조만 한다. 따라서 Query::execute()를 호출하기 전에는 모든 메모리 객체가 유지되어 있어야 한다.

2.5 SQL::Query

SELECT, INSERT, UPDATE, DELETE 등과 같은 데이터 조작을 위한 대부분의 질의는 이 인터페이스의 객체를 사용한다. 이 인터페이스의 객체는 SQL::Connection 인터페이스를 통하여 생성되며

prepare, execute, fetch 세 가지의 주요 멤버를 가진다.

prepare는 SQL을 파싱하고 실행할 수 있도록 준비한다. 입력된 SQL에 ':'를 포함한 단일 문자열을 가지면 이들에 대하여 파라미터 객체의 컬렉션(collection)을 생성하고 execute를 호출하기 전에 파라미터 값을 설정할 수 있도록 한다.

execute는 설정된 파라미터 값을 사용하여 SQL을 실행한다. SELECT 문과 같이 리턴 레코드를 가지는 SQL이면 리턴되는 필드 객체의 컬렉션이 생성된다.

fetch는 서버로부터 단일 레코드를 인출한다.

2.6 SQL::Connection

데이터베이스 서버와의 연결을 유지하고 트랜잭션을 관리하는 인터페이스이다. open으로 데이터베이스 서버에 연결하고 close로 연결을 닫는다. open에 사용할 연결문자열은 해당 드라이버 구현에 의존적이므로 해당 드라이버의 연결문자열 부분을 참고한다.

startTrans, commitTrans, rollbackTrans는 드라이버에서 기본적으로 구현한 트랜잭션 제어를 한다. DBMS 의존적인 트랜잭션 제어를 위해서는 execute에 트랜잭션 처리 SQL을 전달하여 실행한다.

execute는 저장프로시저나 SELECT 문과 같이 리턴값이나 리턴 레코드셋을 포함하는 SQL과 동적 파라미터를 명시하는 문자열(':'로 시작하는 문자열, placeholder)을 포함할 수 없다.

SQL::Field, SQL::Param, SQL::Query, SQL::Connection의 모든 멤버함수 호출 후 실패를 하게 되면 이 인터페이스에 에러정보를 유지하고 있다. 만약 에러의 원인이 SQL::eServerError 이면 Connection::getErrorMessage() 메소드를 호출하여 서버 의존적인 에러 메시지를 얻을 수 있다.

2.7 SQL::DRIVER_MODULE

DDBC의 드라이버는 운영체제의 단일의 동적공유객체(DSO: dynamic shared object) 파일로 구현되며 이들 파일의 SQL::DRIVER_MODULE을 통하여 인터페이스의 객체들을 사용할 수 있다. DSO의 진입점(entry point)은 다른 DCL의 DSO와 마찬가지로 dcl/_Config.h에 정의된 DCL_DSO_ENTRY_POINT로 명명된 전역 변수이다.

uSize, uDCLVersion, uBuildFlag, uModuleType, uVersion은 드라이버의 유효성을 확인하기 위해 사용된다. pszServerTitle은 드라이버 식별을 위한 상수 문자열이다. 가령 Oracle 드라이버에는 "Oracle"이란 문자열을 가리킨다. pszDescription은 DSO의 간략한 설명을 포함한다.

```
struct DRIVER_MODULE
{
        // DCL common members
        UINT32          uSize;          // structure size
        UINT32          uDCLVersion;    // DCL_VERSION
        UINT32          uBuildFlag;     // release(0), debug(1)
        UINT32          uModuleType;    // DCL_SQL_DRIVER_MODULE
```

```
    const char*             pszDescription;         // module description

    // private members
    UINT32                  uVersion;               // DCL_SQL_VERSION
    const char*             pszServerTitle;
    const char*             pszFileVersion;
    bool                    (*pfnInitialize)();
    bool                    (*pfnCleanup)();
    Connection*             (*pfnCreateConnectionInstance)();
};
```

pfnInitialize는 드라이버가 열려지고 최초로 반드시 호출되어야 하고 드라이버가 닫혀지지 전에 pfnCleanup을 호출한다.

드라이버의 적재가 성공적으로 완료되면 pfnCreateConnectionInstance를 통하여 SQL::Connection 객체를 생성하여 사용할 수 있다.

제3장 드라이버 메니저

DDBC 드라이버 메니저는 별도의 환경으로 설치되지 않고 DCL의 Core 라이브러리(DCLCore.dll, libDCLCore.so)에 함께 구현되어 있으며 다음의 SQLDriver와 SQLDriverPool은 드라이버 메니저의 역할을 수행하는 클래스로 3.3에서 설명하는 드라이버 설치정보를 사용하여 동작한다.

3.1 SQLDriver

이 클래스는 드라이버 설치환경으로부터 드라이버를 로드하고 드라이버로부터 SQL::Connection 객체를 얻어오기 위하여 사용된다.

드라이버는 DSO(Dynamic Shared Object)로 구현되어 있기 때문에 SQLDriver는 운영체제의 DSO관련 API를 사용된다. 드라이버 DSO에는 SQL::DRIVER_MODULE로 정의된 단 한 개의 심볼이 있고 이 심볼을 찾은 후 SQL::DRIVER_MODULE의 각 멤버의 값들이 적법하면 이 드라이버로부터 SQL::Connection 객체를 얻어서 사용할 수 있다.

이러한 전체 과정은 SQLDriver의 정적(static)메소드인 getDriver()를 통해서 내부적으로 이루어지며 더 이상 드라이버를 사용하지 않는다면 closeDriver() 정적 메소드를 통하여 드라이버는 닫혀진다.

3.2 SQLDriverPool

DDBC는 하나의 어플리케이션에서 하나 이상의 서로 다른 종류의 서버를 사용할 수 있다. 가령 Oracle을 사용하면서 Informix나 MySQL을 사용할 수 있다. 이렇게 하도록 하는 클래스가 SQLDriverPool 이다. 사실 어플리케이션에서는 이 클래스와 직접적인 작업은 할 수 없으며 대신에 SQLDriver의 정적 메소드인 getDriver()와 closeDriver()를 통하여 내부적으로 사용되어 진다.

4장에서 언급하게 되는 SQLConnection 클래스는 드라이버의 로드와 사용에 관해서 어플리케이션으로부터 감추기 때문이 이를 알 필요가 없으며 어플리케이션 개발자는 드라이버가 정확하게 설치되어 있는지에 대해서만 확인하면 된다.

3.3 드라이버의 설치

드라이버를 사용하기 위해서는 드라이버가 설치되어 있어야 한다. 드라이버의 설치는 운영체제마다 약간 다르지만 기본적으로 파일시스템의 특정 폴더에 드라이버 파일을 복사하고 이들에 대한 설치 정보를 설정하면 된다.

드라이버의 설치는 크게 Microsoft Windows와 UNIX의 두 가지로 구별된다.

■ Windows

윈도우 운영체제에서 드라이버 설치 정보는 시스템 레지스트리에 저장되며 다음의 두 레지스트리키가 사용된다.

```
HKEY_CURRENT_USER\Software\Kim Daejung\DCL
HKEY_LOCAL_MACHINE\Software\Kim Daejung\DCL
```

DCLCore.dll에 포함되어 있는 드라이버 메니저는 위의 두 키 중에서 HKEY_CURRENT_USER 하위에서 드라이버 설치정보를 찾고 만약 없으면 HKEY_LOCAL_MACHINE의 하위에서 찾는다. 만약 이 둘 모두에서 설치정보를 찾지 못하면 드라이버가 설치되지 않은 것으로 간주한다.

다음은 드라이버 설치에 대한 레지스트리 내용을 보여주고 있다.

```
Windows Registry Editor Version 5.00

[HKEY_LOCAL_MACHINE\ Software\Kim Daejung\DCL]

[HKEY_LOCAL_MACHINE\ Software\Kim Daejung\DCL\INFORMIX]
"DEBUG_DRIVER"="D:\\HOME\\dcl\\lib\\DCLInfd.dll"
"DRIVER"="D:\\HOME\\dcl\\lib\\DCLInf.dll"

[HKEY_LOCAL_MACHINE\ Software\Kim Daejung\DCL\INTERBASE]
"DEBUG_DRIVER"="D:\\HOME\\dcl\\lib\\DCLIntd.dll"
"DRIVER"="D:\\HOME\\dcl\\lib\\DCLInt.dll"

[HKEY_LOCAL_MACHINE\ Software\Kim Daejung\DCL\MySQL]
"DEBUG_DRIVER"="D:\\HOME\\dcl\\lib\\DCLMyd.dll"
"DRIVER"="D:\\HOME\\dcl\\lib\\DCLMy.dll"
```

```
[HKEY_LOCAL_MACHINE\ Software\Kim Daejung\DCL\ORACLE]
"DRIVER"="D:\\HOME\\dcl\\lib\\DCLOra.dll"
"DEBUG_DRIVER"="D:\\HOME\\dcl\\lib\\DCLOrad.dll"
```

이 시스템에는 Informix, InterBase, MySQL, Oracle 드라이버의 설치되어 있고 그 모듈들은 D:₩HOME₩dcl₩lib 하위 폴더에 위치하고 있다. “DRIVER”는 릴리즈 모듈을 명시하고 “DEBUG_DRIVER”는 디버그 버전으로 빌드된 모듈이다.

어플리케이션이 DEBUG버전으로 빌드되어 DCLCored.dll을 사용하게 되면 “DEBUG_DRIVER”를 사용하게 되고 그렇지 않고 DCLCore.dll을 사용하는 릴리즈 버전이면 “DRIVER”에 표시된 경로의 모듈을 사용하게 된다.

■ UNIX

GNU/Linux를 비롯한 UNIX 버전은 다음의 두 파일을 순차적으로 확인한다.

```
$HOME/.dcl.ini
/etc/dcl.ini
```

UNIX 버전에서 드라이버 메니저는 프로세스 소유자의 홈 디렉토리를 “HOME” 환경변수를 통하여 확인한 후 홈디렉토리의 “.dcl.ini” 파일을 확인한 다음 이 파일이 없으면 “/etc/dcl.ini” 파일을 사용한다. 두 파일이 모두 없으면 드라이버가 설치되지 않은 것으로 간주한다.

다음의 dcl.ini의 내용은 Oracle과 InterBase 드라이버가 /usr/lib/dcl 디렉토리에 설치되어 있다는 내용이다.

```
; dcl.ini
[ORACLE]
DRIVER=/usr/lib/dcl/DCLOra.so
DRBUG_DRIVER=/usr/lib/dcl/DCLOrad.so
[INTERBASE]
DRIVER=/usr/lib/dcl/DCLInt.so
LOAD_OPTIONS=dllNoDelete
```

UNIX 버전에서의 LOAD_OPTIONS은 드라이버가 언로드될 때 프로세스가 core dump되는 경우 dllNoDelete를 기술할 수 있다. 이것은 dlopen() 시스템콜을 사용할 때 RTLD_NODELETE 플래그를 설정한다. 자세한 내용은 dlopen 매뉴얼을 참고한다.

제4장 어플리케이션의 개발

DDBC 인터페이스를 사용하여 어플리케이션을 개발할 수도 있으나 이는 코드 작성에 있어서 여러모로 불편함을 초래한다.

특히, 어플리케이션 작성에 있어서 C++의 예외(exception) 메커니즘을 사용할 수 없다. SQL::Connection, SQL::Query, SQL::Field, SQL::Param 객체들은 C++ 예외를 사용하지 않는다. 메소드 호출 후 에러가 발생할 경우 false를 리턴하며 어플리케이션은 SQL::Connection 객체에서 에러 결과를 확인하고 에러 메시지를 생성하여야 한다. 이 장에서 설명하고 있는 SQLConnection, SQLQuery, SQLField, SQLParam 클래스는 각각 관련한 인터페이스들을 추상화 함과 동시에 C++ 예외 메커니즘을 사용하도록 작성되어 있다.

DDBC 인터페이스에서 예외의 메커니즘을 사용하지 않는 이유는 DCL에서 사용하고 있는 예외처리 방법이 기본적으로 Exception로부터 상속된 클래스의 객체 포인터를 던지기(throw) 때문이다. 만약 드라이버로부터 예외가 던져지면 이것은 드라이버로부터 생성된 객체를 어플리케이션에서 파괴해야 하기 때문에 어플리케이션은 메모리관련 버그를 포함할 가능성이 매우 높아진다. 따라서 DCL의 모든 모듈들은 각 모듈에서 사용한 동적메모리는 그 모듈 범위 내에서 처리되도록 하고 있다.

4.1 SQLConnection

이 클래스는 SQL::Connection 인터페이스를 추상화 하는 역할 외에 객체 생성시에 드라이버를 통하여 SQL::Connection 객체를 얻어서 자신을 초기화 한다.

SQLConnection::SQLConnection

```
SQLConnection(SQLDriver* pSQLDriver);
SQLConnection(const String& strSQLDriverName) __DCL_THROWS1(SQLDriverException*);
```

드라이버로부터 SQL::Connection 객체를 얻어와서 자신을 초기화 한다. strSQLDriverName은 시스템에 설치된 드라이버의 이름이고 이것을 사용하게 될 경우 드라이버의 로드 또는 드라이버 검사절차 과정에서 오류가 발생할 경우 SQLDriverException* 예외를 던진다. pSQLDriver를 파라미터로 받는 생성자는 이미 얻은 드라이버로부터 새로운 SQL::Connection 객체를 얻어와서 자신을 초기화 한다.

```
try
{
        SQLConnection conn("Oracle");
        conn.open("USER=scott; PASSWORD=tiger; DATABASE=ORCL");
        conn.execute("CREATE TABLE AA(ID NUMBER)");
        conn.startTrans();
        conn.execute("INSERT INTO AA VALUES(1)");
        conn.commitTrans();
        conn.execute("DROP TABLE AA");
```

```
        conn.close();
}
catch (Exception* e)
{
        out << e->getMessage() << "\n";
        e->destroy();
}
```

SQLConnection::open

```
void open(const char* pszConnectionString) __DCL_THROWS1(SQLException*);
```

연결 문자열을 사용하여 데이터베이스 서버에 연결한다. 연결문자열에 포함될 수 있는 기본적인 요소는 SERVER, DATABASE, USER, PASSWORD가 있으며 특정 데이터베이스 서버 의존적인 요소가 있을 수 있기 때문에 이를 위해서는 드라이버 매뉴얼을 참고하여야 한다. 각 요소의 구분은 ';' 문자를 사용하고 각각의 값은 '=' 문자를 사용한다.

SQLConnection::close

```
void close() __DCL_THROWS1(SQLException*);
```

서버와의 연결을 닫는다. 다시 연결하고자 하면 open() 메소드를 사용한다.

SQLConnection::execute

```
void execute(const char* pszSQL) __DCL_THROWS1(SQLException*);
```

DDL과 같은 SQL이나 동적인 파라미터가 없고 리턴 레코드셋을 가지지 않는 SQL을 실행한다. 이 메소드는 서버 의존적인 트랜잭션 관련 SQL을 실행하는데 적절하다.

SQLConnection::startTrans

```
void startTrans() __DCL_THROWS1(SQLException*);
```

드라이버에서 기본적으로 제공하는 트랜잭션을 시작한다. 서버 의존적인 트랜잭션 문장을 실행하려면 execute() 메소드를 사용한다.

SQLConnection::commitTrans

```
void commitTrans() __DCL_THROWS1(SQLException*);
```

트랜잭션을 완료한다.

SQLConnection::rollbackTrans

void rollbackTrans() __DCL_THROWS1(SQLException*);

트랜잭션을 취소한다.

SQLConnection::serverInfo

String serverInfo() __DCL_THROWS1(SQLException*);

서버에 대한 간략한 정보를 얻는다.

SQLConnection::canTransact

bool canTransact() const;

서버가 트랜잭션이 가능하면 true를 그렇지 않으면 false를 리턴한다.

SQLConnection::driver

SQLDriver* driver() const;

객체가 사용하고 있는 드라이버 객체를 리턴한다. 리턴된 객체를 사용하여 또 다른 연결을 만들 수 있다.

SQLConnection::connected

bool connected() const;

객체가 서버에 연결되어 있으면 true를 리턴한다.

SQLConnection::inTransaction

bool inTransaction() const;

트랜잭션 중이면 true를 리턴한다. 트랜잭션 상태에서 commitTrans()나 rollbackTrans()를 사용하여 트랜잭션을 마치면 false를 리턴한다.

4.2 SQLConnectionPool

이 클래스는 하나의 어플리케이션에서 같은 드라이버를 사용하는 여러 개의 SQLConnection 객체를 위해 사용된다. SQLConnection 객체는 둘 이상의 스레드에서 공유될 수 없기 때문에 둘 이상의 스레드가 같은 데이터베이스 서버에 연결하도록 하려면 각 스레드마다 새로운 연결을 만들어야 한다. 또한 사

용이 완료된 연결은 시스템 자원을 위해서 연결을 닫을 수도 있고 나중에 다시 연결하는 오버헤드를 피하기 위하여 유휴(idle)연결 상태를 유지하고 있을 수도 있다. 이 클래스는 이러한 일련의 처리를 자동화 한다.

SQLConnectionPool::SQLConnectionPool

```
SQLConnectionPool(
        const String& strConnectionString,
        const String& strSQLDriverName
        ) __DCL_THROWS1(SQLDriverException*);
```

strSQLDriverName은 SQLConnection이 사용할 드라이버를 기술한다. strConnectionString은 SQLConnection::open() 메소드에서 사용할 문자열이다.

```
try
{
        SQLConnectionPool pool("USER=scott; PASSWORD=tiger; DATABASE=ORCL", "Oracle");

        pool.setMaxCount(2, 10);     // 최대 10개, 유휴 2개로 설정

        SQLConnection* pConn = pool.getConnection();
        if (pConn == NULL)
        {
                // 최대 연결상태에 도달
                return;
        }
        // 정상적인 작업
        …

        pool.release(pConn);
}
catch (Exception* e)
{
        e->destroy();
}
```

SQLConnectionPool::getConnection

```
SQLConnection* getConnection() __DCL_THROWS1(SQLException*);
```

유휴(idle) 연결객체가 있으면 그것을 리턴하고 그렇지 않으면 새로운 연결을 만들어서 리턴한다. 만약 연결풀 객체가 생성된 후 setMaxCount에 nMax를 0보다 큰 값으로 설정할 경우 연결풀이 관리하는 연결객체가 nMax에 도달한 상태에서 유휴 연결객체가 없으면 NULL을 리턴한다.

SQLConnectionPool::release

```
void release(SQLConnection* pConn);
```

사용이 완료된 연결객체 pConn을 연결풀에 되돌린다. 만약 사용 완료된 객체를 되돌리지 않으면 SQLConnectionPool은 getConnection() 호출때마다 새로운 연결을 만들게 된다.

SQLConnectionPool::setMaxCount

```
bool setMaxCount(int nMaxIdle, int nMax);
```

연결풀이 관리할 최대 연결객체의 개수와 최대 유휴 연결객체를 설정한다. nMax가 0이면 최대 개수에 대하여 제한이 없다.

SQLConnectionPool::count

```
int count() const;
```

연결풀에 있는 모든 연결객체의 개수를 리턴한다.

SQLConnectionPool::idleCount

```
int idleCount() const;
```

연결풀에 있는 연결객체 중에 유휴 객체의 개수를 리턴한다.

SQLConnectionPool::clear

```
void clear(int nWaitSeconds);
```

모든 연결을 닫고 연결객체들을 파괴한다. nWaitSeconds는 사용중인 연결객체가 release()를 통하여 유휴상태가 되도록 기다리는 최대 시간(초)으로 이 값이 경과하면 연결객체는 강제로 파괴된다.

4.3 SQLQuery

SQL::Query 인터페이스를 추상화 하여 SELECT문과 동적 파라미터를 가지는 SQL을 실행한다. 이 클래스는 이 외에도 SQLFields와 SQLParams 객체를 멤버로 가지는데 이를 통하여 이름과 인덱스 기반으로 필드나 파라미터에 접근할 수 있도록 해준다.

SQLQuery::SQLQuery

```
SQLQuery(SQLConnection* pSQLConnection) __DCL_THROWS1(SQLException*);
```

pSQLConnection으로부터 SQL::Query 객체를 얻어와서 자신을 초기화 하며 이때 pSQLConnection은 서버와 연결되어 있어야 한다.

SQLQuery::prepare

```
void prepare(const char* pszSQL) __DCL_THROWS1(SQLException*);
```

SQL을 준비(prepare) 한다. 이것의 사용은 ':' 문자를 포함하는 동적 파라미터를 가지는 SQL을 준비하여 한번 이상 실행(execute) 하도록 하는데 사용된다. 이 메소드 실행 후 파라미터 객체의 값을 설정할 수 있다.

```
try
{
        SQLQuery qry(pSQLConn);
        qry.prepare("INSERT INTO AA(ID, NAME) VALUES(:ID, :NAME)");
        qry.params().byName("ID").setValue(1);
        qry.params().byName("NAME").setValue("홍길동");
        qry.execute();
        qry.params()[0].setValue(2);
        qry.params()[1].setValue("강감찬");
        qry.execute();
}
catch(SQLException* e)
{
        e->destroy();
}
```

SQLQuery::execute

```
void execute() __DCL_THROWS1(SQLException*);
void execute(const char* pszSQL) __DCL_THROWS1(SQLException*);
```

준비된(prepared) SQL을 실행하거나 주어진 SQL을 준비(prepare)하고 실행한다. 두 번째의 경우는 동적 파라미터를 가질 수 없다.

SELECT문 일 경우 이 메소드 호출 후의 SQLQuery객체는 리턴되는 레코드의 필드 정보에 대하여 접근할 수 있으나 fetch() 메소드를 호출 하기 전에는 데이터에 접근 할 수 없다.

저장 프로시저(stored procedure)와 같은 SQL의 실행 후 이 메소드 호출 후 아웃풋(output) 파라미터의 값을 읽을 수 있다. 저장 프로시저의 실행 결과를 얻기 위한 방법은 드라이버마다 다를 수 있다. Oracle의 경우는 아웃풋 파라미터를 통해서 결과를 얻지만 Informix나 InterBase의 경우 필드들을 통해서 결과가 리턴된다. 자세한 내용은 드라이버 설명을 참고한다.

SQLQuery::fetch

void fetch() __DCL_THROWS1(SQLException*);

SELECT문과 같이 리턴 레코드셋을 가지는 SQL의 실행 결과로부터 하나의 레코드를 서버로부터 인출(fetch)한다. 이 메소드 실행 후에는 SQLQuery::eof()의 값을 확인하여야 한다. 만약 이 값이 true이면 필드데이터는 유효하지 않다.

```
try
{
        SQLQuery qry(pSQLConn);
        qry.execute("SELECT ID, NAME FROM AA");
        qry.fetch();
        while(!qry.eof())
        {
                SQLFields& fields = qry.fields();
                out << "ID : " << fields[0].asString()
                        << ", NAME : " << fields[1].asString()
                        << "\n";
                qry.fetch();
        }
}
catch(SQLException* e)
{
        e->destroy();
}
```

SQLQuery::eof

bool eof() const;

fetch() 후출 후 EOF를 검사한다.

SQLQuery::affectedRows

int affectedRows() const;

INSERT, UPDATE, DELETE문의 실행 후 삽입된 행의 개수, 수정된 혹은 삭제된 행의 개수를 리턴한다.

SQLQuery::fields

_CONST SQLFields& fields() _CONST;

레코드 접근을 위한 필드들의 콜렉션의 참조를 리턴한다.

SQLQuery::params

_CONST SQLParams& params() _CONST;

파라미터 접근을 위한 파라미터들의 콜렉션의 참조를 리턴한다.

4.4 SQLField

이 클래스는 SQL::Field 인터페이스를 추상화하고 필드데이터에 대한 구체적인 접근 방법들을 제공함과 동시에 C++의 예외 메커니즘을 제공한다.

SQLField::name

const String& name() const;

필드의 이름에 대한 스트링 참조를 리턴한다.

SQLField::dataType

SQL::DataType dataType() const;

필드의 데이터 타입을 리턴한다.

SQLField::dataTypeName

const char* dataTypeName() const;

dataType()에 해당하는 문자열 이름을 리턴하며 그 값은 “typeInteger”, “typeUInteger”, “typeFloat”, “typeText” 등과 같다.

SQLField::serverDataTypeName

const char* serverDataTypeName() const;

서버 의존적인 데이터 타입의 이름을 리턴한다. 예를들어 Oracle 드라이버의 경우 SQL::dataType()의 값이 SQL::typeText일 경우 “CHAR”, “VARCHAR2”을, SQL::typeLongText일 경우 “LONG”이다.

SQLField::isNull

bool isNull() const;

필드가 NULL이면 true를 리턴한다.

SQLField::dataSize

```
size_t dataSize() __DCL_THROWS1(SQLException*);
```

현재 필드의 데이터 크기를 바이트의 개수로 리턴한다. SELECT 문과 같이 리턴 레코드셋을 가지는 경우 SQLQuery::fetch()가 실행된 다음 호출이 가능하다.

SQLField::dataSizeMax

```
size_t dataSizeMax() __DCL_THROWS1(SQLException*);
```

필드데이터의 최대 크기를 리턴한다.

SQLField::getData

```
void getData(
        void*               pv,             // buffer
        size_t*             pn,             // sizeof *pv
        SQL::DataType dataType              // buffer data type
        ) __DCL_THROWS1(SQLException*);
```

필드 객체로부터 데이터를 얻는다. 특수한 경우가 아닌 이상 getValue()나 asXXXX()를 사용하여 데이터에 접근하는 것이 권장된다.

SQLField::getValue

```
void getValue(INT32& rValue) __DCL_THROWS1(SQLException*);
void getValue(UINT32& rValue) __DCL_THROWS1(SQLException*);
void getValue(INT64& rValue) __DCL_THROWS1(SQLException*);
void getValue(UINT64& rValue) __DCL_THROWS1(SQLException*);
void getValue(float& rValue) __DCL_THROWS1(SQLException*);
void getValue(double& rValue) __DCL_THROWS1(SQLException*);
void getValue(SQL::Date& rValue) __DCL_THROWS1(SQLException*);
void getValue(SQL::Time& rValue) __DCL_THROWS1(SQLException*);
void getValue(SQL::TimeStamp& rValue) __DCL_THROWS1(SQLException*);
void getValue(SQL::Interval& rValue) __DCL_THROWS1(SQLException*);
void getValue(String& rValue) __DCL_THROWS1(SQLException*);
void getValue(OutputStream& rValue, size_t n = (size_t)-1) __DCL_THROWS1(SQLException*);
```

rValue로 필드의 데이터를 가져온다. 필드 데이터를 위한 버퍼의 결정은 SQL::Field의 내용을 참고한다. OutputStream을 파라미터로 할 경우 n이 (size_t)-1 일 경우 데이터의 EOF까지를 의미한다.

SQLField::asXXXX

```
bool asBoolean() __DCL_THROWS1(SQLException*);
int asInteger() __DCL_THROWS1(SQLException*);
INT32 asInt32() __DCL_THROWS1(SQLException*);
INT64 asInt64() __DCL_THROWS1(SQLException*);
float asSingle() __DCL_THROWS1(SQLException*);
double asDouble() __DCL_THROWS1(SQLException*);
Date asDate() __DCL_THROWS1(SQLException*);
Time asTime() __DCL_THROWS1(SQLException*);
DateTime asDateTime() __DCL_THROWS1(SQLException*);
Interval asInterval() __DCL_THROWS1(SQLException*);
String asString() __DCL_THROWS1(SQLException*);
String asStringF(const char* pszFormat = NULL) __DCL_THROWS1(SQLException*);
```

필드로부터 리턴될 데이터타입으로 데이터를 변환하여 리턴한다. 변환이 불가능할 데이터 타입인 경우 예외를 던진다.

<표 4> 필드 데이터의 변환함수

Field / DataType / axXXXX	typeInteger	typeUInteger	typeFloat	typeNumeric	typeDate	typeTime	typeTimeStamp	typeTimeStampTz	typeIntervalYm	typeIntervalDs	typeText	typeBinary	typeLongText	typeLongBinary	typeClob	typeBlob
asBoolean	●	●	○	○							○					
asInt32	●	○	○	●							○					
asInt64	●	○	○	●							○					
asSingle	○	○	●	●							○					
asDouble	○	○	●	●							○					
asDate					●		○	○								
asTime						●	○	○								
asDateTime					○	○	●	●								
asInterval									●	●						
asString	○	○	○	●	○	○	○	○	○	○	●	●	●	●	●	●
asStringF	○	○	○	○	○	○	○	○	○	○	●	○	●	○	●	○

asBoolean은 숫자 필드에 대하여 0인 값에 대하여 false를 그렇지 않으면 true를 리턴한다.

asInteger는 라이브러리가 32비트 혹은 64비트로 빌드되는 것에 대하여 asInt32와 asInt64에 대응된다.

asStringF는 필드의 데이터 타입에 적절한 포멧 문자열을 제공할 수 있다. 필드의 데이터 타입이 숫자일 경우 10진 문자열에 관한 포멧 문자열 이어야 하며 이것은 dcl/core/Numeric.h을 참고한다. 날짜 시간일 경우 C 런타임 함수의 strftime()을 사용한다. 필드의 데이터 SQL::typeBinary 및 SQL::typeLongBinary일 경우 최대 40바이트에 대하여 16진수 문자열로 변환하여 리턴한다.

4.5 SQLParam

SQLField로부터 상속되어 SQL::Param 인터페이스를 추상화 한다.

SQLParam::setNull

```
void setNull();
```

파라미터를 NULL 상태로 만든다. SQLQuery::execute() 호출 후의 파라미터 객체는 항상 NULL로 설정된다.

SQLParam::setData

```
void setData(
        _CONST          void* pv,           // data, IN
        size_t          nSize,              // sizeof *pv
        SQL::DataType   dataType,           // typeof *pv
        SQL::DataType   assignType          // assign to server type
        ) __DCL_THROWS1(SQLException*);
```

파라미터 객체에 데이터를 설정한다. 이 메소드 호출 후에 assignType이 파라미터 객체의 데이터타입으로 설정된다.

SQLParam::setDataType

```
void setDataType(SQL::DataType dataType) __DCL_THROWS1(SQLException*);
```

파라미터 객체의 데이터 타입을 설정한다. 파라미터 객체의 데이터타입은 setData()를 호출하는 것으로도 결정이 되지만 아웃파라미터의 경우 SQLQuery::execute() 호출 전에 이 메소드를 호출하여 파라미터의 데이터 타입을 설정해야 한다.

SQLParam::setData

```
void setValue(int value) __DCL_THROWS1(SQLException*);
```

```
void setValue(unsigned int value) __DCL_THROWS1(SQLException*);
void setValue(INT32 value) __DCL_THROWS1(SQLException*);
void setValue(UINT32 value) __DCL_THROWS1(SQLException*);
void setValue(INT64 value) __DCL_THROWS1(SQLException*);
void setValue(UINT64 value) __DCL_THROWS1(SQLException*);
void setValue(float value) __DCL_THROWS1(SQLException*);
void setValue(double value) __DCL_THROWS1(SQLException*);
void setValue(const Decimal& value) __DCL_THROWS1(SQLException*);
void setValue(Date value) __DCL_THROWS1(SQLException*);
void setValue(Time value) __DCL_THROWS1(SQLException*);
void setValue(DateTime value) __DCL_THROWS1(SQLException*);
void setValue(DateTime value, INT16 nTzMin) __DCL_THROWS1(SQLException*);

void setValue(
        Interval value,
        SQL::DataType assignType = SQL::typeIntervalDs
        ) __DCL_THROWS1(SQLException*);

void setValue(
        const String& value,
        SQL::DataType assignType = SQL::typeText
        ) __DCL_THROWS1(SQLException*);

void setValue(
        const char* p, size_t n,
        SQL::DataType assignType = SQL::typeText
        ) __DCL_THROWS1(SQLException*);

void setValue(
        const BYTE* p, size_t n,
        SQL::DataType assignType = SQL::typeBinary
        ) __DCL_THROWS1(SQLException*);

void setValue(
        _CONST InputStream* pInput, size_t n = (size_t)-1,
        SQL::DataType assignType = SQL::typeLongText
        ) __DCL_THROWS1(SQLException*);
```

SQLParam::setData()를 사용하여 필드에 데이터를 설정한다.

파라미터 객체는 설정된 데이터에 대하여 그 객체를 유지하지 않기 때문에 const char*, const

BYTE*, _CONST InputStream* 등과 같이 데이터 또는 객체의 포인터를 사용하여 값을 설정할 때에는 SQLQuery::execute()이 실행될 때 까지 객체의 상태를 유지하여야 한다.

4.6 SQLFields

필드객체의 콜렉션을 유지하고 필드에 접근하기위한 참조 방법을 제공한다. 이 클래스의 인스턴스는 SQLQuery 클래스 내부에서만 사용하기 때문에 어플리케이션은 참조만 가능하다.

SQLFields::operator[]

_CONST SQLField& operator[](int nIndex) const ;

0(zero) 기반으로 필드에 대한 참조를 리턴한다.

SQLFields::byName

_CONST SQLField& byName(const char* pszFieldName) _CONST
__DCL_THROWS1(InvalidIndexException*);

pszFieldName에 해당하는 필드객체의 참조를 리턴한다.

SQLFields::count

int count() const;

필드객체의 개수를 리턴한다.

4.7 SQLParams

필드객체의 콜렉션을 유지하고 필드에 접근하기위한 참조 방법을 제공한다. 이 클래스의 인스턴스는 SQLQuery 클래스 내부에서만 사용하기 때문에 어플리케이션은 참조만 가능하다.

SQLParams:operator[]

_CONST SQLParam& operator[](int nIndex) const ;

0(zero) 기반으로 파라미터객체에 대한 참조를 리턴한다.

SQLParams::byName

_CONST SQLField& byName(const char* pszParamName) _CONST
__DCL_THROWS1(InvalidIndexException*);

pszParamName 에 해당하는 파라미터객체의 참조를 리턴한다.

SQLParams::count

int count() const;

파라미터객체의 개수를 리턴한다.

제5장 Informix

5.1 개요

- 서버버전 : Informix Database Server 5.x 이상의 버전(OnLine, SE, Dynamic Server, XPS, Universal Server)에 적용할 수 있다. INT8, SERIAL8 이외의 Universal Server의 데이터 타입은 지원하지 않는다.
- 구현 : ESQL/C
- System Requirements : Informix Connect
- 드라이버 파일명
  Win32 : DCLInf.dll
  UNIX : DCLInf.so

5.2 연결문자열

- USER
- PASSWORD
- SERVER : 데이터베이스 서버의 호스트명
- DATABASE

연결문자열에서 DATABASE 부분이 생략되면, 연결 이후 CREATE DATABASE 문을 사용하여 데이터베이스를 생성할 수 있다.

5.3 데이터 타입

Informix Internal	SQL::DataType	비고
CHAR(n), CHARACTER(n) NCHAR(n)	typeText	
VARCHAR(m[,r]) CHARACTER VARYING(m[,r]) NVARCHAR(m[,r])	typeText	m : maximum size r : minimum reserved space
SMALINT	typeInteger	

INT, INTEGER, SERIAL(n)	typeInteger	n : reset value
SMALLFLOAT, REAL	typeFloat	
FLOAT[(n)], DOUBLE PRECESION	typeFloat	1 <= n <= 14
DECIMAL(p)	typeNumeric	floating point decimal
NUMERIC(p, s), DECIMAL(p, s) DEC(p, s), MONEY(p, s)	typeNumeric	fixed point decimal p : precision, s : scale
DATE	typeDate	
DATETIME	typeTimeStamp	
DATETIME HOUR TO	typeTime	
INTERVAL YEAR TO	typeIntervalYm	
INTERVAL DAY TO	typeIntervalDs	
TEXT	typeLongText	
BYTE	typeLongBinary	
INT8, SERIAL8	typeInteger	Universal Server
LIST	N/A	Universal Server
SET	N/A	Universal Server
MULTISET	N/A	Universal Server
ROW	N/A	Universal Server
LVARCHAR	typeText	Universal Server
BOOLEAN	N/A	Universal Server
CLOB	N/A	Universal Server
BLOB	N/A	Universal Server

5.4 트랜잭션

인포믹스는 데이터베이스 생성 옵션에 따라 트랜잭션 가능여부가 결정된다. 드라이버에서는 연결문자열에 DATABASE 값이 있고 연결이 성공되면 BEGIN WORK 시도하여 에러가 발생하지 않으면 트랜잭션 가능상태를 설정한다. 그 외에는 트랜잭션 불능상태를 유지한다.

DATABASE 문을 사용한 이후에도 동일한 절차를 거친다.

5.5 EXECUTE PROCEDURE

STORED PROCEDURE가 값을 리턴하는 경우, 리턴값은 SQLQuery::fields()를 통하여 얻을 수 있다. 이 경우 SQLQuery::fetch()는 호출하지 말아야 한다.

다음은 이에 대한 예이다.

```
CREATE PROCEDURE SP_SUM(N INTEGER) RETURNING INTEGER;
        DEFINE S INTEGER;
```

```
        LET S = 0;
        WHILE N > 0
                LET S = S + N;
                LET N = N – 1;
        END WHILE;
        RETURN S;
END PROCEDURE;

static void StoredProcedure(SQLConnection* pConn)
{
        try
        {
                SQLQuery qry(pConn);
                qry.execute("EXECUTE PROCEDURE SP_SUM(10)");
                cout << "SUM is " << qry.fields()[0]->asInteger() << endl;
        }
        catch (SQLException* e)
        {
                cout << e->getMessage() << endl;
                e->destroy();
        }
}
```

제6장 InterBase

6.1 개요

- 서버버전 : InterBase 6
- 구현 : InterBase API
- System Requirements : InterBase Client(gds32.dll, libgds32.so)
- 드라이버 파일명

  Win32 : DCLInt.dll

  UNIX : DCLInt.so

6.2 연결문자열

- USER
- PASSWORD
- SERVER : 데이터베이스 서버의 호스트명, 생략가능
- DATABASE
- SQL_DIALECT : 1, 2, 3, 기본값 3 (V6)

6.3 데이터 타입

InterBase Internal	SQL::DataType	비고
CHAR(n), VARCHAR(n)	typeText	
SMALLINT	typeInteger	
INTEGER	typeInteger	
DECIMAL(p, s), NUMERIC(p, s)	typeInteger	s == 0
DECIMAL(p, s), NUMERIC(p, s)	typeNumeric	s > 0
FLOAT	typeFloat	
DOUBLE PRECISION	typeFloat	
DATE	typeDate	
TIME	typeTime	
TIMESTAMP	typeTimeStamp	
BLOB SUB_TYPE TEXT	typeLongText	
BLOB	typeLongBinary	

InterBase Server에서 DECIMAL(precision, scale), NUMERIC(precision, scale)은 모두 정수형으로 저장된다. 이것은 precision의 크기에 따라 32bit, 64bit의 크기를 가진다. DDBC Driver에서는 scale의 값이 0 보다 크면 typeNumeric로 판정한다. 64bit의 지원은 InterBase 6 에서부터 가능하다.

6.4 트랜잭션

InterBase 서버에서 DML(Data Manipulation Language)을 사용할 때에는 반드시 트랜잭션이 시작되어 있어야 한다. 예를 들어 트랜잭션이 시작되지 않은 상태에서 “SELECT * FROM ATABLE” 과 같은 SQL을 실행하면 “ATABLE”이란 객체가 없다는 에러를 발생한다.

6.5 CREATE DATABASE

연결문자열은 CREATE DATABASE 문으로 대체될 수 있다. 이 경우 SQLConnection::open은 데이터베이스를 생성하고 연결을 유지한다.

다음은 이에 대한 예이다.

```
SQLConnection conn(“InterBase”);
conn.open("CREATE DATABASE localhost:'C:/TEMP/a.gdb'”
                “ USER 'SYSDBA' PASSWORD 'masterkey'");
conn.execute(“SET TRANSACTION”);

…

conn.execute(“COMMIT”);
conn.close();
```

6.6 EXECUTE PROCEDURE

STORED PROCEDURE가 값을 리턴하는 경우, 리턴값은 SQLQuery::fields()를 통하여 얻을 수 있다. 이 경우 SQLQuery::fetch()는 호출하지 말아야 한다.

다음은 이에 대한 예이다.

```
CREATE PROCEDURE SP_SUM(N INTEGER)
        RETURNS (S INTEGER)
AS
BEGIN
        S = 0;
        WHILE (N > 0) DO
        BEGIN
                S = S + N;
                N = N - 1;
        END
END ^

static void StoredProcedure(SQLConnection* pConn)
{
        try
        {
                SQLQuery q(pConn);
                qry.prepare("EXECUTE PROCEDURE SP_SUM 10");
                qry.execute();
                cout << qry.fields()[0]->name() << " ==> "
                        << qry.fields()[0]->asInteger() << endl;
        }
        catch(Exception* e)
        {
                cout << "Exception: " << e->getMessage() << endl;
                e->destroy();
        }
}
```

제7장 MySQL

7.1 개요

- 서버버전 : MySQL 3.2x, MySQL 4.x
- 구현 : mysqlclient 3.2x
- System Requirements :
- 드라이버 파일명
  Win32 : DCLMy.dll
  UNIX : DCLMy.so

7.2 연결문자열

- USER
- PASSWORD
- SERVER : 데이터베이스 서버의 호스트명
- PORT : TCP 포트
- UNIX_SOCKET
- APPLICATION
- CACHED_RESULT : TRUE/FALSE, 기본값 FALSE

UNIX_SOCKET이 기술되면 SERVER 과 PORT는 생략되어야 한다. UNIX_SOCKET은 UNIX 버전에서만 사용할 수 있다. APPLICATION은 서버에 연결된 어플리케이션의 식별 문자열로 생략 가능하다.

CACHED_RESULT는 SQL 실행 결과의 사용에 있어서 mysql_store_result와 mysql_use_result를 선택한다. CACHED_RESULT가 FALSE이면 SQL::Query는 mysql_use_result를 사용하여 동작하기 때문에 SQL::Connection은 단일 SQL::Query만 사용할 수 있다.

7.3 데이터 타입

MySQL Internal	SQL::DataType	비고
BIT, BOOL, BOOLEAN TINYINT[(M)] SMALLINT[(M)]	typeInteger/typeUInteger	UNSIGNED가 지정될 경우 typeUInteger
MEDIUMINT[(M)] INT[(M)], INTEGER[(M)]	typeInteger/typeUInteger	
BIGINT[(M)]	typeInteger/typeUInteger	
FLOAT(precision) FLOAT[(M,D)]	typeFloat	precision <= 24
FLOAT(precision) DOUBLE[(M,D)] DOUBLE PRECISION[(M,D)]	typeFloat	25 <= precision <= 53
DECIMAL[(M[,D])] DEC[(M[,D])] NUMERIC[(M[,D])] FIXED[(M[,D])]	typeNumeric	
DATE	typeDate	
TIME	typeTime	
DATETIME TIMESTAMP[(M)]	typeTimeStamp	M is 14, 12, 8, 6
YEAR	typeInteger	

CHAR(M) VARCHAR(M) ENUM SET	typeText	
TINYTEXT TEXT	typeText	$2^8 - 1$ $2^{16} - 1$
TINYBLOB BLOB	typeBinary	$2^8 - 1$ $2^{16} - 1$
MEDIUMTEXT LONGTEXT	typeLongText	$2^{24} - 1$ $2^{32} - 1$
MEDIUMBLOB LONGBLOB	typeLongBinary	$2^{24} - 1$ $2^{32} - 1$

TIMESTAMP의 경우 MySQL 4.1 버전 이상에서는 M을 지정할 수 없으며 DATETIME과 동일한 크기를 가진다. TIMESTAMP 는 INSERT, UPDATE에서 자동으로 그 값이 수정된다.

TINYBLOB, TINYTEXT의 최대 크기는 255Bytes이고 BLOB, TEXT의 최대 크기는 65535Bytes이다. 드라이버에서는 이 데이터 타입을 각각 SQL::typeBinary와 SQL::typeText로 설정한다.

7.4 트랜잭션

MySQL 드라이버의 SQL::Connection은 항상 트랜잭션이 가능한 상태를 리턴한다. 그러나, 실제 MySQL 서버는 데이터베이스의 생성 옵션에 따라 트랜잭션이 가능할 수도 그렇지 않을 수도 있다.

7.5 파라미터의 구현

MySQL 4.0 이하의 버전에서는 mysqlclient에서 prepare를 지원하지 않는다. 드라이버는 prepare에서 SQL를 조각내고 execute 직전에 파라미터에 설정된 값을 참고하여 SQL을 완성한다.

제8장 Oracle

8.1 개요

- 서버버전 : Oracle 9.x
- 구현 : OCI(Oracle Call Interface)
- System Requirements : Oracle Net 9.x
- 드라이버 파일명
  Win32 : DCLOra.dll
  UNIX : DCLOra.so

8.2 연결문자열

- USER
- PASSWORD
- DATABASE : TNS Name
- MODE : Operation Mode를 의미하며 생략되거나 SYSDBA, SYSOPER 값을 가질 수 있다.

8.3 데이터 타입

Oracle Internal	DDBC	비고
VARCHAR2, NVARCHAR2	typeText	
CHAR, NCHAR	typeText	
NUMBER	typeNumeric	
DATE	typeTimeStamp	
TIMESTAMP	typeTimeStamp	
TIMESTAMP WITH TIME ZONE	typeTimeStampTz	
TIMESTAMP WITH LOCAL TIME ZONE	typeTimeStampTz	
INTERVAL YEAR TO MONTH	typeIntervalYm	
INTERVAL DAT TO SECOND	typeIntervalDs	
User Defined Type (object type, VARRAY, Nested Table)	N/A	
ROWID, UROWID	typeText	
RAW	typeBinary	
LONG	typeLongText	
LONG RAW	typeLongBinary	
CLOB	typeClob	
BLOB	typeBlob	
BFILE	typeBlob	

8.4 트랜잭션

SQL::Connection::startTrans()의 기본 동작은 “SET TRANSACTION READ WRITE”을 실행한다.

8.5 PLSQL과 RETURNING 절

오라클 드라이버는 PLSQL 블록의 파라미터와 SQL의 RETURNING 절의 아웃풋 파라미터를 완벽하게 지원한다. 다만, 입력 데이터가 없는 아웃풋 전용의 파라미터의 경우 SQLQuery::execute 전에 SQLParam::setDataType()을 호출하여 파라미터의 데이터 타입을 정해 주어야 한다.

다음은 이와 관련한 예이다.

```
static void OutParam(OutputStream& out, SQLConnection* pSQLConn)
{
	SQLQuery q(pSQLConn);

	const char* pszSQL =
		"BEGIN                                                  \n"
		"	SELECT COL_TS INTO :OUT                         \n"
		"	FROM BASIC_TYPES                                \n"
		"	WHERE                                           \n"
		"		COL_NUM90 = :COL_NUM90;                 \n"
		"END;                                                   \n"
		;

	out << "SQL:\n" << pszSQL << "\nRESULT:\n";

	q.prepare(pszSQL);
	q.params()[0].setDataType(SQL::typeTimeStamp);
	q.params()[1].setValue(3);
	q.execute();
	if (q.params()[0].isNull())
		out << "null";
	else
		out << q.params()[0].asString();

	out << "\n\n";

	pszSQL =
		"BEGIN                                                  \n"
		"	SELECT COL_BLOB         INTO :OUT               \n"
		"	FROM BASIC_TYPES                                \n"
		"	WHERE                                           \n"
		"		COL_NUM90 = :COL_NUM90;                 \n"
		"END;                                                   \n"
		;

	out << "SQL:\n" << pszSQL << "\nRESULT:\n";

	q.prepare(pszSQL);
	q.params()[0].setDataType(SQL::typeBlob);
	q.params()[1].setValue(3);
	q.execute();
	if (q.params()[0].isNull())
		out << "null";
	else
		out << q.params()[0].asStringF();

	out << "\n\n";

}
```