Data → App → DBMS ↔ OS File I/O API ↔ 파일

App
App
App

- 입출력
- 순서대로
- 동시

SQL
DDL
DML(+DQL)

Join
Cross
Natural
Join ~using
Join ~on =inner join
Outerjoin (left / right)

SQL → Oracle

SQL → MS-SQL

SQL → MYSQL

[결론서]
- 각 DBMS마다 문법, 프로그램이 다름.

Data → Table
중복 방지
→ Table
중복 방지
→ Table

데이터베이스 모델링 - DB모델링
1.데이터가 중복되지 않도록 여러군데로 나눔.
2.테이블간의 관계 (제약조건을 추가)
3.유니크 컬럼 /낫널 여부 설정

*DBMS native Api 등장 ▶DBMS Client 역할, App이 사용

1.사용자가 직접 SQL 작성

사용자 ▶ SQL → [DBMS] client → [DBMS] SERVER ↔ 파일

요청
통신
입출력
주고 받는 규칙(프로토콜)
mysql.exe
mysql.exe
←
응답

✓사람이 아니라 기계가 사용

사용자 ▶App SQL → [DBMS] API → [DBMS] SERVER ↔ 파일

call
요청
통신
입출력
ex)Board-app-client
←
return
mysql.exe
←
응답

▶App에서 DBMS Server에게 SQL을 보낼때 사용하는 메서드(함수)
▶= DBMS Native Api
▶ 언어 별로 제공 > C/C++ , Python 등

*DBMS NATIVE API : DBMS VENDER 에서 제공
*DBMS Native API

단점
DBMS에 맞춰
App을 개발
↓
DBMS에 종속

App → Oracle API → Oracle DBMS
Oracle 용
SQL call → ← return
통신 요청 → ← 응답

App → MS-SQL API → MS-SQL DBMS
MS-SQL 용
SQL call → ← return
통신 요청 → ← 응답

App → MYSQL API → MYSQL DBMS
MySQL 용
SQL call → ← return
통신 요청 → ← 응답

- 언어 별 제공
- DBMS마다 API 규격이 다르다. = 사용법이 다르다.

*ODBC API : DBMS API의 표준 규격
>Open Database Connectivity
개발원 연계자

DBMS 제조사 마음대로가 아니라
ODBC API 표준 규격에 따라 API를 만들었다.

ODBC API 표준 규격에 따라 CALL

APP → Oracle ODBC Driever → Oracle DBMS
APP → MS-SQL ODBC Driver → MS-SQL DBMS
APP → MYSQL ODBC Driver → MYSQL DBMS

통신 요청 → ← 응답

API 규격이 같기 때문에
메서드 호출 코드도 동일하다
> DBMS마다 변경할 필요가 없다.

*JAVA에서 DBMS와 연결할 자바 전용 API가 필요하다.

Java에서 C/C++ 코드를 직접 호출 할 수 없다.

JAVA App → CALL → java DBMS API → RETURN
CALL

C/C++로 작성된

Oracle ODBC Driver ↔ Oracle DBMS
요청/ 응답

MS-SQL ODBC Driver ↔ MS-SQL DBMS
요청/ 응답

MySQL ODBC ↔ MYSQL DBMS
요청/ 응답

ODBC API를
호출할
JAVA API가 필요 → JAVA DATABASE cennectivity api 등장
(JDBC)

Driver

*JDBC API : 자바에서 DBMS와 연결할때 사용하는 객체의 호출 규격을 정의한 것

1. "Type 1 Driver"

> 인터페이스

↘ 규격에 따라 클래스와 메서드를 구현

JAVA App → JDBC Driver (CALL) ← (RETURN)

JDBC Driver → CALL / CALL / CALL

Oracle ODBC Driver ↔ Oracle DBMS (요청/ 응답)

MS-SQL ODBC Driver ↔ MS-SQL DBMS (요청/ 응답)

MySQL ODBC Driver ↔ MYSQL DBMS (요청/ 응답)

"LOCAL" "Remote"

> JRE에 기본으로 포함되어 있음
> ODBC Driver 호출
> Local에 ODBC 드라이버 설치 필요

ODBC API를 호출하는 JDBC Driver를
"Type 1 Driver"
"ODBC-JDBC Bridge"

*JDBC API : 자바에서 DBMS와 연결할때 사용하는 객체의 호출 규격을 정의한 것

2. "Type 2 Driver"

JAVA App → Oracle Type2 Driver (CALL → / ← RETURN) → Oracle Native Api ↔ Oracle DBMS (요청/ 응답)

C/C++ 언어로 구성

JAVA App → MS-SQL Type2 Driver (CALL → / ← RETURN) → MS-SQL Native Api ↔ MS-SQL DBMS (요청/ 응답)

C/C++ 언어로 구성

JAVA App → MySQL Type2 Driver (CALL → / ← RETURN) → MySQL Native Api ↔ MYSQL DBMS (요청/ 응답)

JDBC API 규격에 따라 제작 "LOCAL" "Remote"

> DBMS Native Api 호출
> Local에 DBMS 전용 API 설치 필요
> DBMS Vender로 부터 다운로드 받아야 한다.

Type2 Driver은 자바프로그램이 설치된 폴더에 있어야함.
DBMS Vender로 부터 받아야함 OCI도 받아야함.

Type1이든 Type2든 둘다 JDBC API 규격에 따라 작성되었기 때문에 사용법이 같다.
> JAVA APP을 변경할 필요가 없다.
>"DBMS Native Call 드라이버"

*JDBC API : 자바에서 DBMS와 연결할 때 사용하는 객체의 호출 규격(인터페이스)를 정의하는 것

3.Type3 driver

DBMS가 변경되면 -> Native API/ODBC API 변경
Type2 드라이버 변경

>DBMS에 종속되는 문제 해결
↓
중간 서버를 통해 해결

JAVA App → call → JDBC Type3 Driver → 요청(통신) → JDBC Type3 서버

call /return

Oracle JDBC2 Driver (CALL → / ← RETURN) Oracle Native Api (C/C++ 언어로 구성) (CALL → / ← RETURN) Oracle DBMS

MS-SQL JDBC2 Driver (CALL → / ← RETURN) MS-SQL Native Api (C/C++ 언어로 구성) MS-SQL DBMS

MYSQL JDBC2 Driver (CALL → / ← RETURN) MySQL Native Api (Remote2) MYSQL DBMS

-so(Unix/linux)
-dll(windows)

*DBMS를 바꾸더라도 Local 쪽은 변경할 필요가 없다. ---------->
*Local에는 DBMS와 관련된 파일을 설치할 필요가 없다. ---------->
>여기서 변경하기 때문이다.
>DBMS와 관련된 파일은
여기에 설치하기 때문이다.

Local Remote1 Remote2

"네트워크 드라이버"

"Middleware" (미들웨어)
중간에서 조율하는 소프트웨어

*Type4 Driver

- Local에 ODBC API / Netive API를 설치해야하는 문제를 해결하는 것
- DBMS 서버와 직접 통신
- DBMS Vender에서 다운로드 해야 한다.
- DBMS마다 Driver도 변경해야 한다.
*Locla에 DBMS 관련 ODBC API나 Netive API를 설치할 필요 없다.
- C/C++ (OS종속)

*JDBC Driver만 설치한다. = "Pure Java"
★"DBMS Protocal Driver"

JAVA App → call / return

Oracle Type4 Driver (통신 → / ← 요청 / 응답) Oracle DBMS

MS-SQL Type4 Driver (통신 → / ← 요청 / 응답) MS-SQL DBMS

MYSQL Type4 Driver (통신 → / ← 요청 / 응답) MYSQL DBMS

*JDBC programming

JDBC API java.sql.*
△ javax -> extension

JAVA App	SQL - Call Return	MYSQL Type4 Driver	요청 응답

1.JDBC Driver 다운로드 *.jar
2.프로젝트 classpath에 추가
3.JDBC API 규칙에 따라 Call

---

*JDBC programming 절차

1.java.sql.Driver 구현체 준비<---JDBC Driver에 대한 정보를 갖고 있다.
2.java.sql.DriverManager에 Driver 구현체 등록
3.DriverManager를 통해 java.sql.connection 객체 얻기 <---- DBMS와의 연결정보 갖고 있음.
4.connection을 통해 java.sql.statement/preparedstatedment 객체를 준비시킨다. <----SQL 서버에 보내고 응답을 받는 일을 한다.
5.Statement/PreparedStatement를 통해 java.sql.Resultset을 얻는다. <----- DBMS의 Select결과를 한 객체씩 가져오는 일을 한다.

---

*JDBC API의 기본 객체

Driver 구현체 ▸ java.sql.Driver 인터페이스를 구현한 클래스의 인스턴스

1. 등록 ▼ 등록된 객체에게 위임 = connetion 객체 리턴
2. 연결 요청 ▸ Driver Manager 위임▸ Driver 구현체 생성▸ connetion 구현체
2. DBMS에 연결 및 연결정보 리턴 ▼
4. 질의 준비 ▸ connection 구현체 생성▸ ◂return Statement/PreparedStatement 구현체
5.SQL을 서버에 보낼 객체를 리턴 ▼
6. 질의 수행 ▸ Statememt/preparedStatement 구현체 생성▸ Resultset 구현체
◂7. 질의 결과 리턴 ◂return
insert / update/delete
8.Select를 질의한 경우 DBMS 서버에서 결과 record(tuple/row)를 가져올 객체 리턴▼
9. record 요청 ▸ ResultSet 구현체
▼
10.record 한개 DBMS 서버에서 가져온다

---

*JDBC API의 기본 객체 - 메서드 호출

1. new ▸ Driver 구현체 "Service - probider loading" 규칙에 따라 자동으로 드라이버를 로딩한다.
2. registerDriver() ▼
3. getConnection ▸ Driver Manager 4. Connect ▸ Driver 구현체 5. new ▸ connetion 구현체

connection 구현체

Statememt/preparedStatement 구현체

ResultSet 구현체

---

*클래스 로딩

1. new 클래스( ); ← 인스턴스를 생성할 때

2. 클래스.스태틱필드 = 값; ← 스태틱필드를 사용할 때
*그전에 클래스가 로딩되어있으면 다시 로딩하지 않음.
3. 클래스.스태틱메서드( ) ; ← 스태틱메서드를 호출할 때

4. Class.forName("패키지.클래스"); ← 명시적으로 로딩
fully-qualified class name(FQName=QName)
(완전히 지정된 이름 = 패키지를 포함한 전체이름)

---

java	-Dname -Dage=30	hello	aaa	bbb	ccc
				program argument	

클라이언트	→	t	
c	→	t	서버
c	→	t	

다 쓰고 난 후에 close 하는 것을 잊으면 안됨.