Tibero

tbESQL/COBOL 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 tbESQL/COBOL 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 tbESQL/COBOL의 기본 개념과 이를 활용한 프로그램의 개발 방법을 알고자 하는 데이터베이스 관리자(Database Administrator, 이하 DBA) 및 애플리케이션 프로그램 개발자를 대상으로 기술한다.

안내서의 전제 조건

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

- 데이터베이스의 이해
- RDBMS의 이해
- SQL의 이해
- COBOL 프로그래밍의 이해

안내서의 제한 조건

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

참고

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

안내서 구성

Tibero tbESQL/COBOL 안내서는 총 6개의 장으로 이루어져 있다.

각 장의 주요 내용은 다음과 같다.

- 제1장: tbESQL/COBOL 소개

  tbESQL/COBOL의 기본 개념과 구성요소를 기술한다.

- 제2장: 데이터 타입

  tbESQL/COBOL 프로그램에서 사용되는 데이터 타입과 데이터 타입 간의 대응을 기술한다.

- 제3장: 기본 프로그래밍

  tbESQL/COBOL 프로그램의 문법과 실행 과정, 런타임 에러 처리, 그리고 tbESQL/COBOL 문장의 실행과 커서를 기술한다.

- 제4장: 배열 변수

  배열 변수의 기본 개념과 선언 방법, tbESQL/COBOL 문장에서 입/출력 변수로 배열 변수를 사용하는 방법을 기술한다.

- 제5장: tbESQL/COBOL 문장

  tbESQL/COBOL 프로그램에서 데이터베이스 처리를 위해 사용하는 tbESQL/COBOL 문장을 기술한다.

- 제6장: tbESQL/COBOL 프리컴파일러 옵션

  tbESQL/COBOL 프리컴파일러를 동작시킬 때 사용할 수 있는 옵션을 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 애플리케이션 개발자 안내서	각종 애플리케이션 라이브러리를 이용하여 애플리케이션 프로그램을 개발하는 방법을 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 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 tbESQL/COBOL 소개

본 장에서는 tbESQL/COBOL의 기본 개념과 tbESQL/COBOL 프로그래밍을 시작하기 전에 알아야 할 구성요소를 설명한다.

1.1. 개요

tbESQL은 ESQL(Embedded SQL: 내장 SQL)의 사용을 위해 Tibero가 제공하는 인터페이스이다. 일반적으로 프로그래밍 언어는 매우 복잡하고 세밀한 작업을 빠르게 수행할 수 있으며, SQL 문장은 간단한 문법만으로 데이터베이스에 직접적인 작업을 수행할 수 있다.

ESQL은 이러한 프로그래밍 언어의 연산 능력과 SQL의 데이터베이스(Database)를 조작하는 능력을 결합하기 위한 방법이며, ANSI 및 ISO 표준으로 정의되어 있다.

Tibero에서는 애플리케이션 개발에 사용되는 COBOL과 C에 대한 tbESQL 인터페이스를 제공한다. COBOL 프로그래밍 언어를 위한 ESQL 인터페이스를 **tbESQL/COBOL**이라고 부르며, C 프로그래밍 언어에 대한 인터페이스를 tbESQL/C라고 부른다.

참고

tbESQL/C에 대한 내용은 "Tibero tbESQL/C 안내서"를 참고한다.

1.2. 구성요소

1.2.1. tbESQL/COBOL 문장

tbESQL/COBOL 프로그램에는 COBOL 프로그래밍 언어의 소스 코드와 tbESQL/COBOL 문장이 혼합되어 있다. tbESQL/COBOL 프로그램 내에서 SQL 문장의 질의(Query) 등 데이터베이스 처리와 관련된 문장을 tbESQL/COBOL 문장(tbESQL/COBOL Statement)이라고 한다.

tbESQL/COBOL 문장은 일반 SQL 문장과 비슷하지만 다음과 같은 점에서 차이가 난다.

- 항상 **EXEC SQL**로 시작하고 **END EXEC.**로 종료한다.
- 필요한 경우에 입력 변수와 출력 변수를 포함한다.
- 일반 SQL 문장에는 없는 새로운 절을 포함할 수 있다. 예를 들어 SELECT 문장은 INTO 절을 포함할 수 있다.

다음은 tbESQL/COBOL 프로그램에서 UPDATE 문장을 작성한 예이다.

[예 1.1] tbESQL/COBOL에서의 UPDATE 문장

```
EXEC SQL UPDATE EMP SET SALARY = SALARY * 1.05
         WHERE EMPNO = 5
END-EXEC.
```

위의 예에서는 tbESQL/COBOL 문장임을 나타내는 **EXEC SQL**로 시작되고 **END-EXEC.**로 끝난다는 점에서 일반 SQL 문장과 차이가 난다는 것을 알 수 있다.

참고

자세한 내용은 "제5장 tbESQL/COBOL 문장"을 참고한다.

1.2.2. 프로그램 변수

tbESQL/COBOL 프로그램에서 가장 중요한 작업 중에 하나는 프로그램과 데이터베이스 간에 데이터를 전달하는 것이다. 이러한 작업은 주로 프로그램 변수를 이용하여 수행한다. 즉, 프로그램 변수를 이용하여 질의를 하거나 변수의 값을 데이터베이스에 저장할 수 있고, 데이터베이스의 컬럼 값을 프로그램 변수에 저장할 수도 있다. 이러한 작업을 하기 위해서는 Tibero의 데이터 타입과 tbESQL/COBOL 프로그램의 데이터 타입 간의 연관성이 필요하다. 예를 들어 다음의 Tibero의 데이터 타입과 tbESQL/COBOL 프로그램의 데이터 타입은 서로 대응된다.

Tibero의 데이터 타입	tbESQL/COBOL 프로그램의 데이터 타입
NUMBER(p, s)	PIC S9(n)

NUMBER 타입에서 p는 정밀도(Precision), s는 스케일(Scale)을 의미한다.

참고

Tibero의 데이터 타입과 tbESQL/COBOL 프로그램의 데이터 타입 간의 대응에 대해서는 "2.3.1. 데이터 타입 대응"을 참고한다.

프로그램 변수의 선언

tbESQL/COBOL 프로그램에서의 변수는 COBOL 프로그램의 변수와 거의 동일하게 선언되고 사용된다. 데이터베이스 작업과 관련되지 않는 변수는 COBOL 프로그램에서 사용되는 것과 같이 제약 없이 사용할 수 있다.

다음은 tbESQL/COBOL에서 프로그램 변수를 선언한 예이다.

[예 1.2] tbESQL/COBOL에서의 프로그램 변수의 선언

```
01 OPERATION PIC X(20) VARYING OCCURS 5 TIMES.
01 TELLER.
   03 EMPNO  PIC S9(7).
```

```
   03 ENAME  PIC X(10).
   03 SALARY PIC S9(5).
01 CNT PIC S9(9).
```

위의 예에서 VARYING 키워드로 선언한 VARCHAR 타입은 tbESQL/COBOL 프로그램 내에서만 사용할 수 있는 타입이며, 프리컴파일 과정을 거쳐 COBOL 프로그래밍 언어의 데이터 타입으로 변환된다.

입/출력 변수

데이터베이스 작업과 관련된 변수는 다음과 같이 두 가지로 구분된다.

- 입력 변수

  tbESQL/COBOL 문장을 통해 컬럼의 값을 삽입, 갱신, 삭제할 때 데이터의 값을 설정하기 위한 변수이다.

  입력 변수는 다음과 같은 특징이 있다.

  – 입력 변수를 사용할 때는 반드시 변수 앞에 **콜론(:)**을 붙여야 한다.

  – 입력 변수는 SELECT, INSERT, UPDATE, DELETE 문장과 WHERE 절과 SET 절 등의 컬럼 값의 위치에 사용될 수 있다.

  – 입력 변수는 테이블 이름이나 컬럼 이름의 위치에는 사용 될 수 없다.

- 출력 변수

  tbESQL/COBOL 문장의 질의 수행 결과로 반환된 값을 저장하기 위한 변수이다.

  출력 변수는 다음과 같은 특징이 있다.

  – 입력 변수와 마찬가지로 변수 앞에 반드시 **콜론(:)**을 붙여야 한다.

  – 출력 변수는 SELECT 문장의 INTO 절에 사용될 수 있다.

  – INTO 절에는 출력 변수와 함께 INDICATOR 키워드와 지시자 변수가 올 수 있다.

다음은 WHERE 절에 사용된 입력 변수 EMPNO와 INTO 절에 사용된 출력 변수 ENAME, SALARY, ADDR를 사용한 예이다.

[예 1.3] 입/출력 변수의 사용

```
EXEC SQL SELECT ENAME, SALARY, ADDR
         INTO :ENAME, :SALARY, :ADDR
         FROM EMP
         WHERE EMPNO = :EMPNO
END-EXEC.
```

위의 문장을 실행하기 전에 EMPNO 값을 설정하는 코드가 와야 하며, 또한 문장이 실행된 후에는 반환된 ENAME, SALARY, ADDR 컬럼 값에 대한 적절한 작업이 진행되어야 한다.

1.2.3. 구조체 및 배열 변수

구조체와 배열 변수는 동시에 여러 개의 입/출력 변수를 처리하는 자료 구조이다.

구조체

tbESQL/COBOL 프로그램에서는 보통 동시에 여러 개의 입력 변수나 출력 변수가 사용된다. [예 1.3]의 경우에도 SELECT 문장의 INTO 절에서 세 개의 출력 변수(ENAME, SALARY, ADDR)가 사용되었다. 또한 INSERT 문장을 사용할 경우에는 여러 개의 입력 변수가 사용될 수 있다.

이렇게 여러 개의 입력 변수나 출력 변수가 사용될 경우 tbESQL/COBOL 프로그램에서도 COBOL 프로그래밍 언어에서처럼 여러 개의 변수를 묶어 하나의 구조체로 사용할 수 있다.

참고

입/출력 구조체 변수에 대한 자세한 내용은 “2.3.6. 구조체”를 참고한다.

배열 변수

SELECT 문장을 실행한 결과의 로우 개수는 대개의 경우 하나 이상이며, INSERT 문장을 실행할 때에도 보통 하나 이상의 로우를 삽입하게 된다. 이때 각각의 로우에 대해 개별적으로 SQL 문장을 여러 번 실행하지 않고, 출력 변수나 입력 변수를 배열로 선언하여 SQL 문장을 한 번만 실행할 수도 있다.

출력 변수나 입력 변수를 배열로 선언한 경우 각각을 출력 배열 변수, 입력 배열 변수라고 부른다. 또한 이 두 가지 변수를 한꺼번에 지칭할 때는 입/출력 배열 변수라고 한다.

출력 배열 변수를 이용하여 SELECT 문장을 실행하는 경우 각 결과 로우는 결과 로우의 개수와 동일한 크기를 갖는 출력 배열 변수에 저장된다.

tbESQL/COBOL 프로그램에서는 하나 이상의 결과 로우가 반환되는 SELECT 문장에 출력 배열 변수나 커서를 사용하지 않으면 에러를 반환한다. 구조체도 배열 변수로 선언하여 사용할 수 있으며, 이를 구조체 배열 변수라고 부른다.

참고

입/출력 배열 변수와 구조체 배열 변수에 대한 자세한 내용은 “제4장 배열 변수”를 참고한다.

1.2.4. 커서

커서는 SELECT 문장을 실행한 결과로 반환된 다수의 로우 각각에 차례대로 액세스하는 데이터 구조이다.

SELECT 문장을 실행한 결과로 반환되는 로우의 개수는 기본 키에 대한 질의가 아니라면 대부분 하나 이상이다. 또한 반환되는 로우의 개수를 미리 알 수 없는 경우가 많기 때문에 SELECT 문장의 INTO 절에 변수를 하나만 명시해서는 모든 로우의 데이터를 저장할 수 없다.

이러한 경우에 프로그램을 보다 간편하고 편리하게 작성하기 위해 앞 절에서 설명한 배열 변수를 사용할 수도 있지만 커서를 사용할 수도 있다.

커서를 사용하는 방법 및 순서는 다음과 같다.

1. DECLARE CURSOR를 이용하여 커서를 선언한다.
2. OPEN을 실행하여 커서를 연다. 커서를 열면, 연관된 SQL 문장이 실행되고 질의의 결과가 반환된다.
3. FETCH를 실행한다. FETCH를 실행할 때마다 하나 또는 그 이상의 결과 로우를 얻을 수 있다. 커서는 항상 현재 처리 중인 로우를 가리킨다.
4. 모든 결과 로우에 대한 FETCH를 실행한 후에는 CLOSE를 실행하여 커서를 닫는다.

참고

자세한 내용은 "3.4. 커서"를 참고한다.

1.2.5. 프리컴파일러

tbESQL/COBOL 프로그램 내에는 COBOL 프로그래밍 언어와 tbESQL/COBOL 문장이 함께 포함되어 있다. tbESQL/COBOL 문장은 COBOL 프로그래밍 언어의 문법을 따르지 않기 때문에, tbESQL/COBOL 프로그램을 컴파일하기 전에 적절한 처리를 해야 한다.

이러한 컴파일 이전의 처리 과정을 **프리컴파일**(Precompile)이라고 하며, 그때 사용하는 유틸리티를 **프리컴파일러**(Precompiler)라고 부른다.

프리컴파일러는 tbESQL/COBOL 프로그램에 포함된 tbESQL/COBOL 문장을 tbESQL/COBOL 라이브러리 함수로 호출할 수 있는 COBOL 프로그래밍 언어의 소스 코드로 변환해준다.

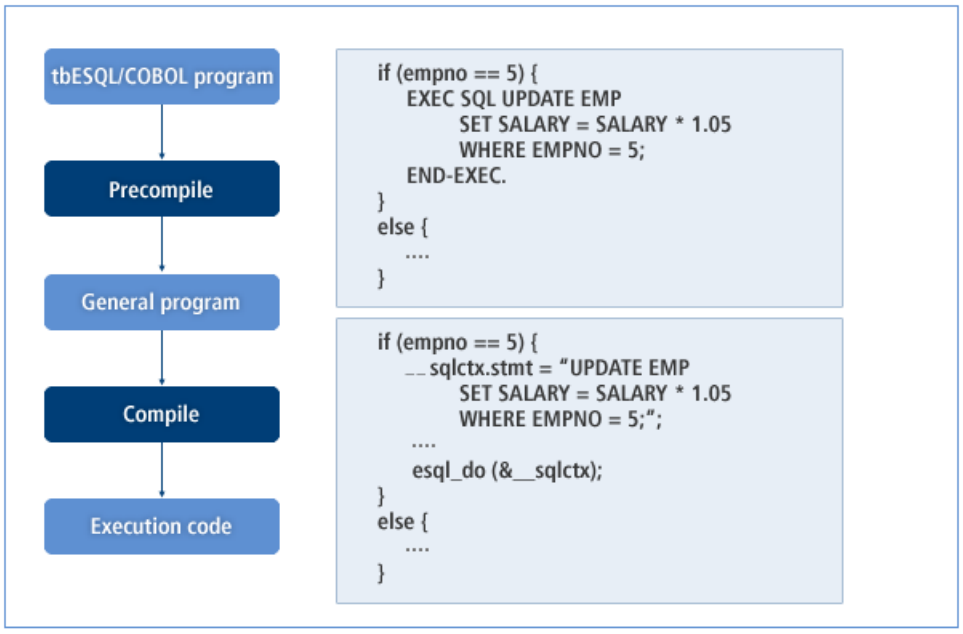
이러한 과정을 위해서 tbESQL/COBOL에서는 다음과 같은 함수를 정의하고 있다.

```
esql_do( )
```

이 함수의 파라미터로는 SQL 문장과 입/출력 변수의 정보가 포함된다.

다음은 tbESQL/COBOL 프로그램의 컴파일 과정을 나타내는 그림이다.

[그림 1.1] tbESQL/COBOL 프로그램 컴파일 과정



tbESQL/COBOL 프로그램의 컴파일 과정은 크게 두 가지 과정으로 나눌 수 있다.

- 프리컴파일(Precompile)

  tbESQL/COBOL 프로그램을 작성하여 프리컴파일 과정을 거치고 나면 COBOL 프로그래밍 언어로만 구성된 소스 코드가 생성된다. 생성된 소스 코드는 **.cob** 확장자를 가진 파일의 형태로 저장된다.

- 컴파일(Compile)

  프리컴파일 과정을 거친 프로그램은 다시 컴파일 과정을 거치게 되고 최종적으로 실행 파일이 생성된다.

제2장 데이터 타입

본 장에서는 tbESQL/COBOL 프로그램에서 사용하는 데이터 타입을 알아보고, 데이터 타입 간의 대응을 설명한다.

2.1. 개요

tbESQL/COBOL 프로그램에서 데이터 타입은 tbESQL/COBOL 문장에 값을 입력하고, 질의 결과를 얻어오는데 사용된다.

tbESQL/COBOL은 다음의 두 가지 타입을 지원한다.

- Tibero 데이터 타입

  데이터베이스에 저장된 데이터에 접근할 때 사용한다.

- tbESQL/COBOL 데이터 타입

  애플리케이션 프로그램에서 데이터를 조작할 때 사용한다.

2.2. Tibero 데이터 타입

본 절에서는 Tibero 데이터 타입을 간략히 설명한다.

Tibero에서 기본으로 제공하는 데이터 타입은 데이터베이스의 스키마 객체를 생성하는 데 사용하는 것으로 tbESQL/COBOL 프로그램 내에서도 모든 데이터 타입에 대응되는 변수를 사용할 수 있다.

다음은 Tibero의 데이터 타입이다.

구분	데이터 타입	설명
문자형	CHAR, VARCHAR, RAW	문자열을 표현하는 데이터 타입이다.
숫자형	NUMBER, INTEGER, FLOAT	정수나 실수의 숫자를 저장하는 데이터 타입이다.
날짜형	DATE, TIME, TIMESTAMP	시간이나 날짜를 저장하는 데이터 타입이다.
대용량 객체형	BLOB, CLOB	LOB 타입을 의미한다. 다른 데이터 타입이 지원하는 최대 길이(8KB 이하)보다 훨씬 큰 길이를 가질 수 있는 객체이다. 4GB까지 가능하다.
내재형	ROWID	사용자가 명시적으로 선언하지 않아도 Tibero가 자동으로 삽입되는 로우마다 포함하는 컬럼의 타입이다.

참고

자세한 내용은 "Tibero SQL 참조 안내서"를 참고한다.

다음은 각 데이터 타입에 대한 세부 설명이다.

데이터 타입	설명
CHAR	일반 문자열을 저장하는 데이터 타입이다. (예: CHAR(10))
VARCHAR	일반 문자열을 저장하는 데이터 타입이다. (예: VARCHAR(10))
RAW	임의의 바이너리 데이터를 저장하는 데이터 타입이다. (예: RAW(10))
NUMBER	정수 또는 실수를 저장하는 타입이다. NUMBER 타입을 선언할 때 정밀도와 스케일 을 함께 선언할 수 있다. – 정밀도 : 데이터 값의 전체 자릿수 – 스케일 : 소수점 이하 자릿수
INTEGER FLOAT	기본적으로는 NUMBER 타입이다. 단, NUMBER 타입과는 다르게 정밀도와 스케일을 선언할 때 범위에 한계를 둔다. NUMBER 타입의 값은 Tibero에서 가변 길이로 저장되며, 실제 값과 정밀도, 스케일에 따라 그 길이가 달라진다.
DATE TIME TIMESTAMP	특정 날짜와 시간을 나타내는 데이터 타입이다. – DATE : 특정 날짜 – TIME : 특정 시간 – TIMESTAMP : 특정 날짜와 시간
BLOB	임의의 바이너리 데이터를 데이터베이스에 저장하는 데이터 타입이다. 한 테이블의 여러 컬럼에 선언할 수 있다.
CLOB	읽을 수 있는 문자열을 데이터베이스에 저장하는 데이터 타입이다. 한 테이블의 여러 컬럼에 선언할 수 있다.
ROWID	시스템이 각 로우마다 자동으로 부여하는 데이터 타입이다. 각 로우가 저장된 물리적인 위치를 포함한다.

2.3. tbESQL/COBOL 데이터 타입

본 절에서는 tbESQL/COBOL의 데이터 타입을 설명한다.

2.3.1. 데이터 타입 대응

Tibero에서 제공하는 데이터 타입을 tbESQL/COBOL 프로그램에서 그대로 사용할 수는 없다.

tbESQL/COBOL에는 Tibero의 각 데이터 타입에 대응되는 tbESQL/COBOL의 데이터 타입이 정의되어 있다. tbESQL/COBOL의 데이터 타입은 대체로 COBOL 프로그래밍 언어의 데이터 타입과 동일하다. 또한 각 Tibero의 데이터 타입에 대응되는 tbESQL/COBOL의 데이터 타입은 하나 이상일 수도 있다.

다음은 Tibero의 데이터 타입에 대응되는 tbESQL/COBOL의 데이터 타입이다.

Tibero의 데이터 타입	**tbESQL/COBOL의 데이터 타입**	**설명**
CHAR, VARCHAR	PIC X(n), PIC X(n) VARYING	길이 n의 문자열
RAW	PIC X(n) VARYING	길이 n의 바이너리 데이터
NUMBER	PIC S9(n)	정수 데이터
	PIC S9(n)V9(n)	실수 데이터 (데이터가 삽입될 때 이미 정해진 컬럼의 정밀도 및 스케일을 초과할 수 있다.)
DATE, TIME, TIMESTAMP	PIC X(n), PIC X(n) VARYING	길이 n의 문자열로 변환
ROWID	PIC X(n), PIC X(n) VARYING	길이 n의 문자열로 변환

tbESQL/COBOL의 데이터 타입 중 VARYING 키워드를 사용해서 선언하는 **VARCHAR 타입**은 Tibero의 데이터 타입 중에 VARCHAR 타입을 비롯한 여러 가지 타입에 대응하기 위해 새롭게 정의된 타입이다.

tbESQL/COBOL 프로그램에서는 각 데이터 타입 간의 변환을 지원한다. 예를 들어 VARCHAR 타입의 문자열이 정수를 표현하고 있는 내용이라면, 그 값을 PIC S9(n) 타입의 변수에 저장할 수 있다. 또한 실제로 DATE, TIME, TIMESTAMP 타입과 ROWID 타입에 바로 대응되는 타입은 없으며, 항상 변환 과정을 거쳐서 저장해야 한다.

참고

VARCHAR 타입의 자세한 내용은 “2.3.5. VARCHAR”를 참고한다.

2.3.2. 데이터 타입 변환

tbESQL/COBOL 프로그램에서는 Tibero 데이터 타입 각각에 대응되는 타입 이외에 다른 데이터 타입을 사용할 수 있다. 예를 들면 데이터베이스의 NUMBER 타입의 컬럼 값을 저장하기 위해 tbESQL/COBOL 프로그램에서는 출력 변수로 VARCHAR 타입을 사용할 수 있다. 이와 반대로 NUMBER 타입 프로그램 변수의 값을 VARCHAR 타입의 컬럼에 저장할 수도 있다.

tbESQL/COBOL 프로그램을 프리컴파일하면 입/출력 변수의 데이터 타입이 프로그램 내에 함께 포함된다. 이렇게 포함된 데이터 타입을 기준으로 데이터의 값이 입/출력될 때 컬럼 타입과 비교하여 필요한 경우에는 데이터 타입의 변환을 수행한다. 만약 데이터 타입의 변환이 불가능한 경우에는 에러를 반환한다.

변환 가능한 데이터 타입

다음은 Tibero 데이터 타입으로부터 변환 가능한 tbESQL/COBOL 데이터 타입이다. “2.3.1. 데이터 타입 대응”의 일부 내용이 포함되어 있다.

Tibero 의 데이터 타입	tbESQL/COBOL의 데이터 타입	설명
NUMBER	PIC S9(n)	정수 데이터
	PIC S9(n)V9(n)	실수 데이터
CHAR, VARCHAR	PIC X(n), PIC X(n) VARYING	문자열 데이터(실수를 포함한다.)
DATE, TIME, TIMESTAMP	PIC X(n), PIC X(n) VARYING	날짜형 데이터
ROWID	PIC X(n), PIC X(n) VARYING	ROWID 데이터

tbESQL/COBOL 데이터 타입으로부터 Tibero 데이터 타입으로 변환할 때에도 위의 변환 관계가 적용된다. 예를 들어 tbESQL/COBOL 프로그램의 PIC S9(n) 타입의 변수 값을 VARCHAR 타입 컬럼에 저장할 수 있으며, VARYING 키워드를 통해 선언하는VARCHAR 타입 값을 이용하여 ROWID에 대한 질의를 수행할 수 있다.

다음은 데이터 타입의 변환을 수행하는 예이다.

[예 2.1] 데이터 타입의 변환

```
01 SAL-STR  PIC X(8)  VARYING.
01 EMP-DATE PIC X(20) VARYING.
...
EXEC SQL SELECT SALARY, EMP_DATE
          INTO :SAL-STR, :EMP-DATE
          FROM EMP
          WHERE EMPNO = 20
END-EXEC.

DISPLAY 'SALARY = ' SAL-STR-ARR IN SAL-STR.
DISPLAY 'EMP-DATE = ' EMP-DATE-ARR IN EMP-DATE.
```

위의 [예 2.1]을 실행하면 다음과 같은 내용이 출력된다.

```
SALARY = 35000
EMP-DATE =2001-12-01
```

데이터 타입을 변환할 때에는 데이터 값의 범위와 내용에 유의해야 한다. 데이터 값의 범위는 그 값을 저장할 장소가 충분히 포함할 수 있는 한도 내이어야 한다.

예를 들면 tbESQL/COBOL 프로그램의 PIC S9(3) 타입의 변수에 그 범위를 넘어서는 값인 문자열 "327680"을 변환하거나, 마찬가지로 VARCHAR(3)의 타입을 갖는 컬럼에 그 범위를 넘어서는 값인 65535를 변환할 수 없다. 데이터의 내용은 데이터베이스 컬럼에 저장된 "ABCDE" 문자열을 프로그램 PIC S9(3) 타입의 변수에 변환할 수 없으며, 프로그램 변수에 저장된 "가나다" 문자열을 DATE 타입의 컬럼에 변환할 수 없다.

내장 함수를 이용한 데이터 타입 변환

Tibero **내장 함수**를 이용하여 데이터 타입의 변환을 실행할 수도 있다. 그러한 함수로는 **TO_CHAR**, **TO_DATE**, **TO_NUMBER** 등이 있다.

다음은 TO_CHAR 함수를 이용하여 실수 데이터를 문자 데이터로 변환하여 출력하는 예이다.

[예 2.2] TO_CHAR 함수를 이용한 데이터 타입 변환

```
01 SAL-STR PIC X(8) VARYING.
...
EXEC SQL SELECT TO_CHAR(SALARY, '$99,999.99')... ① ...
         INTO :SAL-STR... ② ...
         FROM EMP
         WHERE EMPNO = 20
END-EXEC.

DISPLAY 'SALARY = ' SAL-STR-ARR IN SAL-STR.... ③ ...
```

① 실수 데이터를 담고 있는 변수 SALARY를 TO_CHAR 함수를 통해 형식 문자열('$99,999.99')을 지정하여 문자 데이터로 변환한다.

② VARCHAR 타입 변수 SAL-STR에 저장한다.

③ DISPLAY 문을 통해 SAL-STR 변수를 출력한다.

위의 [예 2.2]을 실행하면 다음과 같은 내용이 출력된다.

```
SALARY = $35,000
```

참고

데이터 타입을 변환하는 내장 함수에 대한 자세한 내용은 "Tibero SQL 참조 안내서"를 참고한다.

2.3.3. 데이터 변수 사용

COBOL 프로그래밍 언어의 변수와는 달리 tbESQL/COBOL 프로그램에서 데이터베이스 작업과 관련된 변수는 모두 **DECLARE 영역** 내에 선언되어야 한다.

다음은 DECLARE 영역 내에 선언된 변수의 예이다.

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 OPERATION PIC X(20) VARYING OCCURS 5 TIMES.
01 TELLER.
   03 EMPNO  PIC S9(7).
   03 ENAME  PIC X(10).
   03 SALARY PIC S9(5).
01 CNT PIC S9(9).
EXEC SQL END DECLARE SECTION END-EXEC.
```

위의 예에서 알 수 있듯이 DECLARE 영역은 **'EXEC SQL BEGIN DECLARE SECTION END-EXEC.'**로 시작하고 **'EXEC SQL END DECLARE SECTION END-EXEC.'**로 끝난다.

VARCHAR 타입은 일반적인 CHAR 배열 타입과 유사하게 선언한다. CHAR 타입과는 달리 배열이 아닌 형태로는 선언이 불가능하다. 프리컴파일 과정을 거치면 VARCHAR 타입은 tbESQL/COBOL에서 정의한 구조체 타입으로 변환된다.

DECLARE 영역 내에 선언된 변수는 COBOL 프로그래밍 언어의 변수와 동일한 방법으로 프로그램 내에서 사용된다. 하지만 tbESQL/COBOL 문장 내에서의 변수는 tbESQL/COBOL 문장과의 구별을 위하여 반드시 콜론(:) 뒤에 와야 한다. 이렇게 콜론(:) 뒤에 사용된 tbESQL/COBOL 문장 내의 변수를 **입/출력 변수**라고 부른다.

다음은 tbESQL/COBOL 문장 내에서 사용된 입력 변수와 출력 변수에 대한 예이다.

```
MOVE 20 TO EMPNO.
EXEC SQL SELECT ENAME, SALARY, ADDR
         INTO :ENAME, :SALARY, :ADDR
         FROM EMP
         WHERE EMPNO = :EMPNO
END-EXEC.

DISPLAY 'SALARY = ' SALARY.
```

위의 예에서는 SELECT 문장을 실행하기 위해 먼저 입력 변수 EMPNO의 값을 읽어와 tbESQL/COBOL 문장을 완성한다. 그리고 나서 SELECT 문장을 실행하고 실행 결과로 반환된 로우의 각 컬럼 값이 출력 변수 ENAME, SALARY, ADDR에 할당된다. 출력 변수 ENAME, SALARY, ADDR은 COBOL 프로그래밍 언어의 변수와 마찬가지로 사용될 수 있다.

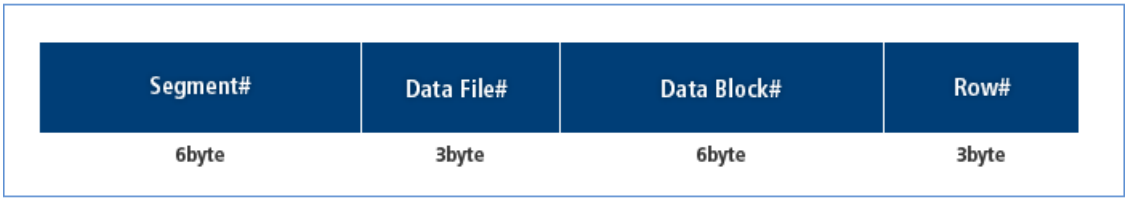
만약 SELECT 문장의 실행 결과로 반환된 로우가 없거나 둘 이상의 로우가 반환되면 에러가 발생한다. 이러한 경우 에러를 처리하는 루틴이 미리 정의되어 있으면 그 루틴을 실행하게 되고, 그렇지 않으면 프로그램을 종료한다.

2.3.4. ROWID

ROWID 타입은 로우의 물리적인 위치 정보를 포함하는 데이터 타입이다. tbESQL/COBOL에서는 **ROWID**를 위한 별도의 데이터 타입을 제공하지 않는다. **PIC X(n)** 타입 또는 VARCHAR(VARYING) 타입을 이용해 데이터 타입을 변환하여 사용해야 한다.

ROWID 타입의 값은 다음의 그림에서처럼 4부분으로 구성된다.

[그림 2.1] ROWID 구성



문자열 변수에 저장되는 ROWID 값은 전체 18bytes를 가지므로, 문자열 변수의 길이는 NULL 값을 포함하여 최소한 19bytes가 되어야 한다. Tibero 의 데이터베이스에서는 다른 형태로 저장된다.

참고

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

다음은 ROWID 타입을 사용하는 예이다.

[예 2.3] ROWID 타입의 사용

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 ROWID PIC X(20).
EXEC SQL END DECLARE SECTION END-EXEC.
...
EXEC SQL SELECT ROWID, ...
         INTO :ROWID, ...
         FROM EMP
         WHERE EMPNO = :EMPNO
END-EXEC.

DISPLAY 'rowid = ' ROWID.
```

ROWID의 값은 이처럼 출력 변수로 사용될 뿐만 아니라 SELECT 문장의 WHERE 절이나 INSERT 문장에서 입력 변수로 사용될 수도 있다.

위의 [예 2.3]을 실행하면 다음과 같은 내용이 출력된다.

```
rowid = AAAABkAAUAAAAD6AAA
```

2.3.5. VARCHAR

VARCHAR 타입은 Tibero의 데이터 타입의 VARCHAR 타입을 tbESQL/COBOL 프로그램에서 사용하기 위해 새롭게 정의한 데이터 타입이다. VARCHAR 타입은 RAW, DATE, ROWID 타입에도 대응하여 사용할 수 있다.

VARCHAR 타입 변수 선언

VARCHAR 타입 변수 선언은 COBOL 프로그래밍 언어에서 PIC X(n) 타입의 배열 변수를 선언하는 것과 동일한데, 마지막에 VARYING 키워드를 삽입한다. PIC X(n)과 마찬가지로 문자열의 최대 크기를 반드시 지정해 주어야 한다.

다음은 VARCHAR 타입 변수를 선언하는 예이다.

[예 2.4] VARCHAR 타입 변수 선언

```
01 USERNAME PIC X(16) VARYING.
```

위의 [예 2.4]에서 선언된 VARCHAR 타입 변수는 프리컴파일러를 통하여 다음과 같은 구조체 타입의 변수로 변환된다.

[예 2.5] VARCHAR 타입이 변환된 구조체

```
01 USERNAME.
    03 USERNAME-LEN PIC S9(4) COMP-5.
    03 USERNAME-ARR PIC X(9).
```

VARCHAR 타입 변수의 참조와 일관성

VARCHAR 타입 변수를 선언할 때를 제외하고, VARCHAR 타입 변수를 사용하기 위해서는 프리컴파일러가 변환한 구조체의 문법을 따라야 한다.

예를 들어 위의 [예 2.4]에서 선언한 USERNAME을 출력하고자 한다면, 프리컴파일러에 의해 변환된 [예 2.5]의 구조체를 출력하는 문법에 맞춰 다음과 같이 코드를 작성한다.

```
DISPLAY USERNAME-ARR.
또는
DISPLAY USERNAME-ARR IN USERNAME.
```

VARCHAR 구조체 내의 변수 LEN과 ARR은 입력 변수일 때 또는 출력 변수일 때 상관없이 항상 일관성을 유지해야 한다. 즉, 다음과 같은 조건을 만족해야 한다.

[예 2.6] VARCHAR 변수의 일관성

```
LENGTH(USERNAME-ARR) EQUALS USERNAME-LEN
```

VARCHAR 타입 변수가 입력 변수로 사용되었는지, 출력 변수로 사용되었는지에 따라서 위의 조건을 만족시키기 위한 방법은 다르다.

VARCHAR 타입 변수	설명
입력 변수	입력 변수로 사용된 경우 tbESQL/COBOL 프로그램 내에 조건을 유지시키는 코드를 작성해야 한다. 조건을 만족하지 않는다면 tbESQL/COBOL 라이브러리에서는 LEN 변수를 우선적으로 참조한다.
출력 변수	출력 변수로 사용된 경우 tbESQL/COBOL 라이브러리 내에서 자동으로 조건을 유지시킨다.

NULL 값의 처리

VARCHAR 타입 변수의 값이 NULL인 경우에는 프리컴파일러를 통해 변환된 구조체 내의 변수 ARR과 LEN의 값은 다음과 같다.

```
ARR EQUAL Z""
LEN SQUAL 0
```

VARCHAR 타입 변수의 값이 **NULL**일 때 VARCHAR 타입 변수가 입력 변수로 사용되었는지, 출력 변수로 사용되었는지에 따라 변환된 구조체 내의 멤버 변수 ARR과 LEN의 값은 다르다.

VARCHAR 타입 변수	설명
입력 변수	입력 변수인 경우에는 LEN 변수에 0을 할당하는 코드를 작성해야 한다. 만약 ARR 문자열의 길이가 0이 아니더라도 LEN 변수를 먼저 참조하므로 NULL로 인식한다.
출력 변수	출력 변수인 경우에는 ARR과 LEN의 값을 tbESQL/COBOL 라이브러리에서 자동으로 설정한다.

다음은 VARCHAR 타입의 출력 변수 ADDR의 값으로 NULL이 반환되었는지 검토하여 출력하는 예이다.

```
01 ADDR PIC X(32) VARYING.
...
EXEC SQL SELECT ADDR INTO :ADDR
         FROM EMP
         WHERE EMPNO = 20
END-EXEC.

IF (ADDR-LEN EQUALS 0)
   DISPLAY 'ADDR = NULL'
ELSE
```

```
   DISPLAY 'ADDR = ' ADDR-ARR
END-IF.
```

2.3.6. 구조체

tbESQL/COBOL 프로그램에서도 COBOL 프로그래밍 언어의 **구조체**를 사용할 수 있다.

tbESQL/COBOL 프로그램의 SELECT 문장에서는 질의 결과로 반환되는 컬럼의 개수만큼 INTO 절에 출력 변수를 명시해야 한다. 이런 경우에 INTO 절에 명시될 다수의 출력 변수를 한데 묶어 구조체를 만들고, INTO 절에 이 구조체 변수 하나만 명시해 프로그램을 간소화할 수 있다.

구조체 변수를 사용할 때에 유의할 점은 SELECT 문장의 결과 로우 내의 컬럼의 순서와 구조체 변수 내의 변수의 순서가 같아야 한다는 점이다.

다음은 3개의 출력 변수를 포함하는 구조체를 사용한 예이다.

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 EMP.
   03 ENAME  PIC X(20) VARYING.
   03 SALARY PIC S9(7) COMP-5.
   03 ADDR   PIC X(32) VARYING.
...
EXEC SQL END DECLARE SECTION END-EXEC.
...
EXEC SQL SELECT ENAME, SALARY, ADDR
         INTO :EMP
         FROM EMP
         WHERE EMPNO = :EMPNO
END-EXEC.
```

구조체 변수는 INSERT 문장 등에서 입력 변수로 사용될 수도 있다. 이때에도 출력 변수와 동일하게 하나의 구조체 변수만 사용할 수 있으며, 구조체 내부에 삽입하려는 컬럼과 같은 순서로 변수가 정의되어 있어야 한다.

다음은 3개의 컬럼에 값을 삽입하는 예이다.

```
MOVE Z"Smith" TO ENAME-ARR IN ENAME IN EMP.
MOVE 35000 TO SALARY IN EMP.
MOVE Z"Los Angeles" TO ADDR-ARR IN ADDR IN EMP.

EXEC SQL INSERT INTO EMP(ENAME, SALARY, ADDR)
         VALUES (:EMP)
END-EXEC.
```

2.3.7. 지시자

일반 프로그램과 달리 tbESQL/COBOL 프로그램에서만 사용되는 변수로 **지시자**(INDICATOR) 변수가 있다. tbESQL/COBOL 문장을 통해 데이터베이스와 tbESQL/COBOL 프로그램 간에 데이터를 주고 받을 때 지시자 변수는 전달된 데이터에 대한 정보를 저장하고 있다.

지시자 변수의 선언

지시자 변수는 PIC S9(4) COMP-5 타입을 가지며, 반드시 **DECLARE 영역** 안에 선언되어야 한다.

SELECT 문장에 사용된 지시자 변수는 INTO 절에서 INDICATOR 키워드와 콜론(:) 다음에 오거나, INDICATOR 없이 데이터 변수 바로 뒤에 콜론(:)과 함께 올 수도 있다.

다음은 SELECT 문장에서 출력 변수와 지시자 변수가 사용된 예이다.

[예 2.7] 출력 변수와 지시자 변수

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 IND-ENAME PIC S9(4) COMP-5.
01 IND-ADDR  PIC S9(4) COMP-5.
...
EXEC SQL END DECLARE SECTION END-EXEC.
...
EXEC SQL SELECT ENAME, ADDR
         INTO :ENAME INDICATOR :IND-ENAME,
              :ADDR INDICATOR :IND-ADDR
         FROM EMP
         WHERE EMPNO = :EMPNO
END-EXEC.
```

위의 예에서는 출력 변수 ENAME에 대응되는 지시자 변수로 IND_ENAME이 사용되었고, 출력 변수 ADDR에 대응되는 지시자 변수로는 IND-ADDR가 사용되었다.

INDICATOR 키워드를 명시하지 않고 다음과 같이 작성해도 위의 [예 2.7] 문장과 동일한 의미를 갖는다.

```
EXEC SQL SELECT ENAME, ADDR
         INTO :ENAME :IND-ENAME,:ADDR :IND-ADDR
FROM EMP
         WHERE EMPNO = :EMPNO
END-EXEC.
```

다음은 출력 변수와 함께 사용된 지시자 변수 값의 의미를 정리한 표이다. tbESQL/COBOL 프로그램에서는 필요한 경우 지시자 변수의 값을 검토하여 그 값에 따른 처리를 해야 한다.

지시자 변수의 값	설명
0	데이터 값이 성공적으로 저장되었다.
-1	데이터 값이 NULL이다.
> 0	문자열 데이터 변수에 저장된 값이 잘린(truncated) 값이다. 지시자 변수에 주어진 값은 실제 데이터베이스에 저장된 문자열의 길이이다.

다음은 INSERT 문장에서 입력 변수와 함께 지시자 변수를 사용한 예이다.

[예 2.8] 입력 변수와 지시자 변수

```
01 ENAME PIC X(24) VARYING.
01 ADDR  PIC X(36) VARYING.

01 IND-ADDR PIC S9(4) COMP-5.
01 EMPNO    PIC S9(9).
...

MOVE Z"Peter" TO ENAME-ARR.
MOVE Z"New York" TO ADDR-ARR.

MOVE -1 TO IND-ADDR.
MOVE 25 TO EMPNO.

EXEC SQL INSERT INTO EMP (ENAME, ADDR, EMPNO)
         VALUES (:ENAME, :ADDR INDICATOR :IND-ADDR, :EMPNO) END-EXEC.
```

위의 예에서는 입력 변수로 ENAME, ADDR, EMPNO가 사용되었으며, 입력 변수 ADDR에 대응되는 지시자 변수로 IND-ADDR이 사용되었다.

INDICATOR 키워드를 명시하지 않고 다음과 같이 작성해도 위의 [예 2.8] 문장과 동일한 의미를 갖는다.

```
EXEC SQL INSERT INTO EMP (ENAME, ADDR, EMPNO)
          VALUES (:ENAME,:ADDR:IND-ADDR, :EMPNO) END-EXEC.
```

위의 예에서는 INDICATOR 키워드를 생략하고 입력 변수 ADDR 뒤에 지시자 변수 IND-ADDR을 바로 붙여서 명시하였다.

지시자 변수 값이 -1인 경우에는 입력 변수의 값이 NULL이라는 의미이다. 이때 입력 변수에 저장된 실제 값은 무시된다. 따라서 지시자 변수의 값이 -1인 경우 앞의 INSERT 문장은 다음과 같이 고쳐 써도 된다.

```
EXEC SQL INSERT INTO EMP (ENAME, ADDR, EMPNO)
          VALUES (:ENAME, NULL, :EMPNO) END-EXEC.
```

지시자 변수 값이 -1인 경우는 입력 변수 ADDR의 값이 NULL이라는 것이므로 입력 변수와 지시자 변수를 명시할 필요 없이 NULL만 명시해도 된다.

다음은 입력 변수와 함께 사용된 지시자 변수 값의 의미를 정리한 표이다.

지시자 변수의 값	설명
-1	데이터 값이 NULL이다.
>= 0	입력 변수에 저장된 값을 그대로 사용한다.

지시자 변수를 사용하지 않고 tbESQL/COBOL 프로그램을 작성할 수도 있지만, SQL 문장의 질의 결과로 반환되는 값에 대해 충분히 알고 있지 않다면 지시자 변수를 사용하여 검토하는 코드를 삽입하는 것이 좋다.

구조체 타입의 지시자

SELECT 문의 INTO 절에 구조체 변수와 지시자 변수를 함께 사용하는 경우 지시자 변수 역시 마찬가지로 별도의 구조체 변수로 구성해야 한다. 이러한 지시자 변수를 **구조체 타입의 지시자**(STRUCTURAL INDICATOR)라고 부른다.

구조체 타입의 지시자도 출력 구조체 변수를 구성하는 것과 마찬가지로 질의 결과 컬럼과 같은 순서로 지시자 변수가 와야 한다. 또한 모든 지시자 변수는 PIC S9(4) COMP-5 타입을 갖는다.

다음은 구조체 타입의 지시자 변수를 사용하는 예이다.

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 EMP.... ① ...
  03 ENAME  PIC X(24) VARYING.
  03 SALARY PIC S9(5).
  03 ADDR   PIC X(32) VARYING.
01 EMP-IND.... ② ...
  03 ENAME-IND PIC S9(4) COMP-5.
  03 SAL-IND   PIC S9(4) COMP-5.
  03 ADDR-IND  PIC S9(4) COMP-5.

...

EXEC SQL END DECLARE SECTION END-EXEC.

...

EXEC SQL SELECT ENAME, SALARY, ADDR
         INTO :EMP INDICATOR :EMP-IND
         FROM EMP
```

```
         WHERE EMPNO = :EMPNO
END-EXEC.
```

① 컬럼 ENAME, SALARY, ADDR의 내용을 저장하기 위해서 구조체 타입의 변수로 EMP를 선언한다.

② 이 구조체 변수를 SELECT 문장의 출력 변수로 사용하면서, 이에 대응되는 지시자 변수 EMP_IND 역시 구조체 변수로 선언한다. 구조체 변수 EMP_IND를 정의할 때 구조체 내의 멤버 변수를 EMP에 대응되게 정의하고, EMP_IND의 모든 멤버 변수를 PIC S9(4) COMP-5 타입으로 정의한다.

제3장 기본 프로그래밍

본 장에서는 tbESQL/COBOL 프로그램의 문법과 실행 과정, 런타임 에러(runtime error) 처리, 그리고 tbESQL/COBOL 문장의 실행, 커서를 설명한다.

3.1. 개요

본 절에서는 tbESQL/COBOL 프로그램의 문법과 런타임 에러 처리에 대해서 설명한다.

3.1.1. tbESQL/COBOL 프로그램 문법

tbESQL/COBOL 프로그램의 문법은 다음과 같다.

- SQL 문장의 시작과 끝
  - tbESQL/COBOL 프로그램에 포함되는 SQL 문장은 항상 **EXEC SQL**로 시작되며 **END-EXEC.**로 끝난다.
  - 하나의 SQL 문장은 여러 줄에 걸쳐 있을 수 있다.
- DECLARE 영역
  - DECLARE 영역은 **'BEGIN DECLARE SECTION'**으로 시작되며, **'END DECLARE SECTION'**으로 끝난다.
  - DECLARE 영역에는 변수 선언 이외에 다른 코드가 삽입되어서는 안 된다.
  - SQL 문장과 함께 사용되는 입/출력 변수는 항상 DECLARE 영역에 선언해야 한다.

    입/출력 변수가 구조체나 배열의 형태로 선언된 경우에도 DECLARE 영역에 선언해야 한다.

    단, 프리컴파일러 옵션에 따라 그렇지 않은 경우도 있다. 프리컴파일러 옵션에 대해서는 “제6장 tbESQL/COBOL 프리컴파일러 옵션”을 참고한다.
  - 입/출력 변수가 아닌 일반적인 프로그램 변수의 경우에는 DECLARE 영역 밖에 선언되어도 무방하다.
  - 다음은 DECLARE 영역의 예이다.

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 OPERATION PIC X(20) VARYING OCCURS 5 TIMES.
01 TELLER.
   03 EMPNO  PIC S9(7).
   03 ENAME  PIC X(10).
   03 SALARY PIC S9(5).
```

```
01 CNT PIC S9(9).
EXEC SQL END DECLARE SECTION END-EXEC.
```

- 문자열
  - COBOL 프로그래밍 코드에 포함된 문자열은 큰따옴표(" ")를 사용한다.
  - tbESQL/COBOL 문장에 포함되는 문자열은 작은따옴표(' ')를 사용한다.
- 주석(Comment)
  - 주석은 COBOL 프로그래밍 언어에서 사용하는 방법 이외에 두 개의 마이너스 부호(--)를 이용하는 방법이 있다.
  - 두 개의 마이너스 부호(--)를 사용하는 주석은 부호(--)가 시작되는 곳에서부터 그 라인의 끝까지 주석으로 처리한다. 또한 EXEC SQL 문장에만 사용될 수 있으며, COBOL 프로그래밍 코드 부분에는 사용되지 못한다.
  - 다음은 주석을 사용하는 예이다.

```
EXEC SQL SELECT ENAME, SALARY, ADDR
         INTO :EMP -- 구조체 변수를 사용한다.
         FROM EMP
         WHERE EMPNO = :EMPNO
END-EXEC.
```

3.1.2. 프로그램 실행 과정

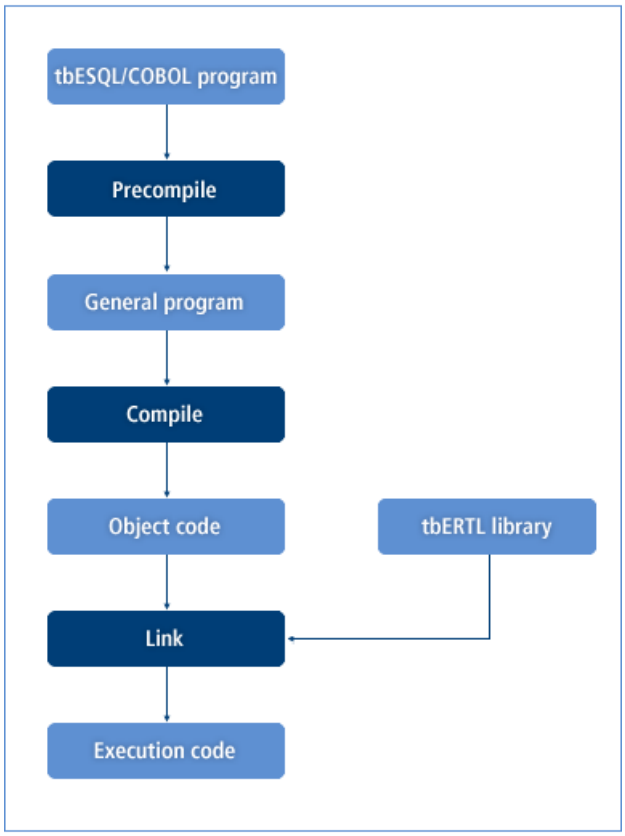
tbESQL/COBOL 프로그램을 작성할 때는 필요한 데이터 타입이나 함수 프로토타입 등을 이용하기 위해서 반드시 **SQLCA** 파일을 포함해야 한다. 즉, 아래의 내용이 항상 tbESQL/COBOL 프로그램 소스 코드의 맨 위에 명시되어 있어야 한다.

```
EXEC SQL INCLUDE SQLCA END-EXEC.
```

만약 존재하지 않으면 프리컴파일러 실행 후에 생성되는 COBOL 코드에 자동으로 추가된다.

다음 그림은 tbESQL/COBOL 프로그램 소스 코드를 실행 파일로 생성하기 위해 거치는 전 과정이다. 프리컴파일 과정을 제외하면 COBOL 프로그램의 경우와 별로 다르지 않다.

[그림 3.1] tbESQL/COBOL 프로그램 실행 과정



위의 [그림 3.1]의 과정을 순서대로 설명하면 다음과 같다.

1. tbESQL/COBOL program

   tbESQL/COBOL 프로그램을 작성한 뒤 소스 코드를 저장하면 **.tbco** 확장자를 갖는 파일이 생성된다.

2. Precompile

   작성된 프로그램을 실행하려면, 먼저 프리컴파일 과정을 거쳐야 한다. tbESQL/COBOL의 프리컴파일러를 실행하는 명령어는 **tbpcb**이다.

   다음은 emp.tbco 프로그램 파일에 대해 프리컴파일을 실행하는 예이다.

   [예 3.1] emp.tbco 프로그램의 프리컴파일

   ```
   $ tbpcb emp.tbco
   ```

   프리컴파일러를 실행하는 명령어는 옵션을 포함할 수 있다.

   다음은 프리컴파일러 옵션을 사용하여 COPY 또는 SQLCA ESQL INCLUDE 파일의 경로를 지정하는 예이다.

   [예 3.2] 프리컴파일러 옵션을 사용한 COPY 또는 SQLCA ESQL INCLUDE 파일의 경로 지정

   ```
   $ tbpcb INCLUDE=../include emp.tbco
   ```

프리컴파일러 옵션에 대한 자세한 내용은 "제6장 tbESQL/COBOL 프리컴파일러 옵션"을 참고한다.

3. General program

   프리컴파일의 결과로는 COBOL 프로그램 소스 코드가 생성되며, 이때 파일의 이름은 원본 파일의 이름과 동일하고 확장자만 **.cob**로 변경된다. 예를 들어 emp.tbco 파일을 프리컴파일하면 emp.cob라는 이름을 가진 파일이 생성된다.

4. Compile, Link

   프리컴파일이 완료된 파일은 그 다음으로 컴파일 과정과 링크 과정을 거쳐야 한다. [그림 3.1]에서는 컴파일과 링크 과정이 따로 표현되었지만, 실제로는 대개의 경우에 두 과정이 함께 수행된다.

   다음은 [예 3.1]의 실행결과로 생성된 emp.cob 파일을 컴파일하고 링크하는 예이다. 본 예제는 64-bit 머신의 MF-COBOL 컴파일러를 사용하는 경우이다.

   [예 3.3] 컴파일과 링크 과정

   ```
   $ cob64 -xo emp emp.cob -L$TB_HOME/client/lib -ltbertl -ltbcli -lpthread -lm
   ```

   tbESQL/COBOL에서는 tbCLI 함수도 함께 사용하기 때문에 tbERTL 라이브러리 이외에 tbCLI 라이브러리를 함께 링크한다.

   위의 [그림 3.1]에서 링크(Link) 과정에서 링커(Linker)의 입력으로 받아들이는 **tbERTL 라이브러리**는 tbESQL/COBOL의 함수 라이브러리이다. 이 라이브러리에는 esql_do 함수 등이 정의되어 있으며, tbESQL/COBOL 프로그램을 안전하고 효율적으로 실행하기 위한 여러 가지 작업을 수행한다.

5. Execution code

   컴파일 과정과 링크 과정을 거치고 나면 emp라는 이름의 실행 파일이 생성된다.

3.1.3. 런타임 에러 처리

tbESQL/COBOL 프로그램 내의 SQL 문장을 실행했을 때 에러 또는 경고 등의 여러 가지 예외 상황이 발생할 수 있다. 예를 들면 SELECT 문장의 실행 결과로 반환되는 로우가 존재하지 않거나 특정 컬럼의 일부 내용이 잘린 경우를 들 수 있다. tbESQL/COBOL 프로그램 내에서는 에러 또는 경고 상황이 발생한 경우 그에 대한 적절한 처리를 프로그램 내에서 수행할 수 있다.

tbESQL/COBOL에서는 이러한 런타임 에러 처리를 위해 다음의 세 가지 인터페이스를 지원한다.

인터페이스	설명
상태 변수	상태 변수(Status Variable)는 임의의 SQL 문장이 실행된 결과가 저장되는 변수이다. 프로그램 내에서는 SQL 문장을 실행한 후에 상태 변수의 값을 검토하여, 에러 또는 경고 상황의 발생을 알 수 있고, 그에 따른 처리를 수행할 수 있다.

인터페이스	설명
SQLCA	SQLCA(SQL 통신 영역: SQL Communication Area)는 임의의 SQL 문장이 실행된 결과가 저장되는 구조체 변수이다. 이 구조체는 SQLCA라는 이름으로 SQLCA ESQL INCLUDE 파일에 정의되어 있으며, 상태 변수를 포함하고 있다. 상태 변수와 마찬가지로, SQL 문장을 실행한 후에 SQLCA 내의 적절한 멤버 변수의 값을 검토하여 에러 또는 경고 상황의 발생을 알 수 있고, 그에 따른 처리를 수행할 수 있다.
WHENEVER	WHENEVER 문장은 에러 또는 경고 상황이 발생하면 미리 정해진 특정 동작을 수행한다. 상태 변수나 SQLCA 구조체를 이용하면 SQL 문장을 실행할 때마다 에러 또는 경고 상황이 발생하였는지 검토해야 한다. 하지만 WHENEVER 문장을 사용하면 tbESQL/COBOL이 자동으로 예외 상황을 검토하고 그에 따른 처리를 수행한다.

3.2. 프로그램 구조

tbESQL/COBOL 프로그램의 구조는 다음과 같다.

- 변수 선언
- 초기화
- 데이터베이스 작업
- 종료화
- 에러 처리

3.2.1. 변수 선언

변수 선언 부분에는 “1.2. 구성요소”에서 설명한 DECLARE 영역이 포함된다. tbESQL/COBOL 문장에서 데이터베이스 작업에 사용될 모든 변수를 DECLARE 영역에 선언해야 한다. 데이터베이스 작업과 관련이 없는 변수는 DECLARE 영역에 포함하지 않아도 된다.

다음은 변수 선언의 예이다.

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 BRANCH pic x(9) varying.
01 POSTAL pic x(10) varying.
01 REGION pic x(10) varying.
01 OPERATION pic x(20) varying occurs 5 times.
01 TELLER.
    03 IDE pic x(5).
```

```
    03 BR pic x(5).
    03 NAMEE pic x(12).
01 TELLER2 occurs 10 times.
    03 IDE pic x(5).
    03 BR pic x(5).
    03 NAMEE pic x(12).
01 IDK pic x(5) occurs 10 times.
01 NAMEK pic x(12) occurs 10 times.
01 CDK pic x(5).
EXEC SQL END DECLARE SECTION END-EXEC.
```

3.2.2. 초기화

초기화 부분에서는 다음의 두 가지를 수행한다.

- 런타임 에러가 발생했을 때 어떤 작업을 수행할 것인지 선언한다.

  런타임 에러가 발생했을 때 수행하는 작업에는 에러 처리 함수를 호출하는 경우, 에러를 무시하고 프로그램을 계속 진행하는 경우, 프로그램을 종료하는 경우, 특정 위치로 이동한 후 실행을 계속하는 경우 등이 있다. 대부분의 경우에 에러 처리를 위한 함수를 미리 정의하고 그 정의된 함수를 호출한다.

  다음은 런타임 에러가 발생했을 때 `tbesql_error` 함수를 호출하는 예이다.

  [예 3.4] `tbesql_error`함수 호출

  ```
  EXEC SQL WHENEVER SQLERROR
  DO CALL "tbesql_error" USING
  BY REFERENCE SQLCODE IN SQLCA
  END-EXEC.
  ```

- Tibero의 데이터베이스에 접속한다.

  데이터베이스에 접속할 때는 반드시 사용자 이름과 패스워드를 함께 명시해야 한다.

  다음의 소스 코드는 USERNAME과 PASSWORD, 이렇게 두 개의 입력 변수를 이용해 데이터베이스에 접속하는 예이다.

  [예 3.5] 입력 변수를 이용해 데이터베이스에 접속

  ```
  EXEC SQL CONNECT :USERNAME IDENTIFIED BY :PASSWORD END-EXEC.
  ```

3.2.3. 데이터베이스 작업

데이터베이스 작업 부분에서는 tbESQL/COBOL 문장을 사용해 데이터베이스 질의 및 갱신을 수행한다. 이 부분은 tbESQL/COBOL 프로그램에서 가장 중요한 부분 중 하나이다.

데이터베이스와 관련된 작업에는 입력 변수와 출력 변수를 많이 사용하게 된다. 데이터베이스 질의와 관련된 소스 코드에는 커서를 선언하고, 이 선언된 커서를 이용해 로우를 액세스하는 코드가 포함된다.

다음은 데이터베이스 작업 부분의 예이다.

```
EXEC SQL
     DECLARE C1 CURSOR FOR SELECT * FROM BRANCH
     ORDER BY BRANCH_CD
END-EXEC.

EXEC SQL
     OPEN C1
END-EXEC.

EXEC SQL
     FETCH C1 INTO :BRANCH, :POSTAL, :REGION
END-EXEC.
```

3.2.4. 종료화

종료화 부분에서는 모든 데이터베이스 작업을 마치고 커밋을 수행하거나 롤백을 수행한다.

주의

종료화 부분이 tbESQL/COBOL 프로그램에 포함되지 않으면, 자동으로 커밋되지 않으므로 주의한다.

다음은 데이터베이스에 부분 롤백을 수행한 뒤 커밋을 하는 예이다.

```
EXEC SQL ROLLBACK WORK TO SAVEPOINT SP1 END-EXEC.
EXEC SQL COMMIT WORK END-EXEC.
```

3.2.5. 에러 처리

에러 처리 부분에서는 에러를 처리하기 위한 코드가 포함된다. 에러 처리와 관련된 코드는 다른 코드와 섞여 동일한 하나의 함수 안에 포함될 수도 있으며, 별도의 함수로 정의할 수도 있다.

다음은 에러 처리의 예이다.

```
IF SQLCODE IN SQLCA IS LESS THAN 0
THEN DISPLAY "connection failed"
END-IF
```

3.3. tbESQL/COBOL 문장 실행

본 절에서는 tbESQL/COBOL 프로그램에서 SELECT, INSERT, UPDATE, DELETE 문장을 실행하는 방법에 대해 설명한다. 각 문장에는 입/출력 변수가 사용되는데, 입/출력 변수는 각 문장 내에 포함된 스키마 객체와 구별하기 위하여 반드시 앞에 **콜론(:)**이 와야 한다.

3.3.1. SELECT

SELECT 문장은 데이터베이스에 질의를 수행하고 결과 로우를 반환하는 문장이다. 결과 로우의 개수는 보통 하나 이상이지만 하나도 없을 수도 있다.

INTO 절과 출력 변수

tbESQL/COBOL 프로그램 내에서 사용되는 SELECT 문장은 일반 SELECT 문장과 같은 문법을 가진다. 다만 SELECT 리스트 다음에 결과 로우의 각 컬럼 값을 **출력 변수**에 저장하기 위해 **INTO 절**이 삽입된다.

다음은 SELECT 문장의 INTO 절에 출력 변수가 사용된 예이다.

```
EXEC SQL
     SELECT A INT :COL-A FROM TAB1
END-EXEC.
```

다음은 SELECT문장의 INTO 절에 포함되는 출력 변수에 대한 설명이다.

- 컬럼 값과 출력 변수의 대응

  INTO 절에 포함되는 출력 변수는 SELECT 리스트 내의 컬럼과 같은 개수이어야 하며, 지시자 변수와 함께 사용될 수 있다. 질의 결과로 반환된 로우의 각 컬럼 값은 컬럼 값과 동일한 순서로 대응되는 각각의 출력 변수에 저장된다. 출력 변수에 저장될 때 tbESQL/COBOL 프로그램에서는 필요한 경우 데이터 타입의 변환을 수행한다.

- 구조체 변수

  INTO 절에 포함되는 출력 변수에는 구조체 변수를 사용할 수도 있다. 이때 구조체 변수에 포함된 멤버 변수의 개수는 SELECT 리스트 내의 컬럼과 개수가 같아야 한다. tbESQL/COBOL 프로그램에서는 결과 로우의 각 컬럼 값을 구조체 변수 내의 각 멤버 변수에 할당한다.

- 로우의 개수에 따른 출력 변수

  SELECT 문장의 결과 로우의 개수가 반드시 하나라는 보장이 있다면 INTO 절에 단순 출력 변수를 이용하여 처리가 가능하다. 하지만 하나 이상인 경우에는 커서를 사용하거나 INTO 절에 출력 배열 변수를 사용해야 한다.

참고

커서에 대한 자세한 내용은 "3.4. 커서"와 "3.5. 스크롤 가능 커서"를, 배열 변수에 대한 자세한 내용은 "제4장 배열 변수"를 참고한다.

다음은 결과 로우가 하나인 경우 이를 처리하는 예이다.

```
01 ENAME  PIC X(24) VARYING.
01 SALARY PIC S9(5).
01 ADDR   PIC X(32) VARYING.
...
MOVE 20 TO EMPNO.
EXEC SQL SELECT ENAME, SALARY * 1.05
         INTO :ENAME, :SALARY
         FROM EMP
         WHERE EMPNO = :EMPNO
END-EXEC.

DISPLAY 'ENAME = ' ENAME-ARR.
```

위의 예에서 컬럼 EMPNO가 테이블 EMP의 기본 키 컬럼이므로 질의 결과 로우의 개수가 하나라는 것을 알 수 있다.

WHERE 절과 입력 변수

SELECT 문장 내에서 변수의 위치는 INTO 절에 출력 변수가 포함되는 것 외에 **WHERE 절**에 **입력 변수**가 포함된다.

다음은 SELECT 문장의 WHERE 절에 입력 변수가 사용된 예이다.

```
EXEC SQL
  SELECT EMP_NO, EMP_NAME
   INTO :EMP-A, :NAME
  FROM EMP
  WHERE DEPT_NO = :DEPT-NO
END-EXEC.
```

다음은 SELECT 문장의 WHERE 절에 포함되는 입력 변수에 대한 설명이다.

- 입력 변수의 값 설정

  WHERE 절에 포함되는 입력 변수의 값은 SELECT 문장이 실행되기 전에 설정되어 있어야 한다. 그 이유는 SELECT 문장은 실행 직전에 입력 변수의 값을 가져와서 SELECT 문장을 완성한 뒤에 실행되기 때문이다. SELECT 문장의 실행에 필요한 입력 변수의 값은 그 문장이 실행되기 직전에 읽혀지므로, 입력 변수의 값은 프로그램 실행 중에 동적으로 설정할 수 있다.

- 입력 변수 사용의 제약

  SELECT 문장의 입력 변수는 상수를 대신하여 사용할 수 있지만, 스키마 객체 또는 컬럼 등의 이름을 대신하여 사용될 수는 없다. SELECT 문장에 부질의(Subquery)가 사용되는 경우에는 부질의 내에 출력 변수를 포함시킬 수는 없지만 입력 변수를 포함시킬 수는 있다.

다음은 컬럼의 이름을 대신하여 입력 변수가 사용된 예이다.

```
01 ENAME    PIC X(24) VARYING.
01 SALARY   PIC S9(5).
01 COL-NAME PIC X(32) VARYING.
...
MOVE Z"EMPNO" TO COL-NAME-ARR.
EXEC SQL SELECT ENAME, SALARY
         INTO :ENAME, :SALARY
         FROM EMP
         WHERE :COL-NAME = 20
END-EXEC.
```

위의 예에서 WHERE 절 다음에 컬럼의 이름이 나와야 하는데, 대신 변수가 사용되었기 때문에 잘못되었다.

3.3.2. INSERT

INSERT, DELETE, UPDATE 문장은 공통적으로 질의의 결과 로우가 존재하지 않으므로 출력 변수 없이 입력 변수만을 사용한다. **INSERT** 문장에서 입력 변수는 **컬럼**에 삽입할 데이터 값의 위치나 **부질의** 내부에 사용될 수 있다.

INSERT 문장에서 삽입하고자 하는 컬럼 값의 일부에 대해서만 입력 변수를 사용할 수도 있다.

다음은 일부 컬럼에 대해서만 입력 변수를 사용하는 예이다.

```
01 ENAME PIC X(24) VARYING.
01 EMPNO PIC S9(9).
...
MOVE 20 TO EMPNO.
MOVE Z"Peter" TO ENAME-ARR.
EXEC SQL INSERT INTO EMP (ENAME, SALARY, EMPNO)
                VALUES (:ENAME, 35000, :EMPNO)
END-EXEC.
```

삽입하고자 하는 모든 컬럼 값에 대하여 입력 변수를 사용하는 경우 구조체 변수를 사용할 수 있다. 이때 구조체 변수에 포함된 각 변수 값은 삽입하고자 하는 각 컬럼 값에 대응된다.

다음은 구조체 변수를 사용해 데이터를 삽입하는 예이다.

```
01 EMP.
   03 ENAME  PIC X(24) VARYING.
   03 SALARY PIC S9(5).
   03 EMPNO  PIC S9(9).
...
MOVE Z"Peter" TO ENAME-ARR IN ENAME IN EMP.
MOVE 35000 TO SALARY IN EMP.
MOVE 25 TO EMPNO IN EMP.
EXEC SQL INSERT INTO EMP (ENAME, SALARY, EMPNO)
                VALUES (:EMP)
END-EXEC.
```

또한 부질의에 입력 변수를 사용할 수도 있다. 다음은 부질의를 포함하는 INSERT 문장의 예이다.

```
01 SAL-BOUND PIC S9(5).
...
MOVE 30000 TO SAL-BOUND.
EXEC SQL INSERT INTO EMP_SUB (ENAME, EMPNO)
         SELECT ENAME, EMPNO FROM EMP
         WHERE SALARY >= :SAL-BOUND
END-EXEC.
```

3.3.3. UPDATE

UPDATE 문장도 INSERT 문장과 마찬가지로 입력 변수만을 사용한다.

UPDATE 문장에서는 **SET 절**의 컬럼 값의 위치나 **WHERE 절**에서 입력 변수가 사용될 수 있다. UPDATE 문장에 포함된 **부질의**에서도 입력 변수를 사용할 수 있다. UPDATE 문장에서는 INSERT 문장에서와 달리 구조체 입력 변수를 사용할 수 없다.

다음은 일부 컬럼 값과 WHERE 절 내에 입력 변수를 사용하는 예이다.

```
01 ENAME PIC X(24) VARYING.
01 EMPNO PIC S9(9).
...
EXEC SQL UPDATE EMP
         SET ENAME = :ENAME, SALARY = SALARY * 1.05
         WHERE EMPNO = :EMPNO
END-EXEC.
```

다음은 부질의에 입력 변수를 사용하는 예이다.

```
01 EMPNO     PIC S9(9).
01 SAL-BOUND PIC S9(5).
```

```
...
MOVE 20 TO EMPNO.
MOVE 30000 TO SAL-BOUND.
EXEC SQL UPDATE EMP
         SET SALARY = (SELECT SALARY FROM EMP WHERE EMPNO = :EMPNO)
         WHERE SALARY <= :SAL-BOUND
END-EXEC.
```

3.3.4. DELETE

DELETE 문장도 INSERT, UPDATE 문장과 마찬가지로 입력 변수만을 사용한다. DELETE 문장에서는 **WHERE 절**에서 입력 변수가 사용된다. DELETE 문장에서도 UPDATE 문장과 마찬가지로 구조체 입력 변수를 사용할 수 없다.

다음은 입력 변수를 사용하는 DELETE 문장의 예이다.

```
01 SAL-BOUND PIC S9(5).
...
MOVE 25000 TO SAL-BOUND.
EXEC SQL DELETE FROM EMP
         WHERE SALARY <= :SAL-BOUND
END-EXEC.
```

3.4. 커서

본 절에서는 커서의 기본적인 사용 방법에 대하여 설명하고, 갱신 및 삭제를 위한 CURRENT OF 절을 설명한다. 그리고 마지막으로 사용 예제를 제시한다.

3.4.1. 사용 방법

SELECT 문을 통한 질의를 수행할 때 WHERE 절에 기본 키 제약조건을 부여하지 않으면, 대개의 경우 결과 로우의 개수는 하나 이상이다. **커서**는 이렇게 반환된 다수의 결과 로우에 각각 차례로 액세스하기 위한 데이터 구조이다.

다음은 커서를 사용하는 순서이다.

1. 커서를 사용하기 위해서는 DECLARE CURSOR를 사용해 맨 먼저 SQL 문장과 연관하여 커서를 선언해야 한다. 커서를 선언할 때에는 항상 커서의 이름을 주어야 하며, 커서의 선언부는 그 커서를 사용하는 다른 모든 문장의 앞에 와야 한다.

   다음은 EMP-CURSOR라는 이름으로 커서를 선언하는 예이다.

[예 3.6] 커서의 선언

```
EXEC SQL DECLARE EMP-CURSOR CURSOR FOR
         SELECT ENAME, SALARY, ADDR
         FROM EMP
         WHERE DEPTNO = :DEPTNO
END-EXEC.
```

2. 커서를 사용하기 위해서는 OPEN을 사용해 해당 커서를 열어야 한다.

   OPEN을 실행하면 연관된 SELECT 문장이 실행되어 질의의 결과 로우가 반환된다. 또한 커서는 결과 로우 중에서 맨 처음에 위치한 로우의 직전을 가리킨다. 첫 FETCH가 실행되면 첫 번째 로우를 가리키게 된다. OPEN을 실행할 때 유의할 점은 커서를 선언할 때 SELECT 문장에 포함된 입력 변수의 값은 OPEN이 실행될 때 할당된다는 것이다.

   다음은 [예 3.6]에서 선언한 EMP-CUROSR라는 이름의 커서에 OPEN을 실행하는 예이다.

   [예 3.7] OPEN의 실행

   ```
   EXEC SQL OPEN EMP-CURSOR
   END-EXEC.
   ```

3. FETCH를 실행해 로우에 액세스를 한다.

   OPEN의 실행으로는 아직 로우에 액세스를 할 수 있는 것은 아니다. 로우에 액세스를 하기 위해서는 FETCH를 실행해야 한다. FETCH의 INTO 절에는 구조체 변수나 지시자 변수를 함께 사용할 수 있다.

   다음은 FETCH를 실행하는 예이다.

   [예 3.8] FETCH의 실행

   ```
   EXEC SQL FETCH EMP-CURSOR
            INTO :ENAME, :SALARY, :ADDR
   END-EXEC.
   ```

   FETCH를 실행하면 먼저 커서가 다음 결과 로우를 가리키게 되고, 커서가 가리키는 결과 로우를 출력 변수에 저장한다. FETCH를 실행할 때마다 커서는 다음 결과 로우를 가리키고, 결국 맨 마지막 결과 로우의 범위를 넘어 FETCH를 실행하면, NOT FOUND 에러가 발생한다.

   대개의 경우 FETCH를 **무한 루프** 안에 포함시키며, **NOT FOUND 에러**가 발행하면 루프를 빠져 나오도록 코드를 작성한다. 이때 NOT FOUND 에러가 발생했을 때 루프를 빠져 나오도록 하기 위해서는 **WHENEVER** 문장을 사용한다.

   다음은 WHENEVER 문장을 사용하는 예이다.

   [예 3.9] WHENEVER 문장의 사용

   ```
   EXEC SQL WHENEVER NOT FOUND GOTO FETCH-END END-EXEC.
   FETCH-LOOP.
   ```

```
   EXEC SQL FETCH EMP-CURSOR
            INTO :ENAME, :SALARY, :ADDR
   END-EXEC.
...
   GO TO FETCH-LOOP.
FETCH-END.
```

WHENEVER 문장에 GOTO 명령 이외에 DO CALL, DO PERFORM 등의 명령을 사용할 수도 있다.

OPEN을 사용해 커서를 열기 전이나 CLOSE를 사용해 커서를 닫은 후 또는 NOT FOUND 에러가 발생한 이후에 FETCH를 실행하면 에러가 발생한다.

FETCH를 이용해 다음 로우 뿐만 아니라 이전 로우를 액세스할 수도 있다. 이러한 작업을 위해서는 스크롤 가능 커서를 선언해야 한다. 스크롤 가능 커서에 대해서는 "3.5. 스크롤 가능 커서"를 참고한다.

4. 커서 사용의 마지막 단계는 CLOSE를 사용해 커서를 닫는 것이다. 커서를 닫은 이후에는 그 커서에 대해 어떠한 작업도 실행할 수 없다.

   다음은 EMP-CURSOR라는 이름의 커서에 CLOSE를 실행하는 예이다.

   [예 3.10] CLOSE의 사용

```
EXEC SQL CLOSE EMP-CURSOR
END-EXEC.
```

3.4.2. CURRENT OF 절

커서를 이용해 SELECT 문장의 실행 결과 로우를 차례로 액세스하면서 커서가 현재 가리키고 있는 결과 로우를 삭제하거나 갱신하려고 할 때 DELETE 문장과 UPDATE 문장에 **CURRENT OF** 절을 사용한다.

SELECT 문장에서 CURRENT OF 절을 사용하기 위해서는 FOR UPDATE 절을 포함해야 한다. **FOR UPDATE** 절은 질의 결과로 반환된 로우에 잠금(LOCK)을 설정한다. 잠금이 설정된 로우는 현재 트랜잭션이 커밋 또는 롤백되기 전까지는 다른 트랜잭션이 로우를 갱신하거나 삭제할 수 없다.

다음은 UPDATE 문장에서 CURRENT OF 절을 이용하여 컬럼 SALARY만을 갱신하는 예이다.

```
EXEC SQL DECLARE EMP-CURSOR CURSOR FOR
         SELECT ENAME, SALARY, ADDR
         FROM EMP
         WHERE DEPTNO = :DEPTNO
         FOR UPDATE OF SALARY
END-EXEC.
...
EXEC SQL OPEN CURSOR EMP-CURSOR END-EXEC.
EXEC SQL WHENEVER NOT FOUND GOTO FETCH-END END-EXEC.
```

```
FETCH-LOOP.
   EXEC SQL FETCH EMP-CURSOR
            INTO :ENAME, :SALARY, :ADDR
   END-EXEC.
   ...
   EXEC SQL UPDATE EMP
            SET SALARY = SALARY * 1.05
            WHERE CURRENT OF EMP-CURSOR
   END-EXEC.
   ...
   GO TO FETCH-LOOP.
FETCH-END.
```

커서가 현재 가리키고 있는 로우는 FETCH 문장을 실행하여 방금 전에 컬럼 값을 읽은 로우이다. 커서에 대해 OPEN 문장을 실행한 후에 한번도 FETCH 문장을 실행하지 않았거나 모든 결과 로우를 읽고 나서 NOT FOUND 에러가 반환되었다면 커서가 현재 가리키고 있는 로우는 없다.

현재 가리키고 있는 로우가 없는 커서를 이용하여 DELETE 또는 UPDATE 문장을 실행하였다면 에러를 반환한다. 또한 OPEN 문장을 수행하지 않았거나 CLOSE 문장을 이미 수행한 커서를 이용하여 삭제 또는 갱신을 시도할 때에도 에러를 반환한다.

CLOSE_ON_COMMIT 옵션이 'YES'로 지정된 경우를 제외하고는 일반적으로 커서는 현재 트랜잭션이 커밋 또는 롤백한 후에도 사용할 수 있다. 즉, 커서를 이용하여 질의 결과 로우를 액세스할 수 있다. 하지만 FOR UPDATE 절을 포함한 SELECT 문장에 대한 커서는 사용할 수 없다. 왜냐하면 트랜잭션이 커밋되거나 롤백되는 동시에 결과 로우에 설정되었던 잠금을 해제해 버리기 때문이다.

3.4.3. 사용 예제

다음은 커서를 사용하는 예제 프로그램이다.

[예 3.11] 커서의 사용

```
IDENTIFICATION                    DIVISION.
************************************************************
PROGRAM-ID.                       TEST.
************************************************************
ENVIRONMENT                       DIVISION.
************************************************************
CONFIGURATION                     SECTION.
*
SOURCE-COMPUTER.                  TEST-MACHINE.
OBJECT-COMPUTER.                  TEST-MACHINE.
************************************************************
DATA                              DIVISION.
************************************************************
WORKING-STORAGE                   SECTION.
```

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.... ① ...
01 USERPASS PIC X(20) VALUE Z"tibero/tmax".
01 ENAME    PIC X(24) VARYING.
01 SALARY   PIC S9(5).
01 ADDR     PIC X(32) VARYING.
01 DEPTNO   PIC S9(9).
EXEC SQL END DECLARE SECTION END-EXEC.

EXEC SQL INCLUDE SQLCA END-EXEC.

PROCEDURE             DIVISION.

EXEC SQL DECLARE EMP-CURSOR CURSOR FOR... ② ...
         SELECT ENAME, SALARY, ADDR
         FROM EMP
         WHERE DEPTNO = :DEPTNO... ③ ...
END-EXEC.

EXEC SQL DECLARE EMP-UPDATE-CURSOR CURSOR FOR... ④ ...
         SELECT SALARY
         FROM EMP
         WHERE DEPTNO = :DEPTNO... ⑤ ...
         FOR UPDATE OF SALARY END-EXEC.

EXEC SQL CONNECT :USERPASS END-EXEC.
DISPLAY "Connected.".

DISPLAY "Enter dept number to show: "
ACCEPT DEPTNO.

EXEC SQL OPEN EMP-CURSOR END-EXEC.
EXEC SQL WHENEVER NOT FOUND GOTO FETCH-END END-EXEC.
FETCH-LOOP.
   EXEC SQL FETCH EMP-CURSOR
            INTO :ENAME, :SALARY, :ADDR
   END-EXEC.
   DISPLAY 'ENAME = 'ENAME-ARR', SALARY = 'SALARY
           ',ADDR = 'ADDR-ARR
   GO TO FETCH-LOOP.
FETCH-END.

EXEC SQL CLOSE EMP-CURSOR END-EXEC.

DISPLAY "Enter dept number to raise salary: "
ACCEPT DEPTNO.
```

```
EXEC SQL OPEN EMP-UPDATE-CURSOR END-EXEC.
EXEC SQL WHENEVER NOT FOUND GOTO FETCH-UPDATE-END END-EXEC.
FETCH-UPDATE-LOOP.
   EXEC SQL FETCH EMP-UPDATE-CURSOR
            INTO :SALARY
   END-EXEC.
   EXEC SQL UPDATE EMP
            SET SALARY = :SALARY * 1.05
            WHERE CURRENT OF EMP-UPDATE-CURSOR
   END-EXEC.
   GO TO FETCH-UPDATE-LOOP.
FETCH-UPDATE-END.

EXEC SQL CLOSE EMP-UPDATE-CURSOR END-EXEC.
EXEC SQLCOMMIT WORK RELEASE END-EXEC.... ⑥ ...
```

① tbESQL/COBOL 문장 내에 포함되는 모든 입/출력 변수는 DECLARE 영역 안에서 선언한다. 프로그램의 맨 앞쪽에서 커서를 선언하고 있지만, 변수의 선언과는 달리 커서의 선언은 어떠한 위치에 오더라도 상관없으며, 그 커서가 사용되기 전에만 선언되면 된다.

②, ④ 두 개의 커서를 선언한다. 각각 단순 질의와 갱신을 위한 커서인데, 단순 질의를 위한 커서는 EMP-CURSOR이고, 갱신을 위한 커서는 EMP-UPDATE-CURSOR이다.

③, ⑤ 변수 DEPTNO가 두 개의 커서에 공통적으로 사용된다. 커서와 연관된 SELECT 문장은 OPEN 문장으로 커서를 열 때 실행되며, 그 직전에 입력 변수의 값을 읽어 들인다. 따라서 같은 변수를 사용하더라도 각각의 SELECT 문장이 실행될 때 서로 다른 DEPTNO 값이 적용될 수도 있다.

⑥ 프로그램의 맨 마지막에서는 현재 트랜잭션을 커밋한다. 단순 질의를 위한 커서 EMP-CURSOR는 트랜잭션 커밋 후에도 계속 사용할 수 있으나, 갱신을 위한 커서 EMP-UPDATE-CURSOR는 사용할 수 없다.

3.5. 스크롤 가능 커서

본 절에서는 스크롤 가능 커서(Scrollable Cursors)의 사용 방법을 설명하고, 사용 예제를 제시한다.

3.5.1. 사용 방법

커서는 질의의 결과 로우를 액세스할 때 항상 다음에 위치한 로우만 액세스할 수 있는 것에 비해, **스크롤 가능 커서**는 임의의 로우에 액세스를 할 수 있다. 예를 들어 스크롤 가능 커서는 현재 커서가 가리키고 있는 로우의 바로 이전 로우를 액세스하거나 전체 결과 로우 중에서 n번째 로우를 액세스할 수 있다.

스크롤 가능 커서는 사용 방법의 편리성과 유연성을 제공하지만 커서와 비교했을 때 메모리 등의 리소스를 많이 사용할 수 있으므로 프로그램의 실행 성능을 떨어뜨릴 수 있다. 따라서 꼭 필요한 경우가 아니라면 커서를 사용하는 것이 효율적이다.

다음은 커서와 스크롤 가능 커서의 차이점이다.

커서	스크롤 가능 커서
다음 위치에 있는 로우만 차례대로 액세스를 한다.	임의의 위치에 있는 로우를 액세스할 수 있다.
'DECLARE {*커서 이름*} CURSOR'의 형태로 선언한다.	'DECLARE {*커서 이름*} SCROLL CURSOR'의 형태로 선언한다.
FETCH를 실행할 때 옵션을 지정할 수 없다.	FETCH를 실행할 때 반드시 옵션을 지정해야 한다.

스크롤 가능 커서에서도 커서와 동일하게 OPEN과 CLOSE를 사용한다. 스크롤 가능 커서가 현재 가리키고 있는 로우에 대하여 삭제 및 갱신을 수행하고자 할 때에도 커서와 마찬가지로 DELETE 문장과 UPDATE 문장 내에서 CURRENT OF 절을 이용한다. 문장의 작성 및 사용 방법은 커서와 동일하다.

커서와 스크롤 가능 커서의 차이점을 좀더 상세하게 설명하면 다음과 같다.

- 스크롤 가능 커서의 선언

  스크롤 가능 커서의 선언은 **SCROLL** 키워드가 포함된다는 것을 제외하면 커서의 선언과 동일하며 다음과 같은 형태로 선언한다.

  ```
  DECLARE {커서 이름} SCROLL CURSOR
  ```

  다음의 소스 코드는 스크롤 가능 커서를 선언하는 예이다.

  ```
  EXEC SQL DECLARE EMP-SCROLL-CURSOR SCROLL CURSOR FOR
           SELECT ENAME, SALARY, ADDR
           FROM EMP
           WHERE DEPTNO = :DEPTNO
  END-EXEC.
  ```

- 스크롤 가능 커서에서의 FETCH의 사용

  스크롤 가능 커서에서 FETCH를 사용할 때는 항상 액세스할 대상 로우를 지정해야 한다.

  다음은 FETCH에서 액세스를 할 대상 로우를 지정할 때 사용되는 옵션이다.

옵션	설명
NEXT	현재 커서가 가리키고 있는 로우의 다음 로우에 액세스를 한다. PRIOR 옵션과 반대이다. (생략 가능)
PRIOR	현재 커서가 가리키고 있는 로우의 이전 로우에 액세스를 한다. NEXT 옵션과 반대이다.
FIRST	맨 처음에 위치한 로우에 액세스를 한다. LAST 옵션과 반대이다.
LAST	맨 마지막에 위치한 로우에 액세스를 한다. FIRST 옵션과 반대이다.
CURRENT	현재 로우에 액세스를 한다.
RELATIVE offset	현재 커서가 가리키고 있는 로우의 다음 offset 번째에 위치한 로우에 액세스를 한다. offset 값이 음수라면 커서가 현재 위치에서 앞으로 이동한다. 예를 들어 현재

옵션	설명
	커서가 8번째 로우를 가리키고 있는데, 'FETCH RELATIVE -3'을 실행한다면 커서는 5번째 로우를 가리키게 된다.
ABSOLUTE offset	전체 로우 중에서 offset 번째 로우에 액세스를 한다.

다음은 FETCH에 옵션을 사용하는 예이다.

```
EXEC SQL FETCH PRIOR EMP-SCROLL-CURSOR
         INTO :ENAME, :SALARY, :ADDR
END-EXEC.

EXEC SQL FETCH LAST EMP-SCROLL-CURSOR
         INTO :ENAME, :SALARY, :ADDR
END-EXEC.

EXEC SQL FETCH ABSOLUTE 3 EMP-SCROLL-CURSOR
         INTO :ENAME, :SALARY, :ADDR
END-EXEC.
```

위의 예에서는 각각 순서대로 첫 번째 문장은 이전 로우를 액세스하고, 두 번째 문장은 마지막 로우를 액세스한다. 세 번째 문장은 전체 로우 중에서 세 번째 로우를 액세스하고 있다. 만약 액세스하려는 로우가 존재하지 않으면 NOT FOUND 에러가 반환된다.

3.5.2. 사용 예제

다음은 스크롤 가능 커서를 사용하는 예제 프로그램이다.

```
IDENTIFICATION                      DIVISION.
*************************************************************
PROGRAM-ID.                         TEST.
*************************************************************
ENVIRONMENT                         DIVISION.
*************************************************************
CONFIGURATION                       SECTION.
*
SOURCE-COMPUTER.                    TEST-MACHINE.
OBJECT-COMPUTER.                    TEST-MACHINE.
*************************************************************
DATA                                DIVISION.
*************************************************************
WORKING-STORAGE                     SECTION.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 USERPASS PIC X(20) VALUE Z"tibero/tmax".
01 ENAME    PIC X(24) VARYING.
```

```
01 SALARY   PIC S9(5).
01 ADDR     PIC X(32) VARYING.
01 DEPTNO   PIC S9(9).
EXEC SQL END DECLARE SECTION END-EXEC.

EXEC SQL INCLUDE SQLCA END-EXEC.

PROCEDURE             DIVISION.

EXEC SQL DECLARE EMP-SCROLL-CURSOR SCROLL CURSOR FOR
         SELECT ENAME, SALARY, ADDR
         FROM EMP
         WHERE DEPTNO = :DEPTNO
END-EXEC.

EXEC SQL CONNECT :USERPASS END-EXEC.
DISPLAY "Connected.".
MOVE 10 TO DEPTNO.

EXEC SQL OPEN EMP-SCROLL-CURSOR END-EXEC.

* 1st row
 EXEC SQL FETCH FIRST EMP-SCROLL-CURSOR
          INTO :ENAME, :SALARY, :ADDR
 END-EXEC.
* last row
 EXEC SQL FETCH LAST EMP-SCROLL-CURSOR
          INTO :ENAME, :SALARY, :ADDR
 END-EXEC.
* 5th row
 EXEC SQL FETCH ABSOLUTE 5 EMP-SCROLL-CURSOR
          INTO :ENAME, :SALARY, :ADDR
 END-EXEC.
* 8th row
 EXEC SQL FETCH RELATIVE 3 EMP-SCROLL-CURSOR
          INTO :ENAME, :SALARY, :ADDR
 END-EXEC.
* 7th row
 EXEC SQL FETCH PRIOR EMP-SCROLL-CURSOR
          INTO :ENAME, :SALARY, :ADDR
 END-EXEC.
* 7th row
 EXEC SQL FETCH CURRENT EMP-SCROLL-CURSOR
          INTO :ENAME, :SALARY, :ADDR
 END-EXEC.
* 4th row
 EXEC SQL FETCH RELATIVE -3 EMP-SCROLL-CURSOR
```

```
          INTO :ENAME, :SALARY, :ADDR
       END-EXEC.
* 5th row
 EXEC SQL FETCH EMP-SCROLL-CURSOR
          INTO :ENAME, :SALARY, :ADDR
 END-EXEC.

EXEC SQL CLOSE EMP-SCROLL-CURSOR END-EXEC.
EXEC SQL COMMIT WORK RELEASE END-EXEC.
```

위의 예에서는 주석을 삽입하여, FETCH를 실행할 때마다 스크롤 가능 커서의 현재 위치를 설명하였다. 또한 질의 결과 로우가 8개 이상임을 가정한다.

SQL 문장 내에 포함되는 모든 입/출력 변수는 DECLARE 영역 안에서 선언하였다. 커서를 선언할 때와 마찬가지로 스크롤 가능 커서를 선언하는 문장도, 스크롤 가능 커서가 사용되기 전이라면 어떤 위치에 있어도 상관없다.

제4장 배열 변수

본 장에서는 배열 변수의 기본 개념과 선언 방법, 그리고 배열 변수를 tbESQL/COBOL 문장에서 입/출력 변수로 사용하는 방법을 설명한다.

4.1. 개요

tbESQL/COBOL 프로그램의 배열 변수는 COBOL 프로그래밍 언어의 배열 변수의 개념과 동일하다. 배열은 동일한 타입의 값을 여러 개 저장할 수 있는 데이터 구조이다.

tbESQL/COBOL 프로그램에서는 SELECT 문장을 실행한 뒤 다수의 결과 로우를 저장하기 위해서 출력 변수로 배열 변수를 사용하거나, INSERT 문장을 사용할 때도 보통 여러 개의 로우를 삽입하게 되므로 입력 변수로 배열 변수를 사용한다.

배열 변수가 출력 변수로 사용될 때는 **출력 배열 변수**라고 부르며, 입력 변수로 사용될 때는 **입력 배열 변수**라고 한다. 출력 배열 변수와 입력 배열 변수가 사용되는 문장은 다음과 같다.

- 출력 배열 변수
  - SELECT
- 입력 배열 변수
  - INSERT
  - UPDATE
  - DELETE

tbESQL/COBOL 프로그램에서 tbESQL/COBOL 문장과 COBOL 프로그램의 변수 사이에 데이터를 주고받을 때 배열 변수를 사용하면 다음과 같은 장점이 있다.

- 간결하고 구조적인 프로그래밍

  배열 변수를 사용하면 각각의 로우를 처리할 때마다 별도의 SQL 문장을 사용하지 않고, 모든 로우를 하나의 SQL 문장으로 처리할 수 있다. 따라서 프로그램의 소스 코드를 보다 단순화할 수 있으며, 구조적인 프로그래밍이 가능하다.

- tbESQL/COBOL 프로그램의 성능 향상

  클라이언트와 서버 환경에서는 데이터를 주고 받기 위하여 많은 시간이 필요하다. 배열 변수를 사용해 여러 로우를 한꺼번에 처리하면 전송 시간이 크게 줄어든다.

4.2. 배열 변수 선언

tbESQL/COBOL 문장에서 사용될 배열 변수는 대체로 COBOL 프로그램에서 배열 변수를 선언하는 방법과 같다. 다만, tbESQL/COBOL 문장에 사용될 배열 변수를 선언할 때에는 반드시 DECLARE 영역 안에 선언해야 한다.

다음은 배열 변수를 선언하는 예이다.

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
   01 ENAME  PIC X(24) OCCURS 50 TIMES VARYING.
   01 SALARY PIC S9(5) OCCURS 50 TIMES.
   01 ADDR   PIC X(32) OCCURS 50 TIMES VARYING.
...
EXEC SQL END DECLARE SECTION END-EXEC.
```

4.3. 입/출력 배열 변수

본 절에서는 tbESQL/COBOL 문장에서 배열 변수가 입/출력 변수로 사용될 경우 각 tbESQL/COBOL 문장에 따른 배열 변수의 사용 방법에 대해 설명한다.

4.3.1. SELECT

SELECT 문장을 실행하여 반환되는 결과 로우의 개수가 하나 이상이라면 반드시 커서 또는 **배열 변수**를 사용해야 한다. 결과 로우의 개수를 미리 예측할 수 있다면 배열 변수의 크기를 충분히 선언하여 사용할 수 있다. 만약 결과 로우의 개수가 예측하기 힘들거나 매우 크다면, **배열 변수와 커서를 함께** 사용할 수도 있다.

배열 변수

SELECT 문장의 실행 결과로 반환된 로우의 개수가 예측 가능하고 크기가 크지 않다면 **배열 변수**를 이용하여 모든 결과 로우를 한번에 받을 수 있다. 예를 들어 SELECT 문장의 결과로 반환되는 로우의 개수가 50개를 넘지 않는 경우에 다음과 같이 소스 코드를 작성할 수 있다.

[예 4.1] SELECT 문장에서의 배열 변수의 사용

```
01 ENAME  PIC X(24) OCCURS 50 TIMES VARYING.
01 SALARY PIC S9(5) OCCURS 50 TIMES.
01 ADDR   PIC X(32) OCCURS 50 TIMES VARYING.
...
EXEC SQL SELECT ENAME, SALARY, ADDR
         INTO :ENAME, :SALARY, :ADDR
         FROM EMP
         WHERE SALARY >= 50000
END-EXEC.
```

결과 로우의 각 컬럼 값은 각 배열 변수에 저장되며, 저장될 때 같은 순서의 변수에 저장된다. 예를 들어 세 번째 로우의 컬럼 값은 각각 배열 변수의 세 번째 위치인 ENAME(3), SALARY(3), ADDR(3)에 저장된다.

SELECT 문장에서 배열 변수를 사용할 때는 다음의 사항에 유의해야 한다.

- INTO 절에 포함되는 출력 변수는 배열 변수와 일반 변수가 동시에 **함께 올 수 없다.**

  따라서 INTO 절에 포함된 모든 출력 변수는 배열 변수이거나 또는 일반 변수이어야 한다.

- SELECT 문장의 실행 결과로 반환된 로우의 전부가 아닌 **일부**만을 얻고 싶을 때에도 배열 변수를 이용할 수 있다. 위의 [예 4.1]에서 SELECT 문장의 실행 결과로 반환된 로우가 50개가 넘는 경우에 50개까지의 결과 로우만을 배열 변수에 저장한다.

- SELECT 문장의 실행 결과로 배열 변수에 저장된 로우의 실제 개수는 변수인 **SQLERRD(3) IN SQLCA**를 통해 알 수 있다.

  다음은 SELECT 문의 실행 후에 올 수 있는 소스 코드에 SQLERRD(3) IN SQLCA를 사용한 예이다.

```
PERFORM VARYING CNT FROM 1 BY 1
                     UNTIL ( CNT > SQLERRD(3) IN SQLCA )
        DISPLAY ENAME(CNT) ', ' SALARY(CNT) ', ' ADDR(CNT)
END-PERFORM.
DISPLAY 'number of returned rows = ' SQLERRD(3) IN SQLCA
```

  맨 마지막 라인을 보면 반환된 로우의 개수를 출력하는 데 SQLERRD(3) IN SQLCA를 사용하였다.

- 배열 변수의 크기가 일정하지 않은 경우에는 **가장 작은** 배열 변수의 크기를 전체 배열 변수의 크기로 설정한다. 예를 들어 다음의 소스 코드와 같이 배열 변수가 선언된 경우에 배열 변수 ENAME과 SALARY의 크기는 50이지만, ADDR의 크기가 30이므로, SELECT 문의 실행 후에 반환되는 결과 로우의 개수는 30개이다.

```
01 ENAME  PIC X(24) OCCURS 50 TIMES VARYING.
01 SALARY PIC S9(5) OCCURS 50 TIMES.
01 ADDR   PIC X(32) OCCURS 30 TIMES VARYING.
...
EXEC SQL SELECT ENAME, SALARY, ADDR
         INTO :ENAME, :SALARY, :ADDR
         FROM EMP
         WHERE SALARY >= 50000
END-EXEC.
```

- SELECT 문장의 **WHERE** 절에는 배열 변수가 올 수 없다. WHERE 절에 배열 변수가 오는 경우 질의의 의미가 모호해지기 때문이다.

  따라서 다음과 같은 소스 코드는 프리컴파일 과정에서 에러를 반환한다.

```
01 DEPTNO PIC S9(9) OCCURS 50 TIMES.
...
```

```
EXEC SQL SELECT ENAME, SALARY, ADDR
         INTO :ENAME, :SALARY, :ADDR
         FROM EMP
         WHERE DEPTNO = :DEPTNO
END-EXEC.
```

배열 변수와 함께 커서를 사용하더라도 SELECT 문장의 WHERE 절에 배열 변수를 사용할 수는 없다.

배열 변수와 커서

SELECT 문장의 실행 결과로 반환되는 로우의 개수를 예측하기 어렵거나 반환되는 로우의 개수가 많다면 배열 변수와 함께 **커서**를 이용하여야 한다. 이때 일반 커서는 물론 스크롤 가능 커서도 사용할 수 있다.

커서와 배열 변수를 함께 사용하는 방법은 커서와 일반 변수를 함께 사용하는 경우와 거의 유사하다. 하지만 **FETCH**를 실행할 때 루프를 빠져 나오는 방법이 일반 변수를 사용할 때와 다르다. 그 이유는 NOT FOUND 에러가 발생되는 경우가 차이가 있기 때문이다.

일반적으로 커서를 이용하여 루프 내에서 결과 로우를 액세스할 때 더 이상 읽어 올 결과 로우가 없으면 NOT FOUND 에러가 반환된다. 하지만 일반 변수일 때와 배열 변수일 때 **NOT FOUND 에러**는 다음과 같은 차이가 있다.

구분	설명
일반 변수	일반 변수를 이용할 때에는 결과 로우를 하나씩 액세스하므로 NOT FOUND 에러가 발생할 때에는 출력 변수에 저장된 결과 로우는 없다.
배열 변수	배열 변수를 이용할 때에는 배열 변수의 크기보다 작은 수의 결과 로우를 반환하더라도 NOT FOUND 에러를 반환한다. 즉, NOT FOUND 에러를 반환하더라도 출력 배열 변수에는 결과 로우가 포함되어 있을 수 있다.

따라서 배열 변수를 사용할 때는 SQLCA의 변수를 사용해 루프를 빠져 나온다. 커서와 함께 배열 변수가 사용될 때에 SQLERRD(3) IN SQLCA에는 FETCH 문장을 수행할 때마다 현재까지 처리된 결과 로우의 누적 개수가 저장된다. 그러므로 이 누적 개수가 더 이상 증가하지 않을 때 루프를 중단하면 된다.

다음은 루프를 중단하는 소스 코드의 예이다.

[예 4.2] SQLERRD(3) IN SQLCA를 활용한 루프의 중단

```
01 IDX           PIC S9(9).
01 COUNT         PIC S9(9).
01 BEFORE-COUNT  PIC S9(9).
01 CURRENT-COUNT PIC S9(9).
...
EXEC SQL DECLARE CUR CURSOR FOR ...
...
MOVE 0 TO BEFORE-COUNT.
MOVE 0 TO CURRENT-COUNT.
```

```
...
FETCH.
   EXEC SQL FETCH CUR INTO ...
   MOVE SQLERRD(3) IN SQLCA TO CURRENT-COUNT.

   IF (CURRENT-COUNT = BEFORE-COUNT) GOTO FETCH-END.
   COMPUTE COUNT = CURRERNT-COUNT - BEFORE-COUNT.

   PERFORM VARYING IDX FROM 1 BY 1 UNTIL ( IDX > COUNT )
* 각 로우에 대한 처리
   END-PERFORM.

    MOVE CURRENT-COUNT TO BEFORE-COUNT.
    GO TO FETCH.
FETCH-END.
```

위의 예에서는 두 개의 새로운 변수 BEFORE-COUNT와 CURRENT-COUNT를 사용하여 두 변수의 값이 일치하면 FETCH 루프를 중단한다. 출력 배열 변수에 저장된 실제 결과 로우의 개수는 변수 COUNT에 저장된다.

다수의 커서

SELECT 문장에는 배열 변수와 함께 **다수의 커서**를 사용할 수도 있다. 몇 개의 커서를 사용하더라도 하나의 커서를 사용할 때와 동일하게 처리된다. 동시에 여러 개의 커서를 사용할 때 SQLCA의 변수는 각 커서마다 별도로 선언되지는 않는다. 따라서 변수 SQLCA에 저장된 데이터는 직전에 실행된 질의 또는 기타 SQL 문장의 결과에 대한 데이터이다.

다음은 각 커서에 FETCH를 수행할 때마다 변수 SQLCA에 저장되는 데이터의 예이다.

```
EXEC SQL DECLARE CUR1 CURSOR FOR ...
EXEC SQL DECLARE CUR2 CURSOR FOR ...
...
EXEC SQL OPEN CUR1 END-EXEC.
EXEC SQL OPEN CUR2 END-EXEC.
...
EXEC SQL FETCH CUR1 INTO :ENAME END-EXEC.
* SQLERRD(3) = 20
EXEC SQL FETCH CUR2 INTO :SALARY END-EXEC.
* SQLERRD(3) = 30, not 50
EXEC SQL FETCH CUR1 INTO :ENAME END-EXEC.
* SQLERRD(3) = 40, not 70
EXEC SQL FETCH CUR1 INTO :ENAME END-EXEC.
* SQLERRD(3) = 60, not 90
EXEC SQL FETCH CUR2 INTO :SALARY END-EXEC.
* SQLERRD(3) = 60, not 120
```

위의 예에서 배열 변수 ENAME과 SALARY는 각각 크기가 20과 30이며, 커서 CUR1과 CUR2에 연관된 SELECT 문장의 결과 로우의 개수가 충분히 크다고 가정한다.

스크롤 가능 커서

SELECT 문장에는 배열 변수와 함께 **스크롤 가능 커서**를 사용할 수도 있다. 스크롤 가능 커서는 일반 커서의 경우와 거의 동일하게 사용할 수 있으나, **SQLERRD(3) IN SQLCA**에 저장되는 값의 의미가 달라진다.

일반 커서와 스크롤 가능 커서의 SQLERRD(3) IN SQLCA 값의 차이는 다음과 같다.

구분	설명
일반 커서의 SQLERRD(3) IN SQLCA	일반 커서의 경우에는 액세스된 결과 로우의 **누적 개수**를 저장하고 있다.
스크롤 가능 커서의 SQLERRD(3) IN SQLCA	현재까지 액세스된 가장 마지막 결과 로우의 **절대 위치**를 저장하고 있다.

스크롤 가능 커서를 사용할 때 SQLERRD(3) IN SQLCA에 저장되는 값은 다음과 같이 결정된다.

FETCH 문장을 수행할 때마다 옵션에 의하여 정해진 결과 로우의 위치로부터 배열 변수의 크기만큼 액세스하게 되므로, SQLERRD(3) IN SQLCA에 저장되는 값은(액세스하고자 하는 결과 로우의 절대 위치 + 배열 변수의 크기 - 1) 값 중에서 현재까지 가장 큰 값이 된다. 만약 액세스하려는 위치에서부터 남아 있는 결과 로우의 개수가 배열 변수의 크기보다 작다면 SQLERRD(3) IN SQLCA에 저장되는 값은 전체 결과 로우의 개수가 된다. 결과 로우의 절대 위치 값과 SQLERRD(3) IN SQLCA 값은 항상 1 이상이다.

다음의 소스 코드는 배열 변수와 함께 스크롤 가능 커서를 사용하는 예이다.

```
EXEC SQL DECLARE CUR SCROLL CURSOR FOR ...
...
EXEC SQL OPEN CUR END-EXEC.
...

EXEC SQL FETCH CUR INTO :ENAME END-EXEC.
* 1번째 로우부터 액세스. SQLERRD(3) = 20

EXEC SQL FETCH NEXT CUR INTO :ENAME END-EXEC.
* 21번째 로우부터 액세스. SQLERRD(3) = 40

EXEC SQL FETCH ABSOLUTE 11 CUR INTO :ENAME END-EXEC.
* 11번째 로우부터 액세스. SQLERRD(3) = 40

EXEC SQL FETCH RELATIVE 20 CUR INTO :ENAME END-EXEC.
* 51번째 로우부터 액세스. SQLERRD(3) = 70
```

```
EXEC SQL FETCH ABSOLUTE 41 CUR INTO :ENAME END-EXEC.
* 41번째 로우부터 액세스. SQLERRD(3) = 70

EXEC SQL FETCH RELATIVE 20 CUR INTO :ENAME END-EXEC.
* 81번째 로우부터 액세스. SQLERRD(3) = 90

EXEC SQL FETCH ABSOLUTE 51 CUR INTO :ENAME END-EXEC.
* 51번째 로우부터 액세스. SQLERRD(3) = 90
```

위의 예에서 배열 변수 ENAME의 크기는 20이며, 커서 CUR에 연관된 SELECT 문장의 결과로 반환된 로우의 전체 개수는 90이라고 가정한다.

지시자 배열 변수

SELECT 문장에는 출력 배열 변수와 함께 **지시자 배열 변수**를 사용할 수 있다.

지시자 배열 변수는 출력 배열 변수와 같은 크기를 가져야 하며, 배열 변수로 선언한다는 것 외에는 일반적인 지시자 변수와 동일하게 사용할 수 있다. 반환되는 컬럼 값이 NULL이거나 값의 일부가 잘릴 가능성이 있다면 지시자 배열 변수의 사용을 고려해야 한다. 지시자 배열 변수도 다른 변수와 마찬가지로 DECLARE 영역 내에 선언한다.

다음은 지시자 배열 변수를 사용한 예이다.

```
01 ADDR     PIC X(24) OCCURS 50 TIMES VARYING.
01 ADDR-IND PIC S9(4) OCCURS 50 TIMES COMP-5.
...
EXEC SQL SELECT ADDR, ...
         INTO :ADDR INDICATOR :ADDR-IND, ...
         FROM EMP
         WHERE DEPTNO = 5
END-EXEC.
```

참고

입력 배열 변수와 함께 지시자 배열 변수를 사용할 수도 있다. 입력 지시자 변수는 INSERT, UPDATE, DELETE 문장에서 사용할 수 있다.

사용 예제

다음은 [예 3.11]에서 제시한 예제를 배열 변수를 이용하도록 수정한 예이다.

```
IDENTIFICATION                    DIVISION.
************************************************************
PROGRAM-ID.                       TEST.
************************************************************
```

```
ENVIRONMENT                         DIVISION.
************************************************************
CONFIGURATION                       SECTION.
*
SOURCE-COMPUTER.                    TEST-MACHINE.
OBJECT-COMPUTER.                    TEST-MACHINE.
************************************************************
DATA                                DIVISION.
************************************************************
WORKING-STORAGE                     SECTION.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 USERPASS PIC X(20) VALUE Z"tibero/tmax".
01 ENAME    PIC X(24) OCCURS 30 TIMES VARYING.... ① ...
01 SALARY   PIC S9(5) OCCURS 30 TIMES.
01 ADDR     PIC X(32) OCCURS 30 TIMES VARYING.
01 DEPTNO   PIC S9(9).
EXEC SQL END DECLARE SECTION END-EXEC.

01 IDX           PIC S9(9).
01 COUNT         PIC S9(9).
01 BEFORE_COUNT  PIC S9(9).
01 CURRENT_COUNT PIC S9(9).

EXEC SQL INCLUDE SQLCA END-EXEC.

PROCEDURE             DIVISION.

EXEC SQL DECLARE EMP-CURSOR CURSOR FOR
         SELECT ENAME, SALARY, ADDR
         FROM EMP
         WHERE DEPTNO = :DEPTNO
END-EXEC.

EXEC SQL CONNECT :USERPASS END-EXEC.
DISPLAY 'Connected.'.
MOVE 10 TO DEPTNO.

EXEC SQL OPEN EMP-CURSOR END-EXEC.
MOVE 0 TO BEFORE-COUNT.
MOVE 0 TO CURRENT-COUNT.

FETCH.
   EXEC SQL FETCH EMP-CURSOR
            INTO :ENAME, :SALARY, :ADDR
   END-EXEC.
```

```
   MOVE SQLERRD(3) IN SQLCA TO CURRENT-COUNT.

   IF (CURRENT-COUNT = BEFORE-COUNT) GOTO FETCH-END.... ② ...

   COMPUTE COUNT = CURRERNT_COUNT - BEFORE-COUNT.

   PERFORM VARYING IDX FROM 1 BY 1 UNTIL ( IDX > COUNT )
           DISPLAY 'ENAME = ' ENAME-ARR(IDX)... ③ ...
                   ', SALARY = ' SALARY(IDX)
                   ', ADDR = ' ADDR-ARR(IDX)
   END-PERFORM.

   MOVE CURRENT-COUNT TO BEFORE-COUNT.
   GO TO FETCH.
FETCH-END.

EXEC SQL CLOSE EMP-CURSOR END-EXEC.
EXEC SQL COMMIT WORK RELEASE END-EXEC.
```

① [예 3.11]과 비교했을 때 변수를 배열 변수로 선언한다.

② FETCH를 수행할 때 루프를 중단한다.

③ 배열 변수에 저장된 컬럼의 값을 DISPLAY 문을 통해 출력한다.

4.3.2. INSERT

INSERT 문장에서 배열 변수를 사용하는 방법은 일반적인 입력 변수를 사용하는 방법과 동일하다. 입력 배열 변수가 사용되는 각 문장은 문장이 실행되기 직전에 동적으로 배열 변수에 저장된 값을 읽어 들인다. 따라서 각 문장이 실행되기 전에 반드시 배열 변수에 적절한 값을 넣어 주는 코드가 있어야 한다.

INSERT 문장에서 배열 변수를 사용할 때는 다음의 사항에 유의해야 한다.

- INSERT 문장에서 사용되는 입력 변수는 배열 변수와 일반 변수가 함께 올 수 없다. 따라서 모든 입력 변수가 배열 변수이거나 일반 변수로만 구성되어야 한다.
- SELECT 문장과 유사하게 SQLERRD(3) IN SQLCA에는 삽입된 로우의 개수가 저장되어 있다. SQLERRD(3) IN SQLCA에는 항상 하나의 INSERT 문장에 의해 삽입된 로우의 개수만 저장된다.
- FOR 절을 사용해 입력 배열 변수의 크기보다 적은 개수의 로우를 삽입하도록 할 수도 있다.

  예를 들면 크기가 50인 입력 배열 변수의 내용 중에서 30개의 로우 값만을 이용하는 경우이다.
- 입력 배열 변수와 함께 지시자 배열 변수를 사용할 수도 있다. 삽입해야 할 컬럼 값 중 일부만 NULL 값인 경우에는 반드시 지시자 배열 변수를 사용해야 한다.

다음은 배열 변수와 함께 INSERT 문장을 실행하는 예이다.

[예 4.3] INSERT 문장의 배열 변수

```
01 ENAME  PIC X(24) OCCURS 50 TIMES VARYING.
01 SALARY PIC S9(5) OCCURS 50 TIMES.
...
MOVE "BROWN" TO ENAME(1).
MOVE "30000" TO SALARY(1).
MOVE "WHITE" TO ENAME(2).
MOVE "28000" TO SALARY(2).
...
EXEC SQL INSERT INTO EMP (ENAME, SALARY)
         VALUES (:ENAME, :SALARY)
END-EXEC.
```

위의 [예 4.3]에서의 INSERT 문장은 다음의 루프를 이용한 소스 코드와 같은 결과를 갖는다. 하지만 배열 변수를 이용하는 편이 서버와의 통신 비용을 크게 줄일 수 있으므로 성능 면에서 훨씬 효율적이다.

```
PERFORM VARYING IDX FROM 1 BY 1 UNTIL ( IDX > 50 )
        EXEC SQL INSERT INTO (ENAME, SALARY)
                 VALUES (:ENAME(IDX), :SALARY(IDX))
        END-EXEC.
END-PERFORM.
```

다음은 컬럼 SALARY에 대해 지시자 배열 변수를 사용한 예이다.

```
01 ENAME      PIC X(24) OCCURS 50 TIMES VARYING.
01 SALARY     PIC S9(5) OCCURS 50 TIMES.
01 SALARY-IND PIC S9(4) COMP-5 OCCURS 50 TIMES.
...
EXEC SQL INSERT INTO EMP (ENAME, SALARY)
         VALUES (:ENAME, :SALARY:SALARY-IND)
END-EXEC.
```

4.3.3. UPDATE

UPDATE 문장에서 배열 변수를 사용하는 방법도 일반 변수를 사용하는 방법과 유사하다.

UPDATE 문장에서 배열 변수를 사용할 때는 다음의 사항에 유의해야 한다.

- UPDATE 문장에서는 SELECT 문장과는 다르게 **WHERE 절**에도 배열 변수를 사용할 수 있다.

WHERE 절에 배열 변수를 사용하면 SET 절에도 배열 변수를 사용해야 한다. 만약 WHERE 절에 일반 변수를 사용하였다면 SET 절에도 일반 변수를 사용하여야 한다.

- INSERT 문장과 유사하게 SQLERRD(3) IN SQLCA에는 갱신된 로우의 개수가 저장된다. SQLERRD(3) IN SQLCA에는 항상 하나의 UPDATE 문장에 의해 갱신된 로우의 개수만 저장되며, 무결성 제약조건을 만족하기 위해 연속적으로 갱신되거나 삭제된 로우의 개수는 포함되지 않는다.
- UPDATE 문장에서 SET 절 내에 사용된 입력 변수에는 배열 변수와 일반 변수가 함께 올 수 없다.
- SELECT FOR UPDATE 문장 내에서 **UPDATE ... CURRENT OF 절**과 함께 배열 변수를 사용할 수 없다.

다음은 UPDATE 문장의 SET 절과 WHERE 절에 배열 변수를 사용한 예이다.

[예 4.4] UPDATE 문장의 배열 변수

```
01 SALARY PIC S9(5) OCCURS 50 TIMES.
01 EMPNO  PIC S9(9) OCCURS 50 TIMES.
...
MOVE "12000" TO SALARY(1).
MOVE "22000" TO SALARY(2).
MOVE "18000" TO SALARY(3).
MOVE "23000" TO SALARY(4).
MOVE "30000" TO SALARY(5).

MOVE "23401" TO EMPNO(1).
MOVE "12089" TO EMPNO(2).
MOVE "13560" TO EMPNO(3).
MOVE "32109" TO EMPNO(4).
MOVE "10094" TO EMPNO(5).
...
EXEC SQL UPDATE EMP SET SALARY = :SALARY
         WHERE EMPNO = :EMPNO
END-EXEC.
```

위의 [예 4.4]은 다음과 같은 루프를 이용한 연속된 UPDATE 문장과 동일한 결과를 갖는다. 하지만 배열 변수를 이용하는 편이 성능 면에서 더욱 효율적이다.

```
PERFORM VARYING IDX FROM 1 BY 1 UNTIL ( IDX > 50 )
        EXEC SQL UPDATE EMP SET SALARY = :SALARY(IDX)
                 WHERE EMPNO = :EMPNO(IDX)
        END-EXEC.
END-PERFORM.
```

4.3.4. DELETE

DELETE 문장에서 배열 변수를 사용하는 방법도 일반 변수를 사용하는 방법과 유사하다.

DELETE 문장에서 배열 변수를 사용할 때는 다음의 사항에 유의해야 한다.

- DELETE 문장에서도 UPDATE 문장과 같이 **WHERE 절**에 배열 변수를 사용할 수 있다.
- INSERT, UPDATE 문장에서와 유사하게, SQLERRD(3) IN SQLCA에는 삭제된 로우의 개수가 저장된다. SQLERRD(3) IN SQLCA에는 항상 하나의 DELETE 문장에 의하여 삭제된 로우의 개수만 저장되며, 무결성 제약조건을 만족하기 위하여 연속적으로 갱신되거나 삭제된 로우의 개수는 포함되지 않는다.
- DELETE 문장의 WHERE 절에 사용될 입력 변수로 배열 변수와 일반 변수가 함께 올 수 없다.
- DELETE 문장에 배열 변수를 사용할 때는 **DELETE ... CURRENT OF 절**을 사용할 수 없다.

다음은 DELETE 문장의 WHERE 절에 배열 변수를 사용한 예이다.

[예 4.5] DELETE 문장의 배열 변수

```
01 EMPNO PIC S9(9) OCCURS 50 TIMES.
...
MOVE "15009" TO EMPNO(1).
MOVE "13450" TO EMPNO(2).
MOVE "24200" TO EMPNO(3).
MOVE "15832" TO EMPNO(4).
MOVE "20009" TO EMPNO(5).
MOVE "30110" TO EMPNO(6).
...
EXEC SQL DELETE EMP WHERE EMPNO = :EMPNO
END-EXEC.
```

위의 [예 4.5]은 다음의 루프와 동일한 결과를 갖는다. 하지만 INSERT, UPDATE 문장과 마찬가지로 배열 변수를 이용하는 편이 성능 면에서 효율적이다.

```
PERFORM VARYING IDX FROM 1 BY 1 UNTIL ( IDX > 50 )
        EXEC SQL DELETE EMP
                 WHERE EMPNO = :EMPNO(IDX)
        END-EXEC.
END-PERFORM.
```

4.3.5. FOR 절

INSERT, DELETE, UPDATE 문장에서 입력 배열 변수를 사용할 때 배열 변수의 크기보다 적은 개수의 로우를 처리하려는 경우에 **FOR 절**을 사용한다. FOR 절은 EXEC SQL 바로 다음에 오며, 처리하고자 하는 로우의 개수를 명시한다.

FOR 절에 로우의 개수를 명시할 때 다음의 사항에 유의해야 한다.

- FOR 절에 로우의 개수를 명시할 때는 **숫자**를 명시할 수도 있으며, **변수**를 명시할 수도 있다.

  변수를 명시할 경우 해당 변수는 반드시 **DECLARE 영역**에 선언되어 있어야 한다.

- FOR 절에 로우의 개수를 명시할 때 연산식을 사용해서는 안 된다.

- FOR 절의 로우 개수는 항상 입력 배열 변수의 크기보다 작아야 한다.

  tbESQL/COBOL 프로그램에서는 지정된 로우의 개수를 저장할 배열 변수의 크기를 검토하지 않는다.

  만약 지정된 로우의 개수보다 배열 변수의 크기가 작다면, 내부적으로 유효하지 않은 메모리에 접근하게 되어 메모리 에러가 발생하고, 이때 tbESQL/COBOL 프로그램이 어떠한 동작을 할지 보장할 수 없다.

다음의 소스 코드는 FOR 절을 이용하는 예이다.

```
01 ENAME     PIC X(24) OCCURS 50 TIMES VARYING.
01 SALARY    PIC S9(5) OCCURS 50 TIMES.
01 EMPNO     PIC S9(9) OCCURS 50 TIMES.
01 ROW-COUNT PIC S9(9).
...
EXEC SQL FOR 20... ① ...
         INSERT INTO EMP (ENAME, SALARY, ADDR)
         VALUES (:ENAME, :SALARY, NULL)
END-EXEC.
...
MOVE 30 TO ROW-COUNT.
EXEC SQL FOR :ROW-COUNT... ② ...
         UPDATE EMP SET SALARY = :SALARY
         WHERE EMPNO = :EMPNO
END-EXEC.
```

① 로우의 개수를 숫자로 설정하여 FOR 절을 이용할 수 있다.

② 변수를 이용하여 로우의 개수를 설정하여 FOR 절을 이용할 수 있다. 단, 변수 ROW-COUNT는 DECLARE 영역에 선언되어 있어야 한다.

다음은 FOR 절에서 로우의 개수를 명시할 때 임의의 연산식을 사용한 경우로 잘못된 예이다.

```
EXEC SQL FOR( :ROW-COUNT + 10 )
         DELETE EMP WHERE EMPNO = :EMPNO
END-EXEC.
```

4.4. 구조체 배열 변수

tbESQL/COBOL 문장 내에서 여러 컬럼을 동시에 처리할 때 각 컬럼별 입/출력 변수 각각을 나열할 수도 있지만, 각 컬럼에 대한 변수를 모아 하나의 구조체로 정의하여 구조체 타입의 변수를 이용할 수도 있다.

또한 더 나아가 각 컬럼의 여러 로우를 한꺼번에 처리하고자 할 때 구조체로 정의한 타입을 배열로 선언하여, **구조체 배열 변수**를 이용할 수 있다.

4.4.1. 구조체 배열 변수의 선언

구조체 배열 변수를 선언할 때는 다음의 사항에 유의해야 한다.

- 구조체 배열 변수는 반드시 **DECLARE 영역**에 선언되어야 한다.
- tbESQL/COBOL 문장에서 사용되는 구조체 타입의 변수는 중첩되어 정의될 수 없다(일반적인 구조체 변수일 때나 구조체 배열 변수일 때나 동일하다).
- 구조체 변수의 그 하위 멤버는 배열일 수 없다(일반적인 구조체 변수일 때나 구조체 배열 변수일 때나 동일하다).

다음은 구조체 배열 변수를 선언하는 예이다.

```
01 EMP OCCURS 50 TIMES.
   03 ENAME  PIC X(24) VARYING.
   03 SALARY PIC S9(5).
   03 ADDR   PIC X(32) VARYING.
```

다음은 구조체를 잘못 선언한 예이다.

```
01 EMP OCCURS 50 TIMES.
   03 ENAME PIC X(24) VARYING.
   03 SUB-EMP.
      05 SALARY PIC S9(5).
      05 ADDR   PIC X(32) VARYING.
```

위의 예에서는 구조체 EMP 내부에 구조체 SUB_EMP를 중첩하여 정의하였다. 구조체 타입의 변수는 중첩되어 정의될 수 없다.

다음은 구조체 타입 변수의 하위 멤버를 배열로 선언한 경우로 잘못 선언한 예이다.

```
01 ENAME
   03 FIRST-NAME OCCURS 5 TIMES
   03 LAST-NAME  OCCURS 5 TIMES

01 ENAME-ARR OCCURS 10 TIMES
   03 FIRST-NAME OCCURS 5 TIMES
   03 LAST-NAME  OCCURS 5 TIMES
```

위의 예에서 ENAME과 ENAME-ARR의 하위 멤버로 각각 배열을 선언하였다. 구조체 타입의 변수는 하위 멤버로 배열이 올 수 없다.

4.4.2. 사용 방법

단순 구조체 타입의 변수와 마찬가지로 구조체 배열 변수는 SELECT 문장에서의 출력 변수와 INSERT 문장에서의 입력 변수로 사용할 수 있다.

구조체 배열 변수를 사용하는 방법은 일반적인 배열 변수와 동일하며, 배열 변수의 사용 방법에 대한 내용은 구조체 배열 변수에도 동일하게 해당된다. 단순히 구조체 배열 변수만을 사용할 수도 있으며 커서와 함께 사용할 수도 있다.

다음은 구조체 배열 변수를 입/출력 변수로 사용하는 예이다.

```
01 EMP OCCURS 50 TIMES.
   03 ENAME  PIC X(24) VARYING.
   03 SALARY PIC S9(5).
   03 ADDR   PIC X(32) VARYING.
       ...
       EXEC SQL SELECT ENAME, SALARY, ADDR
                INTO :EMP
                FROM EMP
                WHERE SALARY >= 50000
       END-EXEC.
       ...
       EXEC SQL INSERT INTO EMP
                      VALUES(:EMP)
       END-EXEC.
```

제5장 tbESQL/COBOL 문장

본 장에서는 tbESQL/COBOL 프로그램에서 데이터베이스 처리를 위해 사용하는 tbESQL/COBOL 문장을 설명한다.

5.1. 개요

tbESQL/COBOL 프로그램은 COBOL 언어의 소스 코드와 tbESQL/COBOL 문장이 혼합되어 있다. **tbESQL/COBOL 문장**(tbESQL/COBOL Statement)은 tbESQL/COBOL 프로그램 내에서 SQL 질의, 로우의 삽입과 갱신, 제거 등과 같은 데이터베이스와 관련된 처리를 하는 문장을 말한다.

tbESQL/COBOL 문장은 크게 다음과 같이 두 가지로 구성된다.

- **실행 문장**(Executable Statements)

  실행 문장은 말 그대로 데이터베이스에 어떠한 동작을 실행하기 위한 문장이다.

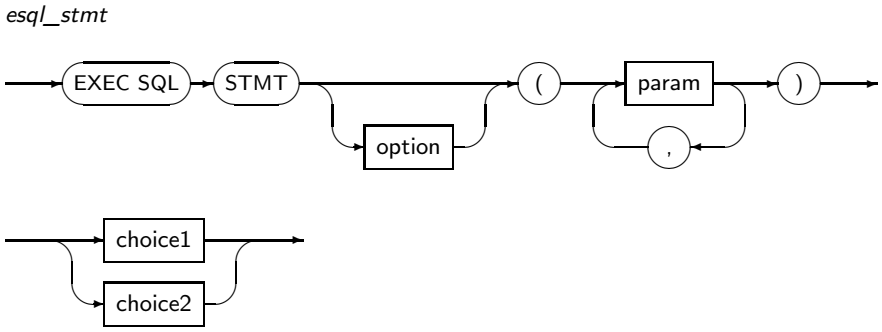
- **지시어**(Directive)

  지시어는 tbESQL/COBOL 프로그램의 실행을 설정하기 위한 것으로 데이터베이스에 대한 실행은 이루어지지 않는다.

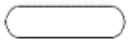
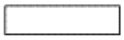
5.2. tbESQL/COBOL 문장 문법

다음은 tbESQL/COBOL 문장의 문법을 나타내는 그림이다.

[그림 5.1] tbESQL/COBOL 문장 문법



[그림 5.1]을 기준으로 tbESQL/COBOL 문장의 문법을 해석하는 방법은 다음과 같다.

항목	설명
esql_stmt	tbESQL/COBOL 문장의 문법을 대표하는 이름은 왼쪽 위에 나타낸다. (예: esql_stmt)
	타원 안의 포함되는 문자는 키워드(Keyword)이며, 반드시 tbESQL/COBOL 문장 내에 포함되어 있어야 한다. (예: EXEC SQL, STMT)
	사각형 안의 문자는 tbESQL/COBOL 문장에 포함된 문법의 구성요소이며, 구성요소에 맞는 적절한 문자열로 바꾸어야 한다. (예: option, param, choice1, choice2)
	원 안의 문자는 tbESQL/COBOL 문장의 기호이며, 반드시 tbESQL/COBOL 문장 내에 포함되어 있어야 한다. (예: 소괄호(( )), 콤마(,))
	tbESQL/COBOL 문장을 완성하는 순서는 화살표를 따라가면 된다. 여러 갈래로 갈라지면서 순방향으로 이동하는 화살표는 두 가지 형태로 띈다. [그림 5.1]에서 option은 포함되거나 포함되지 않은 경우이며, choice1, choice2는 여러 가지 중에 하나만이 tbESQL/COBOL 문장에 포함되어야 하는 경우이다. 또는 역방향으로 이동하는 화살표는 포함되지 않거나 한번 이상 포함되는 경우이다. [그림 5.1]에서 콤마(,)에 해당되며, param은 반드시 한번 이상 포함되어야 한다.

다음의 tbESQL/COBOL 문장은 [그림 5.1]에 따라 모두 유효한 예이다.

```
EXEC SQL STMT (param) choice1 END-EXEC.
EXEC SQL STMT option (param, param) choice1 END-EXEC.
EXEC SQL STMT (param, param) choice2 END-EXEC.
```

5.3. tbESQL/COBOL 문장 공통 문법

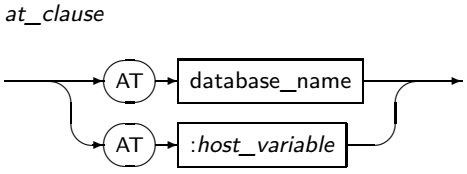
본 절에서는 tbESQL/COBOL 문장에서 공통적으로 자주 사용되는 부분을 설명한다.

5.3.1. AT 절

AT 절은 이름이 있는 데이터베이스 연결을 사용해 tbESQL/COBOL 문장을 수행할 때 사용한다.

AT의 세부 내용은 다음과 같다.

- 문법



- 구성요소

구성요소	설명
database_name	사용할 데이터베이스의 이름을 명시한다. 여러 개의 데이터베이스 접속을 구분하여 관리하고 싶을 때 database_name을 사용한다. 데이터베이스의 이름은 DECLARE DATABASE를 사용해 미리 선언되어 있어야 한다. 만약 선언되어 있지 않은 이름을 사용할 경우 에러가 발생한다.
:host_variable	사용할 데이터베이스의 이름이 저장된 호스트 변수를 명시한다. 데이터베이스의 이름은 미리 선언되어 있어야 한다.

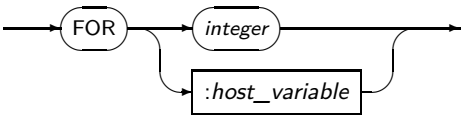
5.3.2. FOR 절

FOR 절은 tbESQL/COBOL 문장을 반복해서 수행할 필요가 있을 때 사용한다.

FOR 절의 세부 내용은 다음과 같다.

- 문법

for_clause



- 구성요소

구성요소	설명
integer	반복 횟수를 명시한다.
:host_variable	반복 횟수가 저장된 호스트 변수를 명시한다. 호스트 변수는 int 등의 숫자를 저장할 수 있는 타입이면 된다.

5.3.3. DESCRIPTOR 이름

DESCRIPTOR 이름은 Dynamic SQL을 사용할 때 필요한 DESCRIPTOR를 가리키는 이름이다.

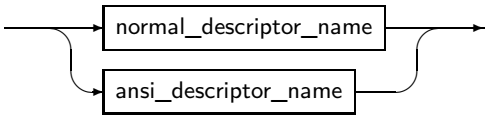
DESCRIPTOR 이름은 프리컴파일러 옵션 즉 MODE, DYNAMIC의 값에 따라 다음과 같이 사용될 수 있다.

옵션 값	설명
ANSI	ansi_descriptor_name이 사용된다.
ANSI 외	normal_descriptor_name이 사용된다.

DESCRIPTOR 이름의 세부 내용은 다음과 같다.

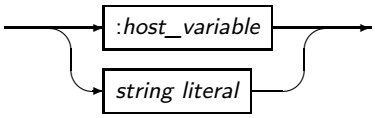
- 문법

descriptor_name



normal_descriptor_name

ansi_descriptor_name



- 구성요소

  – descriptor_name

구성요소	설명
normal_descriptor_name	MODE가 TIBERO(ORACLE)일 때 사용한다.
ansi_descriptor_name	MODE가 ANSI(ISO)일 때 사용한다.

  – normal_descriptor_name

구성요소	설명
identifier	서술자의 이름을 정의한 식별자이다.

  – ansi_descriptor_name

구성요소	설명
:host_variable	서술자의 이름이 저장된 호스트 변수를 명시한다. 호스트 변수는 CHAR* 또는 CHAR ARRAY 타입 등의 문자열을 저장할 수 있는 타입이다.
string literal	서술자의 이름을 작은따옴표(' ')로 감싸서 사용한다.

5.4. tbESQL/COBOL 문장 목록

본 절에서는 Tibero에서 제공하는 tbESQL/COBOL 문장을 알파벳 순으로 설명한다. 단, 각 tbESQL/COBOL 문장의 구성요소를 설명할 때 키워드는 특별한 경우가 아니면 설명하지 않는다.

다음은 tbESQL/COBOL 문장을 요약한 목록이다.

- 실행문장

tbESQL/COBOL문장	설명
ALLOCATE DESCRIPTOR	서술자에 메모리를 할당한다.
CLOSE	커서를 닫고 더 이상 사용하지 않는다.
COMMIT	트랜잭션을 커밋한다.
CONNECT	Tibero의 데이터베이스에 접속한다.
DEALLOCATE DESCRIPTOR	서술자에 할당된 메모리를 해제한다.
DELETE	로우를 삭제한다.
EXECUTE	준비된 Dynamic SQL 문장을 실행한다.
EXECUTE DESCRIPTOR	준비된 Dynamic SQL 문장을 실행한다. (ANSI)
EXECUTE IMMEDIATE	Dynamic SQL 문장을 바로 실행한다.
FETCH	커서를 이용하여 다음 로우를 읽는다.
FETCH DESCRIPTOR	커서를 이용하여 다음 로우를 읽는다. (ANSI)
GET DESCRIPTOR	지정한 서술자에서 원하는 정보를 가져온다.
INSERT	로우를 삽입한다.
OPEN	커서를 열고 연관된 SQL 문장을 실행한다.
PREPARE	Dynamic SQL 문장을 준비한다.
ROLLBACK	트랜잭션에 롤백을 수행한다.
SAVEPOINT	저장점(Savepoint)을 설정한다.
SELECT	SQL 질의를 수행한다.
UPDATE	로우를 갱신한다.

- 지시어

tbESQL/COBOL 문장	설명
DECLARE CURSOR	SQL 문장과 연관된 커서를 선언한다.
DECLARE DATABASE	새로운 데이터베이스 접속을 선언한다.
DESCRIBE	서술자를 초기화한다.
DESCRIBE DESCRIPTOR	서술자에 호스트 변수의 정보를 저장한다.
SET DESCRIPTOR	지정한 서술자에 사용자가 입력해야 할 정보를 쓴다.

tbESQL/COBOL 문장	설명
WHENEVER	에러가 발생했을 때 해당 에러에 대한 처리 방법을 지정한다.

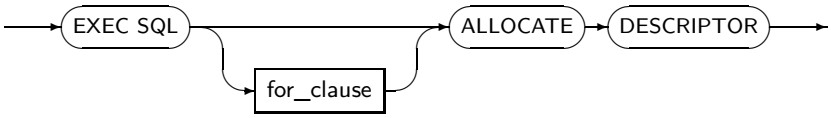
5.4.1. ALLOCATE DESCRIPTOR

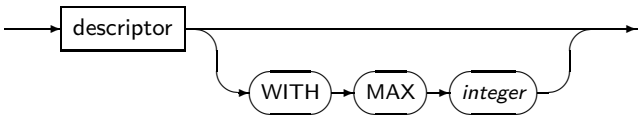
ALLOCATE DESCRIPTOR는 서술자에 메모리를 할당할 때 사용하는 문장이다. 단, 이 문장은 ANSI 타입의 Dynamic SQL 문장에만 사용할 수 있다.

ALLOCATE DESCRIPTOR의 세부 내용은 다음과 같다.

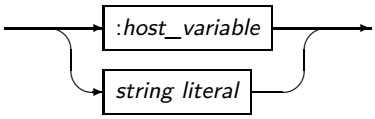
- 문법

allocate_descriptor_statement





descriptor



- 구성요소

  – allocate_descriptor_statement

구성요소	설명
for_clause	반복 횟수를 지정한다. 자세한 내용은 "5.3.2. FOR 절"을 참고한다.
descriptor	서술자의 이름이 저장된 호스트 변수나 문자열을 명시한다.
WITH MAX integer	사용할 호스트 변수의 최대 개수를 지정한다. (기본값: 100)

  – descriptor

구성요소	설명
:host_variable	서술자의 이름이 저장된 호스트 변수를 명시한다. 호스트 변수는 CHAR* 또는 CHAR ARRAY 타입 등의 문자열을 저장할 수 있는 타입이다.

구성요소	설명
string literal	서술자의 이름이 저장된 문자열을 명시한다.

- 예제

  다음은 ALLOCATE DESCRIPTOR를 사용하는 예이다.

```
EXEC SQL ALLOCATE DESSCRIPTOR 'IN-DESC' END-EXEC.

EXEC SQL ALLOCATE DESSCRIPTOR 'OUT-DESC' END-EXEC.
```

5.4.2. CLOSE

CLOSE는 커서를 닫을 때 사용하는 문장이다. 이 문장은 현재 열려 있는 커서에만 사용할 수 있다. 커서를 닫으면 커서를 생성할 때 할당되었던 모든 시스템 리소스가 반환된다.

CLOSE의 세부 내용은 다음과 같다.

- 문법

close_statement

cursor_name

- 구성요소

  – close_statement

구성요소	설명
cursor_name	닫으려는 커서의 이름을 명시한다.

  – cursor_name

구성요소	설명
identifier	커서의 이름을 정의한 식별자이다.

- 예제

  다음은 CLOSE를 사용하는 예이다.

```
EXEC CLOSE EMP-CURSOR END-EXEC.
```

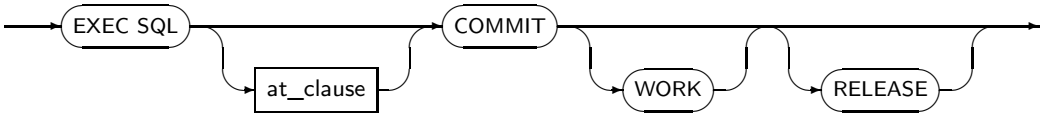
5.4.3. COMMIT

COMMIT은 현재 진행 중인 트랜잭션을 커밋하고 트랜잭션에 의해 갱신된 모든 내용을 데이터베이스에 반영할 때 사용하는 문장이다.

COMMIT의 세부 내용은 다음과 같다.

- 문법

commit_statement



- 구성요소

구성요소	설명
at_clause	커밋을 실행할 데이터베이스를 명시한다. 데이터베이스의 이름을 직접 명시할 수도 있고, 데이터베이스의 이름이 저장된 호스트 변수를 명시할 수도 있다. 자세한 내용은 "5.3.1. AT 절"을 참고한다.
WORK	기존 ESQL 프로그램과의 호환성을 위한 키워드이며 특별한 의미는 없다.
RELEASE	모든 리소스를 반환하고 데이터베이스 접속을 종료한다.

- 예제

다음은 COMMIT을 사용하는 예이다.

```
EXEC SQL COMMIT END-EXEC.

EXEC SQL COMMIT WORK RELEASE END-EXEC.
```

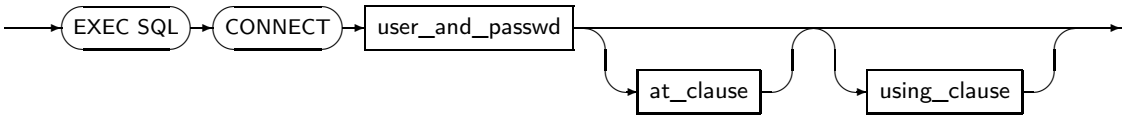
5.4.4. CONNECT

CONNECT는 데이터베이스에 접속할 때 사용하는 문장이다. 이 문장을 사용할 때는 사용자 이름과 패스워드를 반드시 명시해야 한다. 데이터베이스 관리자(DBA: Database Administrator, 이하 DBA)로 접속할 수도 있다.

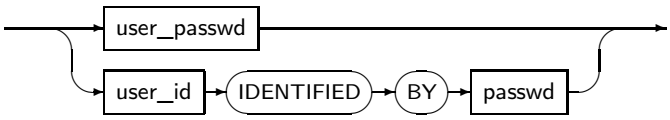
CONNECT의 세부 내용은 다음과 같다.

- 문법

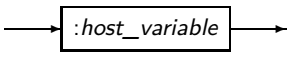
connect_statement



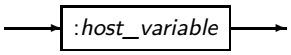
user_and_passwd



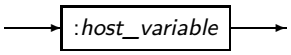
user_passwd



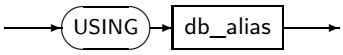
user_id



passwd



using_clause



- 구성요소
  - connect_statement

구성요소	설명
user_and_passwd	사용자 이름과 패스워드를 명시한다.
at_clause	접속할 데이터베이스를 명시한다. 데이터베이스의 이름을 직접 명시할 수도 있고, 데이터베이스의 이름이 저장된 호스트 변수를 명시할 수도 있다. 자세한 내용은 "5.3.1. AT 절"을 참고한다.
using_clause	데이터베이스의 별칭을 명시한다.

  - user_and_passwd

구성요소	설명
user_password	사용자 이름과 패스워드를 명시한다. 사용자 이름과 패스워드의 중간에 슬래시(/)를 포함해 하나의 문자열로 표현한다. (예: 'tibero/tibero')
user_id	사용자 이름을 명시한다. 호스트 변수이며, 문자열이 직접 올 수 없다.
passwd	패스워드를 명시한다. 호스트 변수이며, 문자열이 직접 올 수 없다.

– user_passwd

구성요소	설명
:host_variable	사용자 이름과 패스워드를 저장하고 있는 호스트 변수이다.

– user_id

구성요소	설명
:host_variable	사용자의 계정을 저장하고 있는 호스트 변수이다.

– passwd

구성요소	설명
:host_variable	패스워드를 저장하고 있는 호스트 변수이다.

– using_clause

구성요소	설명
db_alias	데이터베이스의 별칭을 명시한다.

- 예제

  다음은 CONNECT를 사용하는 예이다.

```
EXEC SQL CONNECT :USER IDENTIFIED BY :PASSWORD END-EXEC.
```

5.4.5. DEALLOCATE DESCRIPTOR

DEALLOCATE DESCRIPTOR는 ALLOCATE DESCRIPTOR를 사용해 할당된 서술자의 메모리를 해제할 때 사용하는 문장이다. 단, 이 문장은 ANSI 타입의 Dynamic SQL 문장에만 사용할 수 있다.

DEALLOCATE DESCRIPTOR의 세부 내용은 다음과 같다.

- 문법

  deallocate_descriptor_statement

- 구성요소

구성요소	설명
descriptor_name	서술자의 이름이 저장된 호스트 변수 또는 문자열을 명시한다.

구성요소	설명
	호스트 변수는 PIC X(n) 타입 등 문자열을 저장할 수 있는 타입으로 이미 선언되어 있어야 한다. 서술자는 ALLOCATE DESCRIPTOR를 통해 이미 할당되어 있어야 한다. 자세한 내용은 "5.3.3. DESCRIPTOR 이름"을 참고한다.

- 예제

  다음은 DEALLOCATE DESCRIPTOR를 사용하는 예이다.

```
EXEC SQL DEALLOCATE DESSCRIPTOR 'IN-DESC' END-EXEC.
```

5.4.6. DECLARE CURSOR

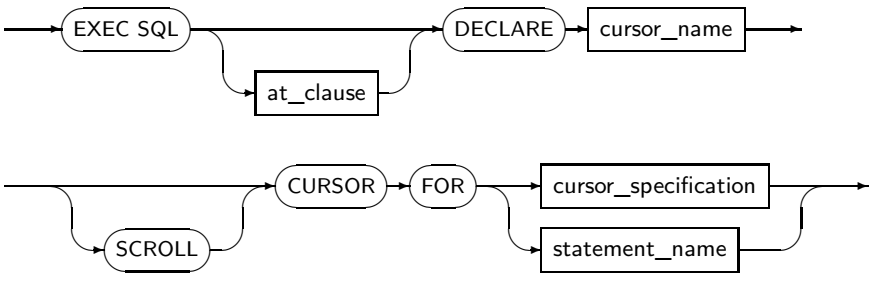
DECLARE CURSOR는 커서를 선언할 때 사용하는 문장이다. 이 문장을 사용해 스크롤 가능 커서(Scrollable Cursor)를 선언할 수도 있다. 커서를 선언할 때는 커서의 이름을 반드시 명시하여야 하며, SELECT 문장과 연관시켜야 한다.

Dynamic SQL 문장에 대한 커서인 경우에는 문장 이름(Statement Name)과 연관시킨다. 이때 문장 이름은 PREPARE를 통해 준비되어 있어야 한다.

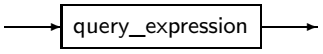
DECLARE CURSOR의 세부 내용은 다음과 같다.

- 문법

declare_cursor_statement



cursor_specification



- 구성요소

  – declare_cursor_statement

구성요소	설명
at_clause	문장을 실행할 데이터베이스를 명시한다.

구성요소	설명
	데이터베이스의 이름을 직접 명시할 수도 있고, 데이터베이스의 이름이 저장된 호스트 변수를 명시할 수도 있다. 자세한 내용은 “5.3.1. AT 절”을 참고한다.
cursor_name	커서의 이름을 명시한다.
SCROLL	스크롤 가능 커서로 선언한다.
cursor_specification	커서로 수행할 문장을 명시한다.
statement_name	준비된 문장의 이름을 명시한다. 사용될 문장은 PREPARE를 통해 준비되어 있어야 한다.

– cursor_specification

구성요소	설명
query_expression	SELECT 문장을 명시한다. INTO 절을 포함할 수 없다.

- 예제

  다음은 DECLARE CURSOR를 사용한 예이다.

```
EXEC SQL DECLARE EMP-CURSOR1 CURSOR FOR
     SELECT EMPNO, ENAME, SALARY FROM EMP
END-EXEC.

EXEC SQL DECLARE EMP-CURSOR2 SCROLL CURSOR FOR
     SELECT EMPNO, ENAME, SALARY FROM EMP
END-EXEC.

EXEC SQL DECLARE DYN-CUR CURSOR FOR DYN-STMT
END-EXEC.
```

5.4.7. DECLARE DATABASE

DECLARE DATABASE는 데이터베이스를 선언할 때 사용하는 문장이다.

여러 개의 데이터베이스 접속을 사용할 때 각각의 접속은 데이터베이스의 이름으로 관리된다. DECLARE DATABASE로 선언한 데이터베이스의 이름을 CONNECT 문장이나 COMMIT 문장의 RELEASE를 통해 각각을 구분하여 관리할 수 있다.

DECLARE DATABASE의 세부 내용은 다음과 같다.

- 문법

declare_database_statement

- 구성요소

구성요소	설명
database_name	선언할 데이터베이스 이름을 명시한다.

- 예제

다음은 DECLARE DATABASE를 사용한 예이다.

```
EXEC SQL DECLARE D1 DATABASE END-EXEC.

EXEC SQL CONNECT :USERPASS AT D1 END-EXEC.
```

5.4.8. DELETE

DELETE는 테이블 또는 뷰에서 로우를 삭제할 때 사용하는 문장이다.

DELETE의 세부 내용은 다음과 같다.

- 문법

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

- 특권

  DELETE를 사용하려면, 대상이 되는 테이블 또는 뷰에 대한 DELETE 객체 특권을 갖고 있거나 DELETE ANY TABLE 시스템 특권을 갖고 있어야 한다.

- 구성요소

구성요소	설명
FOR	입력 배열 변수와 함께 사용될 경우 FOR 절을 이용하여 DELETE를 실행할 입력 배열 변수의 크기를 정할 수 있다. FOR 절이 포함되어 있지 않거나 실행할 배열의 크기가 입력 배열 변수의 크기보다 크면, 배열 전체에 대하여 DELETE를 실행한다.
CURRENT OF	DELETE를 커서와 함께 사용할 수도 있다. 현재 커서가 가리키는 로우를 삭제하려면 WHERE 절에 CURRENT OF와 커서 이름을 포함시킨다. 이때 커서는 닫혀 있지 않아야 한다.

- 예제

다음은 DELETE를 사용하는 예이다.

```
EXEC SQL DELETE FROM EMP END-EXEC.

EXEC SQL DELETE FROM EMP
     WHERE EMPNO = :EMPNO
END-EXEC.

EXEC SQL FOR :CNT DELETE FROM EMP
     WHERE EMPNO = :EMPNO
END-EXEC.

EXEC SQL DELETE FROM EMP
     WHERE CURRENT OF EMP-CURSOR
END-EXEC.
```

위의 예에서 알 수 있듯이 DELETE 문장은 EXEC SQL로 시작하고 END-EXEC.로 끝난다는 것을 제외하면, 일반적인 SQL 문장의 문법과 크게 다르지 않다.

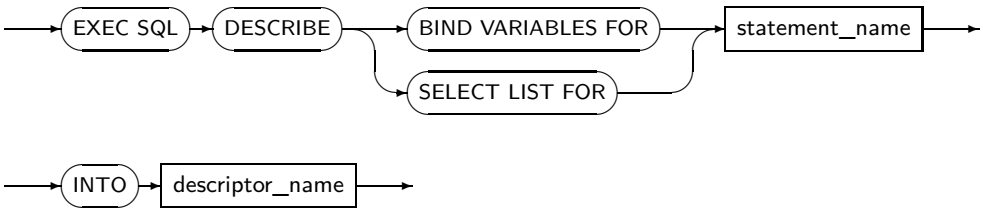
5.4.9. DESCRIBE

DESCRIBE는 호스트 변수의 서술자를 초기화할 때 사용하는 문장이다. 이 문장에는 Dynamic SQL 문장이 사용되는데, 사용될 문장은 미리 PREPARE를 통해 준비해야 한다.

DESCRIBE의 세부 내용은 다음과 같다.

- 문법

describe_statement



- 구성요소

구성요소	설명
BIND VARIABLES FOR	입력으로 사용되는 호스트 변수의 서술자를 초기화하고, 바인드 변수를 바인딩하기 위해 사용한다. 이 부분을 명시하지 않으면 SELECT LIST FOR가 디폴트이다.
SELECT LIST FOR	SELECT 문장의 SELECT 리스트의 정보를 위한 서술자를 초기화한다. 명시하지 않을 경우 디폴트이다.

구성요소	설명
statement_name	사용될 문장의 이름을 명시한다. 사용될 문장의 이름은 이미 PREPARE를 통해 준비되어 있어야 한다.
descriptor_name	서술자의 이름이 저장된 호스트 변수 또는 문자열을 적는다. 호스트 변수는 PIC X(n) 타입 등 문자열을 저장할 수 있는 타입으로 이미 선언되어 있어야 한다. 서술자는 ALLOCATE DESCRIPTOR를 통해 이미 할당되어 있어야 한다. 자세한 내용은 “5.3.3. DESCRIPTOR 이름”을 참고한다.

- 예제

  다음은 DESCRIBE를 사용하는 예이다.

```
EXEC SQL
     PREPARE PSTMT FROM :QUERY-STR
END-EXEC.

EXEC SQL DECLARE EMP-CUR FOR
     SELECT EMPNO, ENAME, SAL, COMM FROM EMP
     WHERE DEPTNO = :DEPT-NO
END-EXEC.

EXEC SQL DESCRIBE BIND VARIABLES FOR PSTMT
     INTO BIND-DESC
END-EXEC.

EXEC SQL OPEN EMP-CUR USING BIND-DESC END-EXEC.

EXEC SQL DESCRIBE SELECT LIST FOR PSTMT
     INTO SELECT-DESC
END-EXEC.

EXEC SQL FETCH EMP-CUR
     INTO SELECT-DESC
END-EXEC.
```

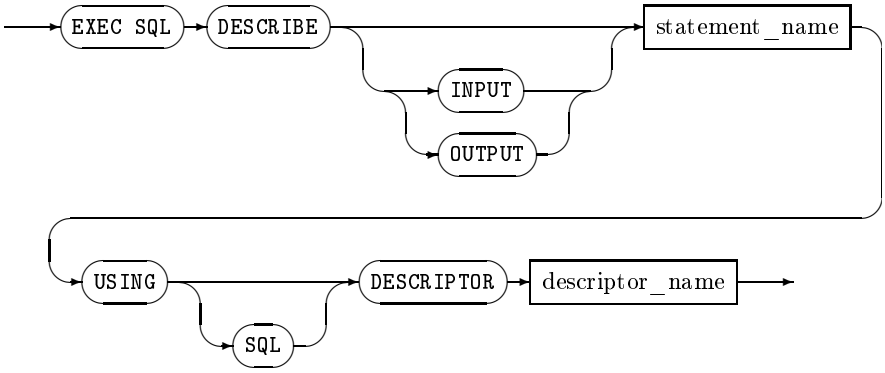
5.4.10. DESCRIBE DESCRIPTOR

DESCRIBE DESCRIPTOR는 서술자에 호스트 변수의 정보를 저장할 때 사용하는 문장이다. 단, 이 문장은 ANSI 타입의 Dynamic SQL 문장에만 사용할 수 있다.

DESCRIBE DESCRIPTOR의 세부 내용은 다음과 같다.

- 문법

ansi_describe_statement



- 구성요소

구성요소	설명
INPUT	서술자가 입력과 출력 중에 어느 곳에 사용될지 지정한다. 생략이 가능하며 기본값은 INPUT이다. 사용자가 바인드 변수에 값을 직접 입력하여, tbESQL/COBOL 라이브러리에서 사용될 수 있도록 한다.
OUTPUT	서술자가 입력과 출력 중에 어느 곳에 사용될지 지정한다. 생략이 가능하며 기본값은 INPUT이다. 사용자가 바인드 변수의 멤버를 통해, tbESQL/COBOL 라이브러리가 동작하면서 저장된 결과 값의 메타데이터(metadata)를 확인하고, 적합한 대응을 할 수 있도록 해준다.
statement_name	사용될 문장의 이름을 명시한다. 사용될 문장의 이름은 이미 PREPARE를 통해 준비되어 있어야 한다.
SQL	기존 ESQL 프로그램과의 호환성을 위한 키워드이며 특별한 의미는 없다.
descriptor_name	서술자의 이름이 저장된 호스트 변수 또는 문자열을 적는다. 호스트 변수는 PIC X(n) 타입 등 문자열을 저장할 수 있는 타입으로 이미 선언되어 있어야 한다. 서술자는 ALLOCATE DESCRIPTOR를 통해 이미 할당되어 있어야 한다. 자세한 내용은 “5.3.3. DESCRIPTOR 이름”을 참고한다.

- 예제

다음은 DESCRIBE DESCRIPTOR를 사용하는 예이다.

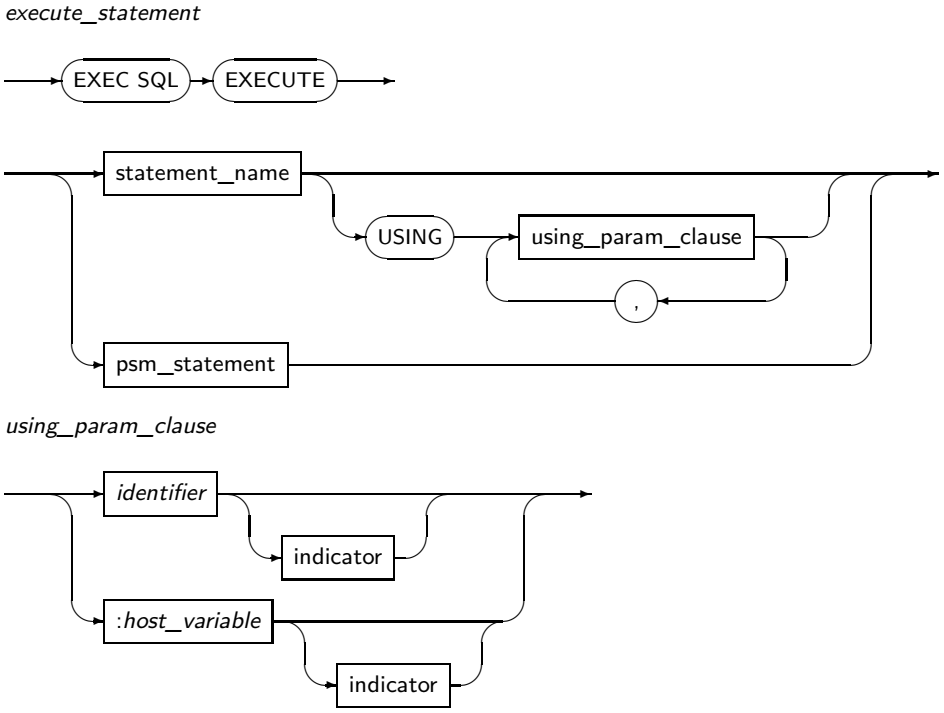
```
EXEC SQL DESCRIBE INPUT S USING DESCRIPTOR 'IN-DESC' END-EXEC.

EXEC SQL DESCRIBE OUTPUT S USING DESCRIPTOR 'OUT-DESC' END-EXEC.
```

5.4.11. EXECUTE

EXECUTE는 Dynamic SQL 문장을 실행할 때 사용하는 문장이다.

EXECUTE의 세부 내용은 다음과 같다.

- 문법



- 구성요소

  – execute_statement

구성요소	설명
statement_name	실행할 문장의 이름을 명시한다. 실행할 문장의 이름은 PREPARE 문장을 통해 이미 준비되어 있어야 한다.
USING using_param_clause	USING 절을 이용하여 입력 변수를 지정한다. 또는 입력 배열 변수를 사용할 수도 있다.
psm_statement	실행할 anonymous psm block 문장의 내용을 서술한다.

  – using_param_clause

구성요소	설명
identifier	입력 변수를 정의한 식별자이다.
:host_variable	입력 변수를 명시한다. 입력 변수의 개수는 하나 이상이고, 준비된 문장 내에 포함된 입력 변수의 개수와 같아야 한다.
indicator	지시자 변수를 명시할 때 사용한다.

- 예제

  다음은 EXECUTE를 사용하는 예이다.

```
EXEC SQL EXECUTE SQL-STMT END-EXEC.

EXEC SQL EXECUTE SQL-STMT USING :EMPNO, :DEPTNO END-EXEC.

EXEC SQL FOR :CNT EXECUTE SQL-STMT
     USING :EMPNO INDICATOR :EMPNO-IND
END-EXEC.
```

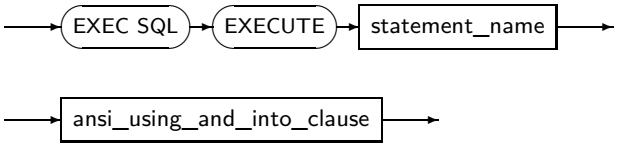
5.4.12. EXECUTE DESCRIPTOR

EXECUTE DESCRIPTOR는 Dynamic SQL 문장을 실행할 때 사용하는 문장이다. 단, 이 문장은 ANSI 타입의 Dynamic SQL 문장에만 사용할 수 있다.

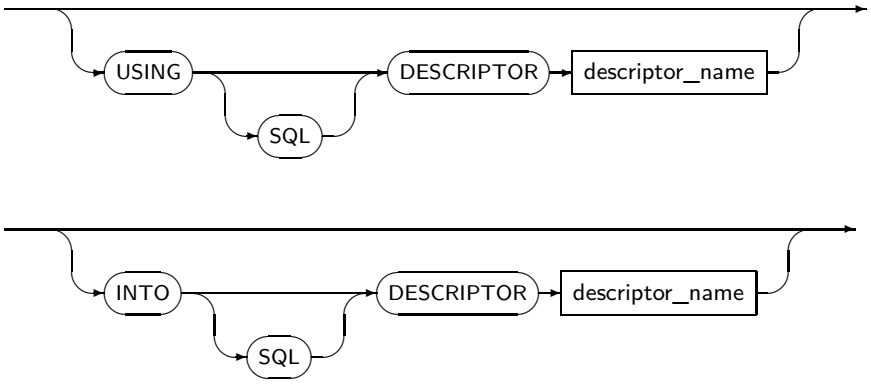
EXECUTE DESCRIPTOR의 세부 내용은 다음과 같다.

- 문법

ansi_execute_statement



ansi_using_and_into_clause



- 구성요소
  - ansi_excute_statement

구성요소	설명
statement_name	서술자의 이름이 저장된 호스트 변수 또는 문자열을 명시한다. 호스트 변수는 PIC X(n) 타입 등 문자열을 저장할 수 있는 타입으로 서술자 이름을 문자열로 가지고 있어야 한다.

  - ansi_using_and_into_clause

구성요소	설명
USING	입력 호스트 변수의 정보를 가지고 있는 지시자를 사용해야 할 경우 명시한다.
INTO	문장 수행 결과가 출력 값을 가지고 있을 경우 사용할 지시자를 명시한다.
SQL	기존 ESQL 프로그램과의 호환성을 위한 키워드이며 특별한 의미는 없다.
descriptor_name	서술자의 이름이 저장된 호스트 변수 또는 문자열을 적는다. 호스트 변수는 PIC X(n) 타입 등 문자열을 저장할 수 있는 타입으로 이미 선언되어 있어야 한다. 서술자는 ALLOCATE DESCRIPTOR를 통해 이미 할당되어 있어야 한다. 자세한 내용은 “5.3.3. DESCRIPTOR 이름”을 참고한다.

- 예제

  다음은 EXECUTE DESCRIPTOR를 사용하는 예이다.

```
EXEC SQL EXECUTE S USING DESCRIPTOR 'input_desc'
         INTO DESCRIPTOR 'output_desc'
END-EXEC.
```

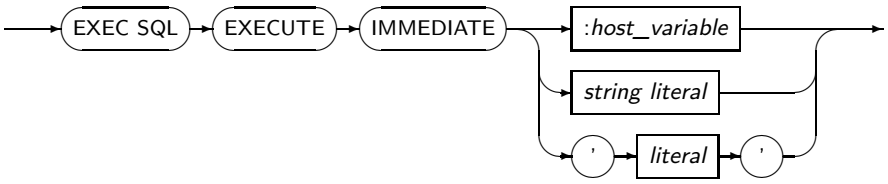
5.4.13. EXECUTE IMMEDIATE

EXECUTE IMMEDIATE는 Dynamic SQL 문장을 준비하지 않고 바로 실행할 때 사용하는 문장이다. 이 문장을 통해 실행할 SQL 문장은 입력 변수가 포함되지 않아야 한다.

EXECUTE IMMEDIATE의 세부 내용은 다음과 같다.

- 문법

execute_immediate_statement



- 구성요소

구성요소	설명
:host_variable	실행할 Dynamic SQL 문장을 호스트 변수를 사용해 명시한다.
string literal	실행할 문장을 문자열 리터럴을 사용해 명시한다.
'literal'	실행할 문장을 문자열을 사용해 명시한다.

- 예제

다음은 EXECUTE IMMEDIATE를 사용하는 예이다.

```
EXEC SQL EXECUTE IMMEDIATE :SQL-STMT
END-EXEC.

EXEC SQL EXECUTE IMMEDIATE
     Z'SELECT EMPNO, ENAME, SALARY FROM EMP'
END-EXEC.
```

5.4.14. FETCH

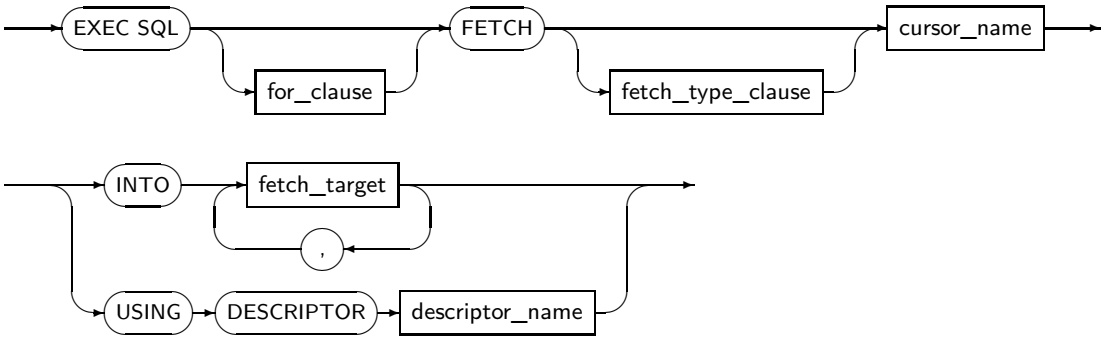
FETCH는 커서가 현재 가리키고 있는 로우의 데이터를 읽어 올 때 사용하는 문장이다.

이 문장을 사용할 때 출력 변수로 배열 변수를 사용할 수 있다. 배열 변수를 사용할 경우 여러 개의 로우 데이터를 동시에 읽어 올 수 있다.

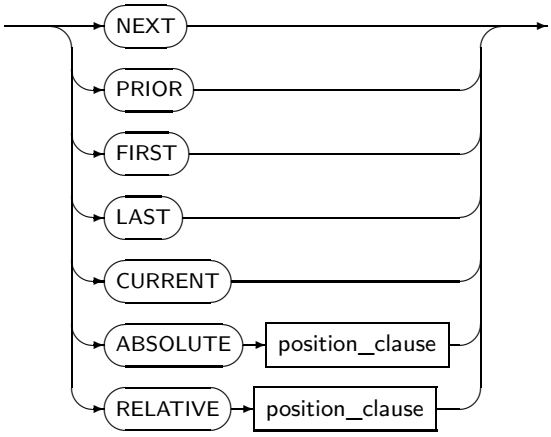
FETCH의 세부 내용은 다음과 같다.

- 문법

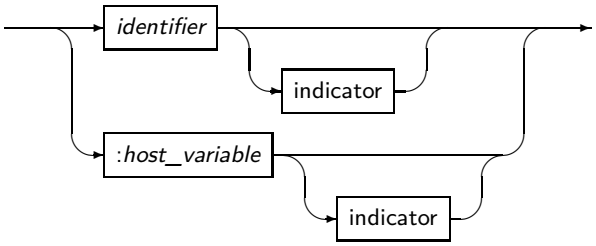
fetch_statement



fetch_type_clause



fetch_target



- 구성요소

  – fetch_statement

구성요소	설명
for_clause	입력 배열 변수를 사용할 때 for_clause를 사용해 동시에 읽을 로우의 개수를 지정할 수 있다. for_clause가 포함되어 있지 않거나, for_clause에 지정된 로우의 개수가 입력 배열 변수의 크기보다 크면, 배열 변수의 크기만큼 로우를 읽는다. 자세한 내용은 "5.3.2. FOR 절"을 참고한다.
fetch_type_clause	스크롤 커서일 경우 스크롤 타입을 명시한다.
cursor_name	커서의 이름을 명시한다. 사용될 커서는 열려있어야 한다.

구성요소	설명
fetch_target	결과 값을 저장할 호스트 변수를 명시한다.
USING DESCRIPTOR	서술자의 이름을 명시할 경우 서술자의 이름 앞에 붙이는 키워드이다.
descriptor_name	서술자의 이름이 저장된 호스트 변수 또는 문자열을 적는다. 호스트 변수는 PIC X(n) 타입 등 문자열을 저장할 수 있는 타입으로 이미 선언되어 있어야 한다. 서술자는 ALLOCATE DESCRIPTOR를 통해 이미 할당되어 있어야 한다. 자세한 내용은 “5.3.3. DESCRIPTOR 이름”을 참고한다.

– fetch_type_clause

구성요소	설명
NEXT	현재 커서가 가리키고 있는 로우의 다음 로우에 액세스를 한다. PRIOR 옵션과 반대이다. (생략 가능)
PRIOR	현재 커서가 가리키고 있는 로우의 이전 로우에 액세스를 한다. NEXT 옵션과 반대이다.
FIRST	맨 처음에 위치한 로우에 액세스를 한다. LAST 옵션과 반대이다.
LAST	맨 마지막에 위치한 로우에 액세스를 한다. FIRST 옵션과 반대이다.
CURRENT	현재 로우에 액세스를 한다.
ABSOLUTE position_clause	전체 로우 중에서 position_clause번째 로우에 액세스를 한다.
RELATIVE position_clause	현재 커서가 가리키고 있는 로우의 다음 position_clause번째에 위치한 로우에 액세스를 한다. position_clause의 값이 음수라면 커서가 현재 위치에서 앞으로 이동한다. 예를 들어 현재 커서가 8번째 로우를 가리키고 있는데, 'FETCH RELATIVE -3'을 실행한다면 커서는 5번째 로우를 가리키게 된다.

– fetch_target

구성요소	설명
identifier	출력 변수를 정의한 식별자이다.
:host_variable	출력 변수를 명시한다. 출력 변수의 개수는 하나 이상이고, 준비된 문장의 결과로 출력될 SELECT 리스트의 컬럼 개수와 동일해야 한다.
indicator	지시자 변수를 명시할 때 사용한다.

- 예제

  다음은 FETCH를 사용하는 예이다.

```
EXEC SQL FETCH EMP-CURSOR
     INTO :EMPNO, :ENAME
END-EXEC.

EXEC SQL FETCH EMP-CURSOR
     INTO :EMPNO, :ENAME:ENAME-IND
END-EXEC.

EXEC SQL FOR :CNT FETCH EMP-CURSOR
     INTO :EMPNO-ARRAY, :ENAME-ARRAY
END-EXEC.
```

5.4.15. FETCH DESCRIPTOR

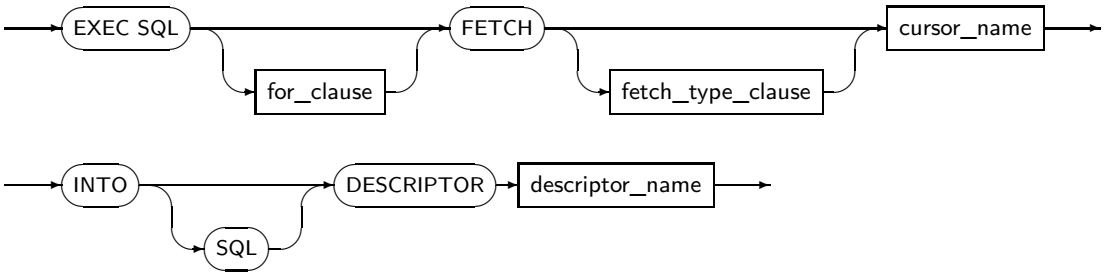
FETCH DESCRIPTOR는 커서가 현재 가리키고 있는 로우 데이터를 읽어 올 때 사용하는 문장이다.

FETCH와 거의 동일하게 동작한다. 단, 이 문장은 ANSI 타입의 Dynamic SQL 문장에만 사용할 수 있다.

FETCH DESCRIPTOR의 세부 내용은 다음과 같다.

- 문법

ansi_fetch_statement



- 구성요소

구성요소	설명
for_clause	입력 배열 변수를 사용할 때 for_clause를 사용해 동시에 읽을 로우의 개수를 지정할 수 있다. for_clause가 포함되어 있지 않거나, for_clause에 지정된 로우의 개수가 입력 배열 변수의 크기보다 크면, 배열 변수의 크기만큼 로우를 읽는다. 자세한 내용은 "5.3.2. FOR 절"을 참고한다.
fetch_type_clause	FETCH의 fetch_type_clause와 동일하다.
cursor_name	커서의 이름을 명시한다. 사용될 커서는 열려 있는 커서이어야 한다.
SQL	기존 ESQL 프로그램과의 호환성을 위한 키워드이며 특별한 의미는 없다.

구성요소	설명
descriptor_name	서술자의 이름이 저장된 호스트 변수 또는 문자열을 적는다. 호스트 변수는 PIC X(n) 타입 등 문자열을 저장할 수 있는 타입으로 이미 선언되어 있어야 한다. 서술자는 ALLOCATE DESCRIPTOR를 통해 이미 할당되어 있어야 한다. 자세한 내용은 “5.3.3. DESCRIPTOR 이름”을 참고한다.

- 예제

  다음은 FETCH DESCRIPTOR를 사용하는 예이다.

```
EXEC SQL FETCH C1 INTO DESCRIPTOR 'OUT-DESC' END-EXEC.
```

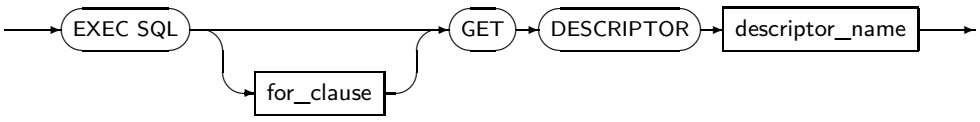
5.4.16. GET DESCRIPTOR

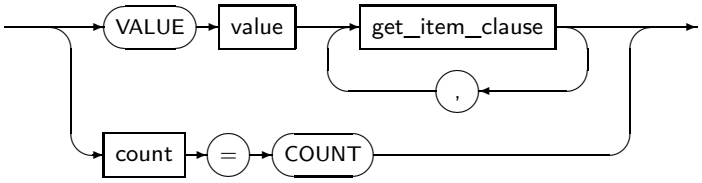
GET DESCRIPTOR는 지정한 서술자에서 원하는 정보를 가져올 때 사용하는 문장이다. 단, 이 문장은 ANSI 타입의 Dynamic SQL 문장에만 사용할 수 있다.

GET DESCRIPTOR의 세부 내용은 다음과 같다.

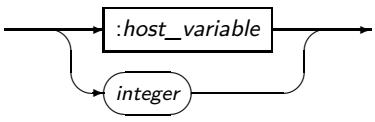
- 문법

get_descriptor_statement

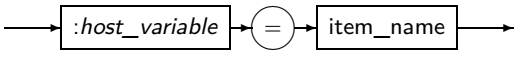




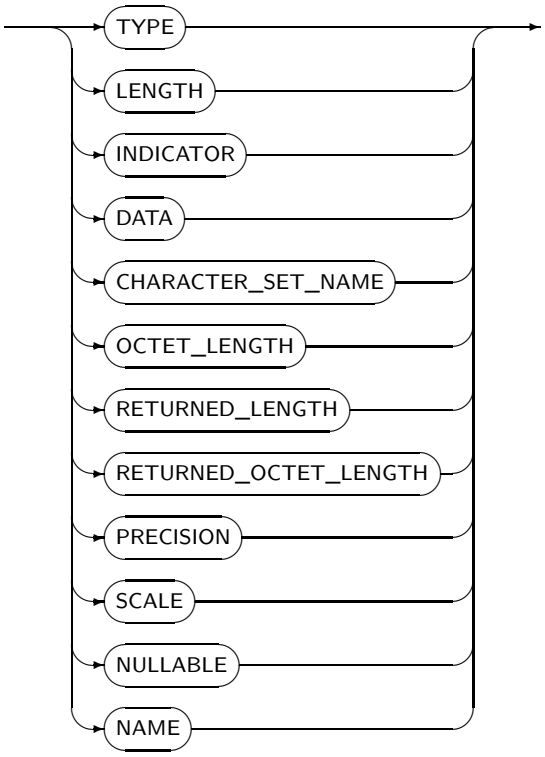
value



get_item_clause



item_name



- 구성요소

  – get_descriptor_statement

구성요소	설명
for_clause	입력 배열 변수를 사용할 때 for_clause를 사용해 동시에 읽을 로우의 개수를 지정할 수 있다. for_clause가 포함되어 있지 않거나, for_clause에 지정된 로우의 개수가 입력 배열 변수의 크기보다 크면, 배열 변수의 크기만큼 로우를 읽는다. 자세한 내용은 “5.3.2. FOR 절”을 참고한다.
descriptor_name	서술자의 이름이 저장된 호스트 변수 또는 문자열을 적는다. 호스트 변수는 PIC X(n) 타입 등 문자열을 저장할 수 있는 타입으로 이미 선언되어 있어야 한다. 서술자는 ALLOCATE DESCRIPTOR를 통해 이미 할당되어 있어야 한다. 자세한 내용은 “5.3.3. DESCRIPTOR 이름”을 참고한다.
VALUE value	정보를 가져올 호스트 변수의 순서를 지정한다.
get_item_clause	가져올 정보의 항목과 정보를 저장할 호스트 변수를 명시한다.
count	사용된 호스트 변수의 개수를 알고자 할 경우 그 개수를 저장할 호스트 변수를 명시한다.

– value

구성요소	설명
:host_variable	값이 들어 있는 호스트 변수를 사용할 때 명시한다.
integer	값을 직접 정수로 입력할 때 사용한다.

– get_item_clause

구성요소	설명
:host_variable	해당 항목을 가져올 호스트 변수를 명시한다.
item_name	값을 가져올 구체적인 항목을 명시한다.

– item_name

구성요소	설명
TYPE	데이터의 타입을 가져오고자 할 때 사용한다.
LENGTH	데이터의 최대 길이를 가져오고자 할 때 사용한다.
INDICATOR	데이터 연관된 지시자 값을 가져오고자 할 때 사용한다.
DATA	데이터 값을 가져오고자 할 때 사용한다.
CHARACTER_SET_NAME	데이터가 저장된 컬럼의 문자 세트를 가져오고자 할 때 사용한다.
OCTET_LENGTH	데이터 길이를 Byte 단위로 환산해 가져오고자 할 때 사용한다.
RETURNED_LENGTH	FETCH를 할 때 실제로 받아올 데이터 길이를 가져오고자 할 때 사용한다.
RETURNED_OCTET_LENGTH	FETCH를 할 때 실제로 받아올 데이터 길이를 Byte 단위로 환산해 가져오고자 할 때 사용한다.
PRECISION	받아올 데이터의 정밀도를 가져오고자 할 때 사용한다.
SCALE	받아올 데이터의 스케일을 가져오고자 할 때 사용한다.
NULLABLE	해당 컬럼의 데이터가 NULL이 될 수 있는지 여부를 알고자 할 때 사용한다. – 값이 1이면, 해당 컬럼은 NULL값을 가질 수 있다. – 값이 0이면, 해당 컬럼은 NULL값을 가질 수 없는 키이거나 NOT-NULL 제약조건을 가지고 있는 컬럼이다.
NAME	해당 컬럼의 이름을 가져오고자 할 때 사용한다.

- 예제

  다음은 GET DESCRIPTOR를 사용하는 예이다.

```
EXEC SQL GET DESCRIPTOR 'OUT-DESC' :INPUT-CNT = COUNT END-EXEC.
```

5.4.17. INSERT

INSERT는 테이블 또는 뷰에 로우를 삽입할 때 사용하는 문장이다. 전체 컬럼 또는 일부 컬럼에 데이터를 삽입할 수 있다. INSERT를 사용할 때 입력 변수로 배열 변수를 사용할 수도 있으며, 지시자 변수를 함께 사용할 수 있다.

또한 입력할 데이터의 값을 사용자가 직접 지정할 수도 있고, 부질의를 통하여 지정할 수도 있다. 부질의를 이용하는 경우 부질의의 결과 로우 모두가 테이블이나 뷰에 삽입된다.

INSERT의 세부 내용은 다음과 같다.

- 문법

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

- 특권

  INSERT를 사용하려면, INSERT의 대상 테이블 또는 뷰에 대하여 INSERT 객체 특권을 갖거나 INSERT ANY TABLE 시스템 특권을 갖고 있어야 한다.

- 구성요소

구성요소	설명
FOR	입력 배열 변수와 함께 사용될 경우 FOR 절을 이용하여 삽입할 데이터의 개수를 지정할 수 있다. FOR 절이 포함되어 있지 않거나 FOR 절에서 지정한 크기가 배열 입력 변수의 크기보다 크면, 전체 배열 변수에 저장된 데이터가 테이블에 삽입된다.

- 예제

  다음은 INSERT를 사용하는 예이다.

```
EXEC SQL INSERT INTO EMP
         VALUES (34, ‘James', 45000)
END-EXEC.

EXEC SQL INSERT INTO EMP (EMPNO, ENAME)
         VALUES (:EMPNO, :ENAME)
END-EXEC.

EXEC SQL FOR :CNT
     INSERT INTO EMP (EMPNO, ENAME)
     VALUES (:EMPNO-ARRAY,:ENAME-ARRAY :ENAME-IND-ARRAY)
END-EXEC.

EXEC SQL INSERT INTO HIGH_EMP
```

```
     SELECT * FROM EMP WHERE SALARY > 50000
END-EXEC.
```

위의 예에서 알 수 있듯이 INSERT 문장은 EXEC SQL로 시작하고 END-EXEC.로 끝난다는 것을 제외하면, 일반적인 SQL 문장의 문법과 크게 다르지 않다.

5.4.18. OPEN

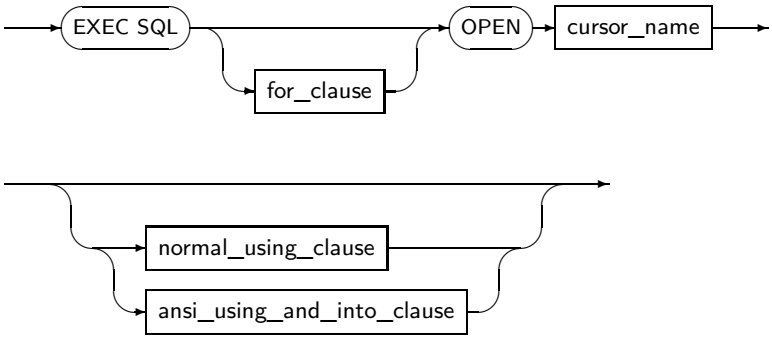
OPEN은 커서를 열 때 사용하는 문장이다. 이 문장을 사용해 열려고 하는 커서는 이미 선언되어 있어야 하며, SELECT 문장과 연관되어 있어야 한다. 커서의 선언은 DECLARE CURSOR를 통하여 실행된다.

커서를 여는 것과 동시에 연관된 SELECT 문장이 실행되며, 커서는 질의 문장이 반환한 결과의 제일 처음에 위치한 로우를 가리킨다. 커서는 SELECT 문장과 연관되기도 하지만, Dynamic SQL 문장과 연관되기도 한다.

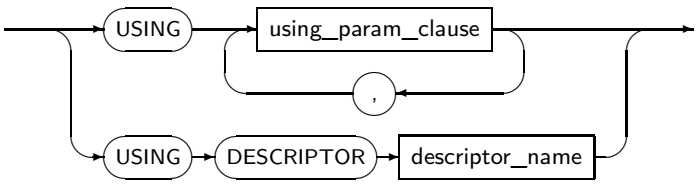
OPEN의 세부 내용은 다음과 같다.

- 문법

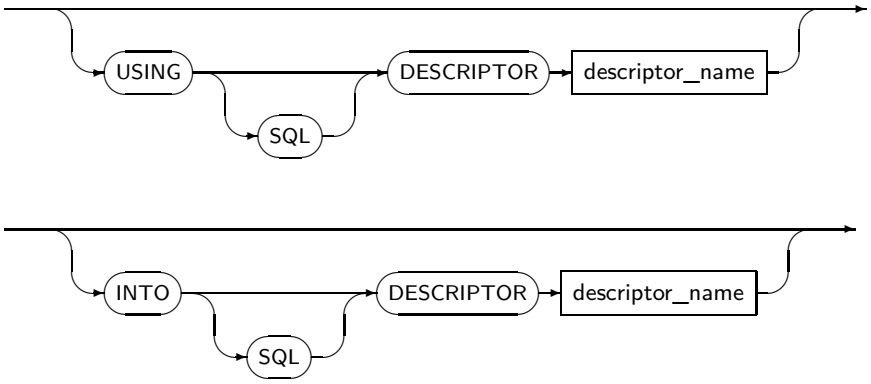
open_statement



normal_using_clause



ansi_using_and_into_clause



- 구성요소

  – open_statement

구성요소	설명
for_clause	실행 반복 횟수를 지정한다. 자세한 내용은 "5.3.2. FOR 절"을 참고한다.
cursor_name	열려고 하는 커서의 이름을 명시한다.
normal_using_clause	일반적으로 사용하는 OPEN 문장의 형태이다.
ansi_using_and_into_clause	이미 정의된 서술자를 이용한다. ANSI 타입의 Dynamic SQL 문장에만 사용된다.

  – normal_using_clause

구성요소	설명
USING	커서가 Dynamic SQL 문장과 연관된 경우 USING 절을 이용하여 SELECT 문장의 입력 변수에 할당할 값을 지정할 수 있다. 입력 변수로 배열 변수를 사용할 수도 있으며, 서술자 변수를 함께 사용할 수도 있다.
using_param_clause	입력 호스트 변수의 정보가 필요할 경우에 사용한다.
USING DESCRIPTOR	입력 호스트 변수의 정보를 가져왔던 서술자를 사용할 경우 명시한다.
descriptor_name	서술자의 이름이 저장된 호스트 변수 또는 문자열을 적는다. 호스트 변수는 PIC X(n) 타입 등 문자열을 저장할 수 있는 타입으로 이미 선언되어 있어야 한다. 서술자는 ALLOCATE DESCRIPTOR를 통해 이미 할당되어 있어야 한다. 자세한 내용은 "5.3.3. DESCRIPTOR 이름"을 참고한다.

  – ansi_using_and_into_clause

구성요소	설명
USING	입력 호스트 변수의 정보가 들어 있는 서술자 변수가 필요할 경우에 사용한다.
INTO	출력 호스트 변수의 정보가 들어 있는 서술자 변수가 필요할 경우에 사용한다.
SQL	기존 ESQL 프로그램과의 호환성을 위한 키워드이며 특별한 의미는 없다.
descriptor_name	서술자의 이름이 저장된 호스트 변수 또는 문자열을 적는다. 호스트 변수는 PIC X(n) 타입 등 문자열을 저장할 수 있는 타입으로 이미 선언되어 있어야 한다. 서술자는 ALLOCATE DESCRIPTOR를 통해 이미 할당되어 있어야 한다. 자세한 내용은 “5.3.3. DESCRIPTOR 이름”을 참고한다.

- 다음은 OPEN을 사용하는 예이다.

```
EXEC SQL OPEN EMP-CURSOR END-EXEC.

EXEC SQL OPEN EMP-CURSOR USING :EMPNO END-EXEC.

EXEC SQL FOR :CNT
     OPEN EMP-CURSOR USING :EMPNO INDICATOR :EMPNO-IND
END-EXEC.
```

5.4.19. PREPARE

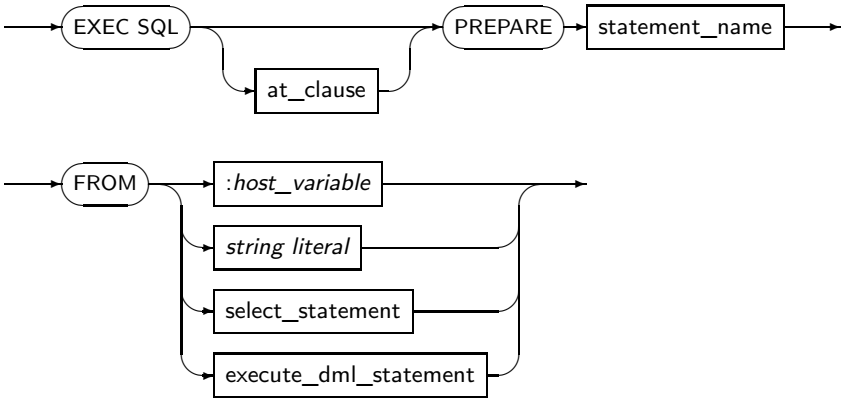
PREPARE는 Dynamic SQL 문장을 준비할 때 사용하는 문장이다. 준비된 SQL 문장은 문장의 이름을 통해 서로 식별되며, 이후에 DECLARE CURSOR 또는 EXECUTE에서 참조된다.

SQL 문장을 준비한다는 것은 단지 문장을 파싱(Parsing)한다는 의미일 뿐이며, 문장이 실행된다는 의미는 아니다. 문장이 실제로 실행되는 것은 EXECUTE 문장을 통해서이다. SQL 문장에는 하나 이상의 입력 변수가 포함될 수 있는데, 입력 변수가 포함될 때 입력 변수의 이름은 별다른 의미를 갖지 않으며, tbESQL 프로그램에 미리 선언되어 있을 필요도 없다.

PREPARE의 세부 내용은 다음과 같다.

- 문법

prepare_statement



- 구성요소

구성요소	설명
statement_name	준비를 실행할 SQL 문장의 이름을 명시한다.
FROM	FROM 절 뒤에는 준비할 Dynamic SQL 문장이 온다. SQL 문장 자체가 올 수도 있으며, SQL 문장의 문자열을 저장한 호스트 변수가 올 수도 있다. SELECT 문장이라면 문장 자체가 올 수 있다.
:host_variable	Dynamic SQL 문장을 저장한 호스트 변수를 명시한다.
string literal	Dynamic SQL 문장을 나타내는 문자열을 명시한다.
select_statement	SELECT 문장 자체를 명시한다.
execute_dml_statement	실행할 INSERT, UPDATE, DELETE 문장 자체를 명시한다.

- 예제

다음은 PREPARE를 사용하는 예이다.

```
EXEC SQL PREPARE EMP-STMT FROM :SQL-STMT END-EXEC.

EXEC SQL PREPARE EMP-STMT
      FROM Z'UPDATE EMP SET SALARY = SALARY * 1.05'
END-EXEC.

EXEC SQL PREPARE EMP-STMT FROM
     SELECT EMPNO, ENAME, SALARY FROM EMP WHERE DEPTNO = :DEPTNO
END-EXEC.
```

5.4.20. ROLLBACK

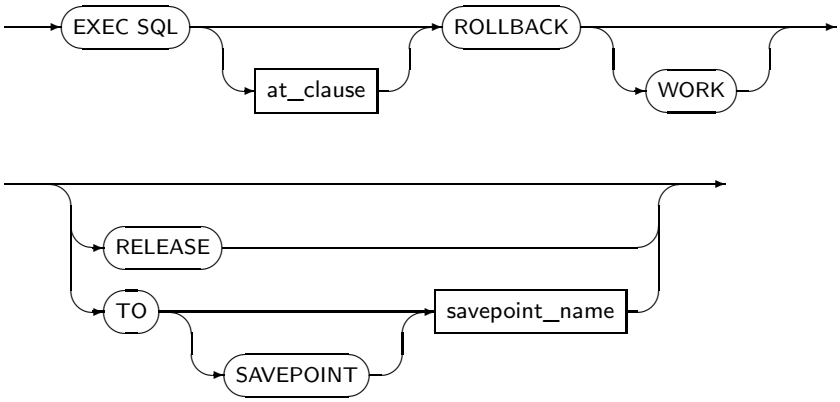
ROLLBACK은 현재 진행 중인 트랜잭션을 롤백하고 갱신된 모든 내용을 취소할 때 사용하는 문장이다.

롤백을 실행하는 것과 동시에 데이터베이스와의 접속을 끊을 수도 있으며, 미리 설정된 저장점까지 부분 롤백(Partial Rollback)을 수행할 수도 있다.

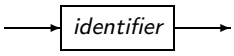
ROLLBACK의 세부 내용은 다음과 같다.

- 문법

rollback_statement



savepoint_name



- 구성요소

  – rollback_statement

구성요소	설명
at_clause	문장을 실행할 데이터베이스를 명시한다. 데이터베이스의 이름을 직접 명시할 수도 있고, 데이터베이스의 이름이 저장된 호스트 변수를 명시할 수도 있다. 자세한 내용은 “5.3.1. AT 절”을 참고한다.
WORK	기존 ESQL 프로그램과의 호환성을 위한 키워드이며 특별한 의미는 없다.
RELEASE	모든 리소스를 반환하고 데이터베이스와의 접속을 종료한다.
TO	특정 저장점까지 부분 롤백을 수행하고자 할 때 명시한다.
SAVEPOINT	단지 문법 호환을 위해 지원하는 부분이다. 이 부분의 명시여부는 문장 실행에 어떤 영향도 없다.
savepoint_name	부분 롤백을 수행할 저장점의 이름을 명시한다. 저장점은 롤백을 실행하기 전에 프로그램 내에 이미 설정되어 있어야 한다.

  – savepoint_name

구성요소	설명
identifier	저장점의 이름을 정의한 식별자이다.

- 예제

  다음은 ROLLBACK을 사용하는 예이다.

```
EXEC SQL ROLLBACK END-EXEC.

EXEC SQL ROLLBACK TO SAVEPOINT SP1 END-EXEC.

EXEC SQL ROLLBACK WORK RELEASE END-EXEC.
```

5.4.21. SAVEPOINT

SAVEPOINT는 저장점을 설정할 때 사용하는 문장이다. 설정된 저장점은 특정 지점까지 부분 롤백을 하고자 할 때 사용된다.

SAVEPOINT의 세부 내용은 다음과 같다.

- 문법

  savepoint_statement

- 구성요소

구성요소	설명
savepoint_name	설정할 저장점에 부여할 이름을 명시한다.

- 예제

  다음은 SAVEPOINT를 사용하는 예이다.

```
EXEC SQL SAVEPOINT SP1 END-EXEC.
```

5.4.22. SELECT

SELECT는 테이블 또는 뷰에 대해 질의를 수행하고, 질의를 수행한 결과 로우의 각 데이터를 출력 변수에 저장할 때 사용하는 문장이다.

SELECT의 세부 내용은 다음과 같다.

- 문법

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

- 특권

  SELECT 문장을 사용해 질의를 수행하기 위해서는, 대상 테이블에 대한 SELECT 객체 특권이 있거나 SELECT ANY TABLE 시스템 권한을 갖고 있어야 한다.

- 구성요소

구성요소	설명
INTO	질의 결과의 컬럼의 개수는 INTO 절에 포함된 출력 변수의 개수와 같아야 한다. INTO 절에는 지시자 변수와 함께 배열 출력 변수를 사용할 수 있다. 출력 변수는 모두 스칼라 변수이거나 또는 모두 배열 변수이어야 하며, 이 두 가지가 서로 섞여 있을 수 없다.
WHERE	WHERE 절에는 입력 변수를 포함할 수 있다. 이때 지시자 변수와 함께 사용될 수 있다. SELECT 문장 내에서는 배열 입력 변수를 사용할 수 없다.
HAVING	HAVING 절에는 입력 변수를 포함할 수 있다. 지시자 변수 및 배열 입력 변수와 관련된 내용은 WHERE 절과 동일하다.

- 예제

  다음은 SELECT를 사용하는 예이다.

```
EXEC SQL SELECT EMPNO, ENAME, SALARY
     INTO :EMPNO, :ENAME, :SALARY FROM EMP
END-EXEC.

EXEC SQL SELECT EMPNO, ENAME, DNAME
     INTO :EMPNO-ARRAY,
          :ENAME-ARRAY :ENAME-ARRAY-IND,
          :DNAME-ARRAY
      FROM EMP E, DEPT D
      WHERE E.DEPTNO = D.DEPTNO AND E.DEPTNO = :DEPTNO
END-EXEC.
```

  위의 예에서 알 수 있듯이 SELECT 문장은 INTO 절을 제외하면, 일반적인 SQL 문장의 문법과 크게 다르지 않다.

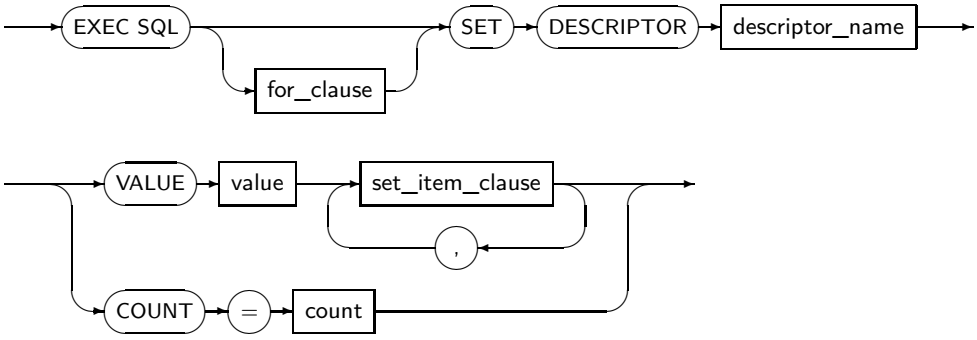
5.4.23. SET DESCRIPTOR

SET DESCRIPTOR는 사용자가 입력해야 할 정보를 지정한 서술자에 기록하는 문장이다. 사용자는 이 문장을 통해 데이터 값이나 데이터 값의 길이 등을 지정할 수 있다. 단, 이 문장은 ANSI 타입의 Dynamic SQL 문장에만 사용할 수 있다.

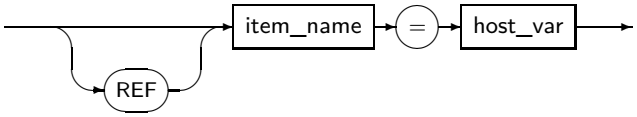
SET DESCRIPTOR의 세부 내용은 다음과 같다.

- 문법

set_descriptor_statement



set_item_clause



- 구성요소

  – set_descriptor_statement

구성요소	설명
for_clause	반복 횟수를 지정한다. 자세한 내용은 "5.3.2. FOR 절"을 참고한다.
descriptor_name	서술자의 이름이 저장된 호스트 변수 또는 문자열을 적는다. 호스트 변수는 PIC X(n) 타입 등 문자열을 저장할 수 있는 타입으로 이미 선언되어 있어야 한다. 서술자는 ALLOCATE DESCRIPTOR를 통해 이미 할당되어 있어야 한다. 자세한 내용은 "5.3.3. DESCRIPTOR 이름"을 참고한다.
VALUE value	몇 번째 출력 호스트 변수인지 명시한다.
set_item_clause	출력 호스트 변수의 정보를 입력하기 위해 서술한다.
COUNT	출력 호스트 변수의 개수를 받아오기 위해 서술한다.
count	출력 호스트 변수의 개수를 받아올 호스트 변수를 명시한다.

  – set_item_clause

구성요소	설명
REF	INDICATOR, DATA, RETURNED_LENGTH 항목을 지정할 때만 사용할 수 있는 키워드로 속도와 편리성을 위해 사용한다.

구성요소	설명
	호스트 변수의 값이 아닌 호스트 변수 자체를 지정하며, GET DESCRIPTOR 문장을 따로 실행하지 않아도 FETCH한 후 지정된 항목의 값이 지정된 호스트 변수에 들어간다.
item_name	GET DESCRIPTOR와 동일하다.
host_var	출력 호스트 변수의 각 정보를 받아올 호스트 변수를 지정한다.

- 예제

  다음은 SET DESCRIPTOR를 사용하는 예이다.

```
EXEC SQL SET DESCRIPTOR 'OUTPUT-DESCRIPTOR' VALUE :OCCURS
     TYPE = :TYPE, LENGTH = :LEN END-EXEC
```

5.4.24. UPDATE

UPDATE는 테이블 또는 뷰의 컬럼 값을 갱신할 때 사용하는 문장이다. 일부 컬럼 또는 전체 컬럼에 대해 갱신을 수행할 수 있다. 이 문장을 사용할 때 입력 변수로 배열 변수를 사용할 수도 있으며, 지시자 변수를 함께 사용할 수 있다.

UPDATE의 세부 내용은 다음과 같다.

- 문법

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

- 특권

  UPDATE를 사용해 갱신을 수행하려면, 대상 테이블 또는 뷰에 대한 UPDATE 객체 특권을 갖거나 UPDATE ANY TABLE 시스템 특권을 가지고 있어야 한다.

- 구성요소

구성요소	설명
SET	SET 절에 입력 배열 변수가 포함되어 있다면, WHERE 절에도 반드시 같은 크기의 입력 배열 변수가 포함되어야 한다.
WHERE	WHERE 절에 입력 배열 변수가 포함되어 있다면, SET 절에도 반드시 같은 크기의 입력 배열 변수가 포함되어야 한다. UPDATE 문장을 커서와 함께 사용할 수도 있다. 현재 커서가 가리키는 로우의 컬럼 값을 갱신하려면 WHERE 절에 CURRENT OF와 커서 이름을 포함시킨다. 커서를 사용하려면 커서는 이미 열려 있는 상태이어야 한다.

구성요소	설명
FOR	입력 배열 변수를 사용할 때 FOR 절을 이용하여 삽입할 데이터의 개수를 지정할 수 있다. FOR 절이 포함되어 있지 않거나 FOR 절에서 지정한 개수가 입력 배열 변수의 크기보다 크다면, 전체 배열 변수에 저장된 데이터에 대하여 갱신을 수행한다.

- 예제

  다음은 UPDATE를 사용하는 예이다.

```
EXEC SQL UPDATE EMP SET SALARY = SALARY * 1.05 END-EXEC.

EXEC SQL UPDATE EMP SET SALARY = SALARY * :RATIO
         WHERE DEPTNO = :DEPTNO
END-EXEC.

EXEC SQL FOR :CNT
     UPDATE EMP SET SALARY = SALARY * :RATIO-ARRAY
     WHERE DEPTNO = :DEPTNO-ARRAY
END-EXEC.

EXEC SQL UPDATE EMP SET SALARY = SALARY * 1.05
     WHERE CURRENT OF EMP-CURSOR
END-EXEC.
```

  위의 예에서 알 수 있듯이 UPDATE 문장은 EXEC SQL로 시작하고 END-EXEC.로 끝난다는 것을 제외하면, 일반적인 SQL 문장의 문법과 크게 다르지 않다.

5.4.25. WHENEVER

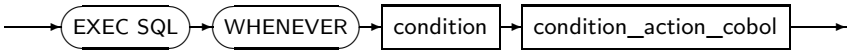
WHENEVER는 tbESQL/COBOL 프로그램을 실행하는 도중, 예외 상황이 발생했을 경우에 대비하여 수행할 작업을 선언할 때 사용하는 문장이다. 발생 가능한 예외 상황에는 액세스할 로우가 없는 경우와 에러 또는 경고가 발생한 경우가 있다.

WHENEVER 문장의 효과 범위는 WHENEVER 문장이 명시된 위치부터 다음 WHENEVER 문장이 나타날 때까지이다. 하지만 문장이 한 번만 명시되었다면 효과 범위는 WHENEVER 문장이 명시된 위치부터 프로그램의 마지막까지이다.

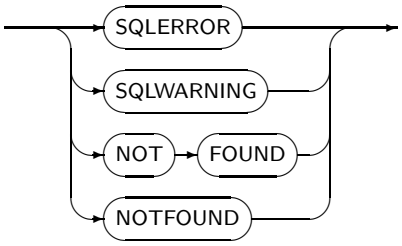
WHENEVER의 세부 내용은 다음과 같다.

- 문법

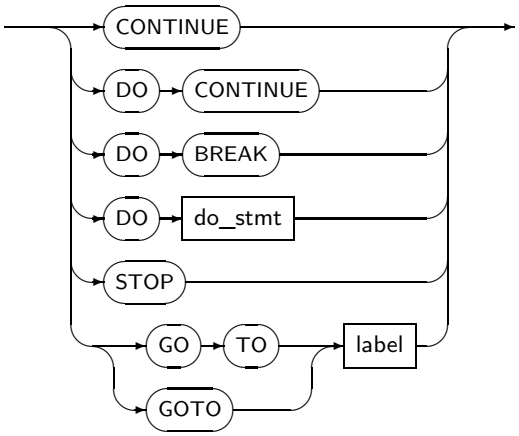
whenever_statement_cobol



condition



condition_action_cobol



- 구성요소

  – whenever_statement_cobol

구성요소	설명
condition	에러나 경고 또는 결과 로우가 없는 등의 예외 상황을 명시한다.
condition_action_cobol	예외 상황이 발생했을 경우 그에 따른 대처 방법을 명시한다.

  – condition

구성요소	설명
SQLERROR	에러가 발생한 경우이다.
SQLWARNING	경고가 발생한 경우이다.
NOT FOUND 또는 NOTFOUND	질의 결과 로우가 없거나 커서를 사용해 읽을 로우가 더 이상 없는 경우이다.

  – condition_action_cobol

구성요소	설명
CONTINUE	다음 라인부터 프로그램을 계속 진행한다.
DO CONTINUE	루프 내에서 CONTINUE를 실행한다.
DO BREAK	루프 내에서 BREAK를 실행한다.
DO do_stmt	COBOL 프로그래밍 언어로 작성한 에러 처리 문장을 수행한다.
STOP	현재 트랜잭션을 롤백하고 프로그램을 정지한다.
GOTO label	해당 label이 있는 곳으로 이동하여 프로그램을 진행한다.

- 예제

  다음은 WHENEVER를 사용하는 예이다.

```
EXEC SQL WHENEVER NOT FOUND GOTO FETCH-END END-EXEC.

EXEC SQL WHENEVER SQLERROR DO PERFORM SQLERROR END-EXEC.

EXEC SQL WHENEVER SQLWARNING CONTINUE END-EXEC.
```

제6장 tbESQL/COBOL 프리컴파일러 옵션

본 장에서는 tbESQL/COBOL 프리컴파일러를 동작시킬 때 사용할 수 있는 옵션에 대해 설명한다.

6.1. 개요

tbESQL/COBOL 프리컴파일러 옵션은 프리컴파일러를 동작시킬 때 프리컴파일러가 추가적인 기능을 수행하도록 설정하기 위해서 사용된다.

우선 먼저 프리컴파일러를 사용하려면, 다음과 같이 입력한다.

[예 6.1] tbESQL/COBOL 프리컴파일 실행

```
tbpcb test1.tbco
```

위 예를 보면 **tbpcb** 유틸리티를 통해 프리컴파일을 실행하게 된다. **'test1.tbco'**은 프리컴파일을 실행할 대상 파일의 이름이다. 프리컴파일을 실행할 때 대상 파일의 확장자가 '.tbco'일 경우 확장자를 생략하고 사용해도 무방하다. 'test1.tbco'는 tbESQL/COBOL에서 제공하는 프리컴파일러의 옵션 중 하나인 INAME에 해당한다. 모든 프리컴파일러의 옵션 중에서 옵션의 이름을 생략할 수 있는 것은 INAME 하나뿐이다.

따라서 INAME을 생략하지 않고, 다음과 같이 사용해도 위의 예와 동일한 의미를 갖는다.

```
tbpcb INAME=test1.tbco
```

다음은 INAME과 INCLUDE 옵션을 사용한 예이다.

[예 6.2] tbESQL/COBOL 프리컴파일러의 옵션을 사용한 프리컴파일 실행

```
tbpcb test1.tbco INCLUDE=/home/tibero/include
```

위의 예는 tbESQL/COBOL에서 제공하는 프리컴파일러 옵션을 두 개를 함께 사용한 것이다. 앞서 언급했듯이 'INAME='은 생략할 수 있다. INCLUDE 옵션은 'test1.tbco' 안에서 사용된 각종 COPY, ESQL INCLUDE 파일이 같은 디렉터리에 있지 않을 경우 프리컴파일을 실행할 때 에러를 발생하게 된다. 따라서 이런 파일들이 존재하는 디렉터리를 지정해줄 때 사용하는 옵션이다.

6.2. tbESQL/COBOL 프리컴파일러 옵션 지정

프리컴파일러 옵션은 명령 프롬프트에서 직접 입력할 수도 있지만, 환경설정 파일이나 프로그램 내부에서도 지정할 수 있다. 옵션의 종류에 따라 지정할 수 있는 장소가 다르다. 어떤 곳에서도 옵션을 지정하지 않았을 경우 기본값이 적용된다.

옵션이 적용되는 순서는 다음과 같다.

1. 기본값
2. 환경설정 파일
3. 명령 프롬프트
4. 프로그램 내부

하나의 항목만을 허용하는 동일한 한 가지 옵션에 여러 번에 걸쳐 다른 항목이 지정되었을 경우 항상 마지막에 지정된 내용만 유효하다. 또한 옵션을 프로그램 내부에서 지정할 경우 옵션의 영향 범위는 COBOL 프로그래밍 언어의 문법에서의 변수의 영향 범위와는 무관하다. 무조건 프로그램의 소스 코드의 진행 순서에서 가장 마지막에 지정된 옵션이 적용된다. INAME의 경우는 다른 옵션과 달리, 두 번 이상 INAME이 나타날 경우 에러가 발생한다.

6.2.1. 환경설정 파일

옵션을 지정할 때 환경설정 파일을 사용하는 방법에는 다음의 두 가지가 있다.

기본 환경설정 파일

tbESQL/COBOL에서는 기본적으로 tbpcb.cfg라는 환경설정 파일을 $TB_HOME/client/config 디렉터리에 두고 있다. 이 환경설정 파일을 수정해서 원하는 옵션을 지정할 수 있다. 만약 별도의 환경설정 파일이 지정되지 않으면 tbpcb 유틸리티는 자동으로 이 환경설정 파일을 먼저 읽는다.

다음은 tbpcb.cfg 파일을 사용하여 프리컴파일러 옵션을 지정하는 예이다.

```
#INCLUDE=$TB_HOME/demo/chb/new_src/OV
INCLUDE=$TB_HOME/client/include
DYNAMIC=ANSI
```

위의 예에서처럼 #을 이용해서 행 전체에 대해 주석 처리를 할 수 있다. 환경설정 파일에서도 하나의 항목만 허용하는 옵션이 두 번 이상 나타났을 경우 마지막에 지정된 옵션이 적용된다.

사용자 환경설정 파일

사용자가 환경설정 파일을 임의로 생성하여 사용할 수도 있다. 파일의 위치와 파일의 이름 등을 사용자가 임의로 정할 수 있다. 이 경우 명령 프롬프트를 통해 사용할 환경설정 파일을 지정한다. 이렇게 환경설정 파일을 지정하면, tbpcb.cfg 파일은 사용되지 않는다.

다음은 사용자 환경설정 파일을 지정하는 예이다.

```
tbpcb test1.tbco CONFIG=config1.cfg
```

6.2.2. 명령 프롬프트

tbESQL/COBOL 프리컴파일러 옵션은 명령 프롬프트에서 지정할 수도 있다.

명령 프롬프트에서는 다음과 같은 형태로 옵션을 지정할 수 있다.

```
[OPTION_NAME=value]
```

다음은 명령 프롬프트에서 프리컴파일러 옵션을 지정하는 예이다.

```
tbpcb test1.tbco SELECT_ERROR=NO
```

6.2.3. 프로그램 내부

프로그램 내부에서도 프리컴파일러 옵션을 지정할 수 있다.

이 방법은 프리컴파일을 실행하는 도중에 옵션을 변경하고자 할 때 유용하게 사용할 수 있다. 또한 운영체제에 따라 입력할 수 있는 글자 길이의 제한으로 인해 명령 프롬프트에서 옵션을 지정할 수 없는 경우가 있다. 이러한 경우 환경설정 파일을 사용할 수도 있지만 프로그램 내부에서 옵션을 지정할 수도 있다.

프로그램 내부에서는 다음과 같은 형태로 옵션을 지정할 수 있다.

```
EXEC TIBERO OPTION (OPTION_NAME=value) END-EXEC.
```

프로그램 내부에서 프리컴파일러의 옵션을 지정할 때 한 가지 주의할 점은 프로그램 내부에서 지정한 옵션은 COBOL 프로그래밍 언어의 문법에 따른 변수의 영향 범위와는 전혀 무관하다는 것이다. 즉, EXEC TIBERO OPTION으로 프로그램 내부에 준 옵션 값은 소스 프로그램에서 그 문장 이후에 나오는 tbESQL/COBOL 문장에만 영향을 미친다.

예를 들면 다음과 같다.

```
...
EXEC TIBERO OPTION (HOLD_CURSOR=NO) END-EXEC.
.../* 이 부분에서는 어떠한 COBOL 프로그래밍 언어의 문법이 있더라도 그 영향 범위와는 무관하게
      모든 tbESQL/COBOL의 HOLD_CURSOR 옵션값은 NO이다. */

EXEC TIBERO OPTION (HOLD_CURSOR=YES) END-EXEC.
.../* 이 부분에서는 어떠한 COBOL 프로그래밍 언어의 문법이 있더라도 그 영향 범위와는 무관하게
      모든 tbESQL/COBOL의 HOLD_CURSOR 옵션값은 YES이다. */
```

6.3. tbESQL/COBOL 프리컴파일러 옵션 목록

본 절에서는 Tibero에서 제공하는 프리컴파일러 옵션을 알파벳 순으로 설명한다.

다음은 tbESQL 프리컴파일러 옵션을 요약한 목록이다.

옵션	설명
CLOSE_ON_COMMIT	커밋할 때 커서를 닫을 것인가를 지정한다.
CODE	전처리기로 전처리하여 생성되는 결과 파일의 코드를 지정한다.
COLUMNS	프로그램을 작성할 때 유효한 컬럼 범위를 지정한다.
COMP5	프리컴파일을 실행할 때 COMP 타입에 대하여 COMP-5 타입으로 변경할 것인지를 지정한다.
CONFIG	옵션이 기록될 환경설정 파일을 지정한다.
DB2_SYNTAX	DB2 array select/insert 문법 사용 여부를 지정한다.
DECLARE_SECTION	프리컴파일을 통해 DECLARE SECTION 내에 선언된 COMP 타입의 변수를 COMP-5로 변환하여 소스를 생성한다.
DYNAMIC	Dynamic SQL 문장의 타입을 지정한다.
END_OF_FETCH	END-OF-FETCH 상황에 대한 SQLCODE 값을 지정한다.
ERROR_CODE	ESQL 애플리케이션에서 많이 사용되는 3개의 에러 코드를 Tibero와 Oracle 중에 어느 것으로 사용할지 지정한다.
HOLD_CURSOR	커서가 닫힌 후 커서 정보를 유지할 것인가를 지정한다.
INAME	프리컴파일을 실행할 파일의 이름을 지정한다.
INCLUDE	COPY, ESQL INCLUDE 파일의 경로를 지정한다.
INSERT_NO_DATA_ERROR	INSERT, SELECT할 때에도 NO_DATA_FOUND 기능을 사용할 것인지를 지정한다.
MODE	프로그램이 전반적으로 Tibero의 형식을 따를 것인가 아니면 ANSI의 기준을 따를 것인가를 지정한다.
ONAME	출력 파일의 이름 지정한다.
ORACA	ORACA 구조체 사용 여부를 지정한다.
PARSE	전처리를 할 때 입력 파일의 내용을 어느 범위까지 파싱할 것인가를 지정한다.
PIC9_WITH_SIGN	PIC 9 타입에 (+) 부호를 가지게 할 것인지를 지정한다.
PICX	PIC X (CHAR Array)와 PIC N (NCHAR Array) COBOL 타입의 데이터 타입을 지정한다.
PREFETCH	프로그램의 속도 향상을 위해 몇 개의 로우를 미리 가져올지 지정한다.
RELEASE_CURSOR	커서가 닫힌 후 커서 정보의 해제 여부를 지정한다.

옵션	설명
RESERVED_WORD_COL	예약어 'COL' 사용 여부를 지정한다.
RESERVED_WORD_CURSOR	예약어 'CURSOR' 사용 여부를 지정한다.
RUNTIME_MODE	런타임에 사용할 드라이버를 지정한다.
SELECT_ERROR	주어진 수 보다 많은 로우가 결과로 나왔을 때 에러를 발생시킬 것인가의 여부를 지정한다.
SQLCHECK	SQL 문장의 내용을 어느 범위까지 검사할 것인가를 지정한다.
STMT_CACHE	Dynamic SQL 문장에 대해 cache size를 지정한다.
SYS_INCLUDE	전처리를 할 때 사용될 시스템 헤더 파일의 경로를 지정한다.
THREADS	멀티 스레드를 지원할 것인가를 지정한다.
TYPE_CODE	Dynamic SQL 문장의 방법 4를 사용하는 방법을 지정한다.
UNSAFE_NULL	지시자 변수가 없어도 NULL 값을 허용할 것인가를 지정한다.
USERID	SQLCHECK가 SEMANTICS으로 지정되었을 때 서버에 접속하기 위한 사용자 정보를 지정한다.
VARCHAR	사용자의 암시적인 varchar structure 사용을 허용한다.
WORDSIZE	생성되는 결과 파일을 사용할 플랫폼의 WORDSIZE를 지정한다.

6.3.1. CLOSE_ON_COMMIT

CLOSE_ON_COMMIT은 커밋을 할 때 커서를 닫을 것인가 아니면 닫지 않을 것인가를 지정하는 옵션이다.

CLOSE_ON_COMMIT의 세부 내용은 다음과 같다.

- 문법

```
CLOSE_ON_COMMIT={YES | NO}
```

항목	설명
YES	EXEC SQL COMMIT을 실행할 때 해당 트랜잭션이 처리되는 동안에 열려있던 커서를 자동으로 닫는다.
NO	EXEC SQL CLOSE cursor_name을 사용해서 사용자가 직접 열린 커서를 닫아야 한다. (기본값)

- 지정 장소

  환경설정 파일, 명령 프롬프트

6.3.2. CODE

CODE는 전처리기로 전처리하여 생성되는 결과 파일의 코드를 지정하는 옵션이다.

CODE의 세부 내용은 다음과 같다.

- 문법

```
CODE={COBOL | NET-COBOL | MF-COBOL}
```

항목	설명
COBOL	프리컴파일러는 COBOL 코드를 생성한다. (기본값)
NET-COBOL	프리컴파일러는 NET-COBOL 코드를 생성한다.
MF-COBOL	COBOL 항목과 동일하다.

- 지정 장소

  환경설정 파일, 명령 프롬프트

6.3.3. COLUMNS

COLUMNS는 프로그램을 작성할 때 유효한 컬럼 범위를 지정하는 옵션이다.

COLUMNS의 세부 내용은 다음과 같다.

- 문법

```
COLUMNS=integer
```

항목	설명
integer	기본값은 72이다.

- 지정 장소

  환경설정 파일, 명령 프롬프트

6.3.4. COMP5

COMP5는 프리컴파일을 실행할 때 COMP 타입에 대하여 COMP-5 타입으로 변경할 것인지를 지정하는 옵션이다.

COMP5의 세부 내용은 다음과 같다.

- 문법

```
COMP5={YES | NO}
```

항목	설명
YES	COMP 타입을 COMP-5 타입으로 변경한다. (기본값)
NO	COMP 타입을 COMP-5 타입으로 변경하지 않고 사용한다.

- 지정 장소

  환경설정 파일, 명령 프롬프트

6.3.5. CONFIG

CONFIG는 옵션이 기록될 환경설정 파일을 지정하는 옵션이다.

CONFIG의 세부 내용은 다음과 같다.

- 문법

```
CONFIG=filename
```

항목	설명
filename	옵션이 기록될 환경설정 파일의 이름을 명시한다. (기본값: $TB_HOME/client/config 디렉터리의 tbpcb.cfg 파일)

- 지정 장소

  명령 프롬프트

6.3.6. DB2_SYNTAX

DB2_SYNTAX는 array select/insert 구문에 대한 DB2 문법을 사용할 것인지를 지정하는 옵션이다.

DB2_SYNTAX의 세부 내용은 다음과 같다.

- 문법

```
DB2_SYNTAX={YES | NO}
```

항목	설명
YES	DB2 array select/insert 문법만을 지원한다.
NO	지원하지 않는다. (기본값)

- 지정 장소

  명령 프롬프트

6.3.7. DECLARE_SECTION

DECLARE_SECTION은 DECLARE SECTION 내에 기술된 COMP 타입 변수에 대해 COMP-5로 변환하여 소스를 생성할지 결정하는 옵션이다.

DECLARE_SECTION의 세부 내용은 다음과 같다.

- 문법

```
DECLARE_SECTION={YES | NO}
```

항목	설명
YES	DECLARE SECTION 내에 선언된 COMP 타입의 변수를 프리컴파일 후에 COMP-5로 변환하여 소스에 기술한다. DECLARE SECTION이 아닌 곳에 선언된 변수는 변환 대상이 아니다.
NO	COMP 타입의 변수를 COMP-5로 변환하지 않는다. (기본값)

- 지정 장소

  명령 프롬프트

6.3.8. DYNAMIC

DYNAMIC은 Dynamic SQL 문장의 타입을 지정하는 옵션이다.

DYNAMIC의 세부 내용은 다음과 같다.

- 문법

```
DYNAMIC={ANSI | ISO | TIBERO | ORACLE}
```

항목	설명
ANSI	ANSI 타입의 동적 SQL을 사용하도록 지정한다. 이 옵션 값이 지정되었을 경우 Tibero 타입의 동적 SQL을 사용하면 프리컴파일러에서 문법 에러를 발생한다.
ISO	ANSI와 같은 결과를 갖는다.
TIBERO	Tibero 타입의 동적 SQL을 사용하도록 지정한다. 이 옵션 값이 지정되었을 경우 ANSI 타입의 동적 SQL을 사용하면 프리컴파일러에서 문법 에러를 발생한다. (기본값)
ORACLE	TIBERO와 같은 결과를 갖는다.

참고

ANSI 타입과 ISO 타입은 실제로 동작 방법이 완전히 일치한다. 또한 TIBERO 타입과 ORACLE 타입도 동작 방법이 같다.

- 지정 장소

  환경설정 파일, 명령 프롬프트

6.3.9. END_OF_FETCH

END_OF_FETCH는 SQL 문장의 수행 후 END-OF-FETCH 상황에서 사용자에게 반환할 SQLCODE 값을 지정하는 옵션이다.

END_OF_FETCH의 세부 내용은 다음과 같다.

- 문법

```
END_OF_FETCH={1403 | 100}
```

항목	설명
1403	미설정할 경우 1403이 SQLCODE 값이 된다. (기본값)
100	ANSI 모드 또는 사용자를 지정할 경우 100이 SQLCODE 값이 된다.

- 지정 장소

  환경설정 파일, 명령 프롬프트

6.3.10. ERROR_CODE

ERROR_CODE는 ESQL 애플리케이션에서 많이 사용되는 3개의 에러 코드를 Tibero와 Oracle 중에 어느 것으로 사용할지 지정한다.

관련된 에러 코드는 다음과 같다.

- `No data found: Tibero -9072, Oracle -1403.`
- `Indicator variable was not specified in INTO clause: Tibero -9094, Oracle -1405.`
- `Unique Constraint violation: Tibero -10007, Oracle -1.`

ERROR_CODE의 사용법은 다음과 같다.

- 문법

```
ERROR_CODE={ TIBERO | ORACLE }
```

항목	설명
TIBERO	Tibero 에러 코드를 사용한다.
ORACLE	Oracle 에러 코드를 사용한다. (기본값)

- 지정 장소

  환경설정 파일, 명령 프롬프트

6.3.11. HOLD_CURSOR

HOLD_CURSOR는 커서가 닫힌 후 커서 정보를 유지할 것인가를 지정하는 옵션이다.

HOLD_CURSOR의 세부 내용은 다음과 같다.

- 문법

```
HOLD_CURSOR={YES | NO}
```

항목	설명
YES	커서가 닫힌 후 커서의 정보를 유지한다.
NO	커서가 닫힌 후 커서의 정보를 삭제한다. (기본값)

- 지정 장소

  명령 프롬프트, 프로그램 내부

6.3.12. INAME

INAME은 프리컴파일을 실행할 대상 파일을 지정하는 옵션이다.

INAME의 세부 내용은 다음과 같다.

- 문법

```
INAME=filename
```

다음은 INAME의 항목에 대한 설명이다.

항목	설명
filename	프리컴파일을 실행할 대상 파일의 이름을 명시한다. (기본값: 없음)

- 지정 장소

  명령 프롬프트

- 사용법

  파일의 이름을 지정할 때 'INAME=' 부분과 파일의 확장자가 tbco일 경우 .tbco도 생략할 수 있다. 따라서 다음의 예는 모두 같은 의미이다.

```
tbpcb INAME=test1.tbco
tbpcb test1.tbco
tbpcb test1
```

6.3.13. INCLUDE

INCLUDE는 COPY, ESQL INCLUDE 파일이 위치한 경로를 지정하는 옵션이다.

프로그램 소스 내부에 사용된 COPY, ESQL INCLUDE 파일이 같은 디렉터리에 있지 않은 경우 프리컴파일할 때 에러가 발생한다. COPY, ESQL INCLUDE 파일이 존재하는 디렉터리를 지정해 주어야 하며, 그 경우 이 옵션을 사용한다.

INCLUDE의 세부 내용은 다음과 같다.

- 문법

```
INCLUDE=pathname
```

항목	설명
pathname	COPY, ESQL INCLUDE 파일이 존재하는 디렉터리를 명시한다. (기본값: 없음)

- 지정 장소

  환경설정 파일, 명령 프롬프트

6.3.14. INSERT_NO_DATA_ERROR

현재 no data found 처리는 UPDATE와 DELETE 문에 대해서만 적용되고 있다. INSERT_NO_DATA_ERROR는 insert, select 구문 사용 시에도 no data found 처리를 할 것인지 지정하는 옵션이다.

INSERT_NO_DATA_ERROR의 세부 내용은 다음과 같다.

- 문법

```
INSERT_NO_DATA_ERROR={YES | NO}
```

항목	설명
YES	EXEC SQL INSERT, SELECT 사용 시에 선택된 row가 없으면 NO_DATA_FOUND를 반환한다.
NO	선택된 row가 없어도 NO_DATA_FOUND 처리 없이 진행된다. (기본값)

- 지정 장소

  환경설정 파일, 명령 프롬프트

6.3.15. MODE

MODE는 tbESQL/COBOL 프로그램이 전반적으로 Tibero의 형식을 따를 것인지 아니면, ANSI의 기준을 따를 것인지를 지정하는 옵션이다. 이 옵션은 여러 가지 옵션을 한꺼번에 지정해 주는 역할을 한다. 이 옵션으로 CLOSE_ON_COMMIT, DYNAMIC, TYPE_CODE 등의 옵션 값을 한꺼번에 지정할 수 있으며, 추가로 다른 조건을 지정할 수도 있다.

MODE의 세부 내용은 다음과 같다.

- 문법

```
MODE={ANSI | ISO | TIBERO | ORACLE}
```

항목	설명
ANSI	ANSI 타입의 동적 SQL을 사용하도록 지정한다. 이 옵션 값이 지정되었을 경우 Tibero 타입의 동적 SQL을 사용하면 프리컴파일러에서 문법 에러를 발생한다.
ISO	ANSI와 같은 결과를 갖는다.

항목	설명
TIBERO	Tibero 타입의 동적 SQL을 사용하도록 지정한다. 이 옵션 값이 지정되었을 경우 ANSI 타입의 동적 SQL을 사용하면 프리컴파일러에서 문법 에러를 발생한다. (기본값)
ORACLE	TIBERO와 같은 결과를 갖는다.

다음은 MODE 옵션에 지정된 값에 따라 설정되는 세부 옵션이다.

옵션	ANSI	TIBERO
CLOSE_ON_COMMIT	YES	NO
DYNAMIC	ANSI	TIBERO
TYPE_CODE	ANSI	TIBERO

참고

ANSI는 ISO와 TIBERO는 ORACLE과 동일한 기능을 한다.

- 지정 장소

  환경설정 파일, 명령 프롬프트

6.3.16. ONAME

ONAME은 프리컴파일러의 결과물로 나오는 출력 파일의 이름을 지정하는 옵션이다.

ONAME의 세부 내용은 다음과 같다.

- 문법

```
ONAME=filename
```

항목	설명
filename	원하는 출력 파일의 이름을 명시한다. 기본값은 입력된 파일 이름에서 확장자만 .cob로 바꾼다.

- 지정 장소

  환경설정 파일, 명령 프롬프트

6.3.17. ORACA

ORACA는 Oracle Communications Area 구조체를 사용할지 여부를 지정하는 옵션이다.

ORACA의 세부 내용은 다음과 같다.

- 문법

```
ORACA={YES | NO}
```

항목	설명
YES	프로그램 내에 EXEC SQL INCLUDE ORACA 문장이 존재해야 한다.
NO	프로그램 내에 EXEC SQL INCLUDE ORACA 문장이 존재하지 않아도 된다. (기본값)

- 지정 장소

  환경설정 파일, 명령 프롬프트

6.3.18. PARSE

PARSE는 tbESQL/COBOL 문장 내용을 전처리할 때 어느 범위까지 파싱할 것인가를 지정하는 옵션이다.

PARSE의 세부 내용은 다음과 같다.

- 문법

```
PARSE={NONE | PARTIAL | FULL}
```

항목	설명
NONE	EXEC SQL {BEGIN | END} DECLARE SECTION 안에 있는 호스트 변수 선언들만을 처리한다.
PARTIAL	EXEC SQL {BEGIN | END} DECLARE SECTION 안에 있는 호스트 변수 선언들과 모든 매크로 명령들을 처리한다.
FULL	전처리기에 내장되어 있는 cobol parser가 EXEC SQL {BEGIN | END} DECLARE SECTION 안에 뿐만 아니라 밖에 선언된 변수들도 처리를 하며, include되는 copybook의 매크로, 변수까지도 처리를 한다. (기본값)

- 지정 장소

  명령 프롬프트, 프로그램 내부

6.3.19. PIC9_WITH_SIGN

PIC9_WITH_SIGN는 부호를 가지지 않는 PIC 9 타입에 (+) 부호를 가지게 할지 여부를 지정하는 옵션이다.

PIC9_WITH_SIGN의 세부 내용은 다음과 같다.

- 문법

```
PIC9_WITH_SIGN={YES | NO}
```

항목	설명
YES	PIC 9 타입이 (+) 부호를 가진다. (기본값)
NO	PIC 9 타입이 부호를 가지지 않는다.

- 지정 장소

  환경설정 파일, 명령 프롬프트

6.3.20. PICX

PICX는 PIC X (CHAR Array) COBOL 타입의 데이터 타입을 지정한다.

PICX 옵션은 INSERT와 WHERE절에 적용되는 옵션이다. VARCHAR2 옵션으로 지정할 때에 CHAR Array와 NCHAR Array에 저장되어 있는 공백을 모두 지운 후에 INSERT 동작을 하게 된다. 따라서 VARCHAR 타입의 컬럼에 INSERT하면 공백이 모두 지워진 채로 INSERT되며, CHAR 타입의 컬럼에 INSERT하면 컬럼의 크기에서 데이터의 크기를 제외한만큼 공백이 들어간다.

WHERE 절을 사용할 때에도 공백 제거가 적용된다. 데이터가 같고 공백의 수가 같더라도 VARCHAR2 옵션을 사용할 때에는 공백이 제거되기 때문에 WHERE 절에서 매칭되지 않는다.

PICX의 세부 내용은 다음과 같다.

- 문법

```
PICX={CHARF | VARCHAR2}
```

항목	설명
CHARF	PIC X (CHAR Array)와 PIC N (NCHAR Array)가 각각 VARCHAR, NVARCHAR 타입으로 지정된다.
VARCHAR2	PIC X (CHAR Array)와 PIC N (NCHAR Array)가 각각 기존의 타입인 CHAR, NCHAR 타입으로 지정된다. (기본값)

- 지정 장소

  명령 프롬프트

6.3.21. PREFETCH

PREFETCH는 속도 향상을 위해 몇 개의 로우를 미리 가져올지를 지정하는 옵션이다.

PREFETCH의 세부 내용은 다음과 같다.

- 문법

```
PREFETCH=integer
```

항목	설명
integer	미리 가져올 로우의 개수를 지정한다. (기본값: 1)

- 지정 장소

  환경설정 파일, 명령 프롬프트

6.3.22. RELEASE_CURSOR

RELEASE_CURSOR는 커서가 닫힌 후 커서에 저장된 정보의 해제 여부를 지정하는 옵션이다.

RELEASE_CURSOR의 세부 내용은 다음과 같다.

- 문법

```
RELEASE_CURSOR={YES | NO}
```

항목	설명
YES	커서가 닫히면 커서의 정보를 해제한다.
NO	커서가 닫혀도 커서의 정보를 해제하지 않는다. (기본값)

- 지정 장소

  명령 프롬프트, 프로그램 내부

6.3.23. RESERVED_WORD_COL

RESERVED_WORD_COL는 예약어 'COL' 사용 여부를 지정하는 옵션이다.

RESERVED_WORD_COL의 세부 내용은 다음과 같다.

- 문법

```
RESERVED_WORD_COL={YES | NO}
```

항목	설명
YES	예약어 'COL'을 사용자 변수명으로 사용할 수 없다. (기본값)
NO	예약어 'COL'을 사용자 변수명으로 사용할 수 있다.

- 지정 장소

  환경설정 파일, 명령 프롬프트

6.3.24. RESERVED_WORD_CURSOR

RESERVED_WORD_CURSOR는 예약어 'CURSOR' 사용 여부를 지정하는 옵션이다.

RESERVED_WORD_CURSOR의 세부 내용은 다음과 같다.

- 문법

```
RESERVED_WORD_CURSOR={YES | NO}
```

항목	설명
YES	예약어 'CURSOR'를 사용자 변수명으로 사용할 수 없다. (기본값)
NO	예약어 'CURSOR'를 사용자 변수명으로 사용할 수 있다.

- 지정 장소

  환경설정 파일, 명령 프롬프트

6.3.25. RUNTIME_MODE

RUNTIME_MODE는 런타임에 사용할 드라이버를 지정하는 옵션이다.

RUNTIME_MODE의 세부 내용은 다음과 같다.

- 문법

```
RUNTIME_MODE={ODBC | TIBERO}
```

항목	설명
ODBC	odbc 함수를 사용한다. 컴파일과 링크 과정에서 tbERTL 라이브러리 이외에 odbc 라이브러리를 함께 링크해야 한다.
TIBERO	tbCLI 함수를 사용한다. 컴파일과 링크 과정에서 tbERTL 라이브러리 이외에 tbCLI 라이브러리를 함께 링크해야 한다. (기본값)

- 지정 장소

  명령 프롬프트, 프로그램 내부

6.3.26. SELECT_ERROR

SELECT_ERROR는 호스트 변수 등으로 인해 주어진 수행 결과 개수보다 실제로 질의를 수행한 결과가 더 많이 반환된 경우 에러를 발생시킬 것인가를 지정하는 옵션이다.

SELECT_ERROR의 세부 내용은 다음과 같다.

- 문법

```
SELECT_ERROR={YES | NO}
```

항목	설명
YES	더 많은 로우가 반환되었을 경우 에러를 발생시킨다. (기본값)
NO	더 많은 로우가 반환되어도 에러를 발생시키지 않는다.

- 지정 장소

  명령 프롬프트, 프로그램 내부

- 사용법

  다음은 tbESQL/COBOL 프로그램의 예이다.

```
01 EMPNO PIC X(4).
01 PNO PIC X(4).

EXEC SQL SELECT EMPNUM, PNUM
         INTO :EMPNO, :PNO
         FROM WORKS
         WHERE EMPNUM = 'E3'
END-EXEC.
```

위 예에서 만약 질의를 수행한 결과가 더 많은 로우를 반환하는 경우 SELECT_ERROR 옵션에 지정된 값에 따라 다음과 같이 나타나는 결과가 다르다.

항목	결과
YES	ERROR_ESQL_TOO_MANY_ROW_IN_SELECT 에러가 발생한다.
NO	수행된 질의 결과의 첫 번째 로우를 호스트 변수에 할당한다. 이때 에러는 발생하지 않는다.

6.3.27. SQLCHECK

SQLCHECK는 tbESQL/COBOL 문장의 내용을 어느 범위까지 검사할 것인가를 지정하는 옵션이다.

SQLCHECK의 세부 내용은 다음과 같다.

- 문법

```
SQLCHECK={SEMANTICS | FULL | SYNTAX}
```

항목	설명
SEMANTICS	SEMANTICS로 지정되면, 프리컴파일러는 서버에 접속을 시도한다. 따라서 SEMANTICS로 설정하기 위해서는 USERID 옵션이 반드시 필요하다. 접속이 성공하면 프리컴파일러는 SQL 문장을 서버로 보내 문법뿐만 아니라 SQL 문장이 참조하는 스키마 객체의 내용까지 검사하며, 컬럼의 타입 정보 등을 바탕으로 올바른 타입끼리 호스트 변수에 바인딩 되는지도 검사한다. tbESQL/COBOL 문장뿐만 아니라 DDL이나 PSM일 경우에도 문법과 내용을 모두 검사한다.
FULL	SEMANTICS와 동일한 의미를 갖는다.
SYNTAX	tbpcb 유틸리티는 서버에 접속하지 않고 자체적으로 ESQL 문법과 SQL 문법을 검사한다. (기본값)

- 지정 장소

  명령 프롬프트, 프로그램 내부

6.3.28. STMT_CACHE

STMT_CACHE는 Dynamic SQL 문장에 대해 cache size를 지정하는 옵션이다.

STMT_CACHE의 세부 내용은 다음과 같다.

- 문법

```
STMT_CACHE=integer
```

항목	설명
integer	애플리케이션에서 distinct한 Dynamic SQL문에 대해 cache size를 지정한다. (기본값: 10)

- 지정 장소

  환경설정 파일, 명령 프롬프트

6.3.29. SYS_INCLUDE

SYS_INCLUDE는 시스템 헤더 파일이 위치한 경로를 지정하는 옵션이다.

SYS_INCLUDE의 세부 내용은 다음과 같다.

- 문법

```
SYS_INCLUDE=pathname
```

항목	설명
pathname	시스템 헤더 파일이 존재하는 디렉터리를 명시한다. (기본값: 없음)

- 지정 장소

  환경설정 파일, 명령 프롬프트

6.3.30. THREADS

THREADS는 멀티 스레드 지원 여부를 지정하는 옵션이다.

THREADS의 세부 내용은 다음과 같다.

- 문법

```
THREADS={YES | NO}
```

항목	설명
YES	멀티 스레드를 지원한다.
NO	멀티 스레드를 지원하지 않는다. (기본값)

- 지정 장소

  환경설정 파일, 명령 프롬프트

- 사용법

  다음과 같은 문장에 멀티 스레드를 사용하려면, THREADS 옵션의 값은 반드시 **YES**로 설정되어 있어야 한다.

  - ENABLE THREADS
  - CONTEXT USE
  - CONTEXT ALLOCATE
  - CONTEXT FREE

6.3.31. TYPE_CODE

TYPE_CODE는 Dynamic SQL 문장을 사용하는 방법 중에 방법 4를 사용하는 방법을 지정하는 옵션이다.

TYPE_CODE의 세부 내용은 다음과 같다.

- 문법

```
TYPE_CODE={ANSI | ISO | TIBERO | ORACLE}
```

항목	설명
ANSI	ANSI 타입의 동적 SQL을 사용하도록 지정한다. 이 옵션 값이 지정되었을 경우 Tibero 타입의 동적 SQL을 사용하면 프리컴파일러에서 문법 에러를 발생한다.
ISO	ANSI와 같은 결과를 갖는다.
TIBERO	Tibero 타입의 동적 SQL을 사용하도록 지정한다. 이 옵션 값이 지정되었을 경우 ANSI 타입의 동적 SQL을 사용하면 프리컴파일러에서 문법 에러를 발생한다. (기본값)
ORACLE	TIBERO와 같은 결과를 갖는다.

참고

ANSI 타입은 ISO 타입과 동작 방법이 동일하며, TIBERO 타입은 ORACLE 타입과 동작 방법이 동일하다.

- 지정 장소

  환경설정 파일, 명령 프롬프트

6.3.32. UNSAFE_NULL

UNSAFE_NULL은 지시자 변수가 없을 때 NULL 값을 허용할 것인지의 여부를 지정하는 옵션이다. NULL 데이터가 예상되는 경우에도 편의상 지시자 변수 사용을 생략하고 싶을 때 이 옵션을 사용한다.

UNSAFE_NULL의 세부 내용은 다음과 같다.

- 문법

```
UNSAFE_NULL={YES | NO}
```

항목	설명
YES	NULL을 허용한다. 지시자 변수 없이 NULL 데이터를 호스트 변수로 받아도 에러는 발생하지 않는다.
NO	NULL을 허용하지 않는다. (기본값) 지시자 변수 없이 NULL 데이터를 호스트 변수로 받으면 **1405** 에러를 발생한다. **1405** 에러는 가져온 데이터가 NULL이나 지시자 변수가 존재하지 않아 tbESQL/COBOL 런타임 라이브러리가 개발자에게 NULL 값의 존재 여부를 알려줄 방법이 없을 경우 발생되는 에러이다.

- 지정 장소

  환경설정 파일, 명령 프롬프트

6.3.33. USERID

USERID는 서버 접속을 할 때 필요한 사용자 계정의 정보를 지정하는 옵션이다. 이 옵션은 SQL 문장의 내용을 검사하기 위한 접속 정보일 뿐이며, 실제 데이터베이스를 실행할 때의 접속 정보와는 무관하다.

USERID의 세부 내용은 다음과 같다.

- 문법

```
USERID=username/password
```

항목	설명
username	사용자의 이름을 명시한다.

항목	설명
password	사용자의 패스워드를 명시한다.

- 지정 장소

  명령 프롬프트

- 사용법

  SQLCHECK를 SEMANTICS나 FULL로 지정할 경우 반드시 이 옵션을 사용해야 한다.

6.3.34. VARCHAR

VARCHAR는 사용자가 선언한 COBOL group item을 ESQL의 VARCHAR 데이터 타입으로 인식하도록 지정하는 옵션이다.

VARCHAR의 세부 내용은 다음과 같다.

- 문법

```
VARCHAR={YES | NO}
```

항목	설명
YES	사용자의 implicit한 group item을 VARCHAR 데이터 타입으로 처리한다. 사용자는 반드시 49레벨에 VARNAME-LEN과 VARNAME-ARR을 필드로 선언해야 한다.
NO	implicit한 group item을 VARCHAR external 데이터 타입으로 허용하지 않는다. (기본값)

- 지정 장소

  명령 프롬프트

6.3.35. WORDSIZE

WORDSIZE는 64bits에서 32bits용 소스 또는 그 반대로도 프리컴파일러로 소스를 생성할 수 있도록 하는 옵션이다. (기본값: 64)

WORDSIZE의 세부 내용은 다음과 같다.

- 문법

```
WORDSIZE={64 | 32}
```

- 지정 장소

  환경설정 파일, 명령 프롬프트

색인

O

P

R

S

T

U

V

W

ㄱ

ㄴ

ㄷ

ㅁ

ㅂ

ㅅ

ㅇ

ㅈ

ㅊ

ㅋ

ㅍ