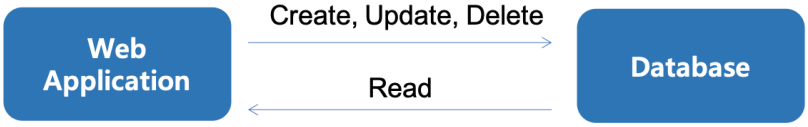
ORM(Object Relational Mapping)을 이해한다.

Goal

> - 영속성(Persistence)이란
> - ORM(Object Relational Mapping)이란
> - ORM의 장단점
> - The Object-Relational Impedance Mismatch
> - Association(연관성)
>   - One-To-One Relationship
>   - One-To-Many Relationship

영속성(Persistence)

- 데이터를 생성한 프로그램이 종료되더라도 사라지지 않는 데이터의 특성을 말한다.
- 영속성을 갖지 않는 데이터는 단지 메모리에서만 존재하기 때문에 프로그램을 종료하면 모두 잃어버리게 된다.
- **Object Persistence(영구적인 객체)**
  - 메모리 상의 데이터를 파일 시스템, 관계형 테이터베이스 혹은 객체 데이터베이스 등을 활용하여 영구적으로 저장하여 영속성 부여한다.
  - 

  - 데이터를 데이터베이스에 저장하는 3가지 방법
    - 1) JDBC (java에서 사용)
    - 2) Spring JDBC (Ex. JdbcTemplate)
    - 3) Persistence Framework (Ex. Hibernate, Mybatis 등)
- **Persistence Layer**
  - 프로그램의 아키텍처에서, 데이터에 영속성을 부여해주는 계층을 말한다.
  - JDBC를 이용하여 직접 구현할 수 있지만 Persistence framework를 이용한 개발이 많이 이루어진다.
- **Persistence Framework**
  - JDBC 프로그래밍의 복잡함이나 번거로움 없이 간단한 작업만으로 데이터베이스와 연동되는 시스템을 빠르게 개발할 수 있으며 안정적인 구동을 보장한다.
  - Persistence Framework는 SQL Mapper와 ORM으로 나눌 수 있다.
    - Ex) JPA, Hibernate, Mybatis 등

ORM이란

Object Relational Mapping, 객체-관계 매핑

- 객체와 관계형 데이터베이스의 데이터를 **자동으로 매핑(연결)해주는 것**을 말한다.
  - 객체 지향 프로그래밍은 **클래스**를 사용하고, 관계형 데이터베이스는 **테이블**을 사용한다.
  - 객체 모델과 관계형 모델 간에 불일치가 존재한다.
  - ORM을 통해 객체 간의 관계를 바탕으로 SQL을 자동으로 생성하여 불일치를 해결한다.
- 데이터베이스 데이터 <—매핑—> Object 필드
  - 객체를 통해 간접적으로 데이터베이스 데이터를 다룬다.
- Persistant API라고도 할 수 있다.
  - Ex) JPA, Hibernate 등

ORM의 장단점

- 장점
  - 객체 지향적인 코드로 인해 더 직관적이고 **비즈니스 로직에 더 집중**할 수 있게 도와준다.
    - ORM을 이용하면 SQL Query가 아닌 직관적인 코드(메서드)로 데이터를 조작할 수 있어 개발자가 객체 모델로 프로그래밍하는 데 집중할 수 있도록 도와준다.
    - 선언문, 할당, 종료 같은 부수적인 코드가 없거나 급격히 줄어든다.
    - 각종 객체에 대한 코드를 별도로 작성하기 때문에 코드의 가독성을 올려준다.
    - SQL의 절차적이고 순차적인 접근이 아닌 객체 지향적인 접근으로 인해 생산성이 증가한다.
  - **재사용 및 유지보수**의 편리성이 증가한다.
    - ORM은 독립적으로 작성되어있고, 해당 객체들을 재활용 할 수 있다.
    - 때문에 모델에서 가공된 데이터를 컨트롤러에 의해 뷰와 합쳐지는 형태로 디자인 패턴을 견고하게 다지는데 유리하다.
    - 매핑정보가 명확하여, ERD를 보는 것에 대한 의존도를 낮출 수 있다.
  - **DBMS에 대한 종속성이 줄어든다.**
    - 객체 간의 관계를 바탕으로 SQL을 자동으로 생성하기 때문에 RDBMS의 데이터 구조와 Java의 객체지향 모델 사이의 간격을 좁힐 수 있다.
    - 대부분 ORM 솔루션은 DB에 종속적이지 않다.
    - 종속적이지 않다는것은 구현 방법 뿐만아니라 많은 솔루션에서 자료형 타입까지 유효하다.
    - 프로그래머는 Object에 집중함으로 극단적으로 DBMS를 교체하는 거대한 작업에도 비교적 적은 리스크와 시간이 소요된다.
    - 또한 자바에서 가공할경우 equals, hashCode의 오버라이드 같은 자바의 기능을 이용할 수 있고, 간결하고 빠른 가공이 가능하다.
- 단점
  - 완벽한 ORM 으로만 서비스를 구현하기가 어렵다.
    - 사용하기는 편하지만 설계는 매우 신중하게 해야한다.

    - 프로젝트의 복잡성이 커질경우 난이도 또한 올라갈 수 있다.
    - 잘못 구현된 경우에 속도 저하 및 심각할 경우 일관성이 무너지는 문제점이 생길 수 있다.
    - 일부 자주 사용되는 대형 쿼리는 속도를 위해 SP를 쓰는등 별도의 튜닝이 필요한 경우가 있다.
    - DBMS의 고유 기능을 이용하기 어렵다. (하지만 이건 단점으로만 볼 수 없다 : 특정 DBMS의 고유기능을 이용하면 이식성이 저하된다.)
  - 프로시저가 많은 시스템에선 ORM의 객체 지향적인 장점을 활용하기 어렵다.
    - 이미 프로시저가 많은 시스템에선 다시 객체로 바꿔야하며, 그 과정에서 생산성 저하나 리스크가 많이 발생할 수 있다.

The Object-Relational Impedance Mismatch

Mismatch	Description
Granularity	Sometimes you will have an object model which has more classes than the number of corresponding tables in the database. Lets take an example of Person details, we could broke down person details into two classes one is Person and another is Address for code reusability and code maintainability purpose. But assume that to store Person details in database there is only one table called Person.
Inheritance	RDBMSs do not define anything similar to Inheritance which is a natural paradigm in object-oriented programming languages.
Identity	A RDBMS defines exactly one notion of 'sameness': the primary key. Java, however, defines both object identity (a==b) and object equality (a.equals(b)).
Associations	Object-oriented languages represent associations using object references whereas an RDBMS represents an association as a foreign key column.
Navigation	The ways you access objects in Java and in a RDBMS are fundamentally different. In Java, you navigate from one association to another walking the object network (graph). For example, aUser.getBillingDetails().getAccountNumber(). This is not an efficient way of retrieving data from a relational database. You typically want to minimize the number of SQL queries and thus load several entities via JOINs and select the targeted entities before you start walking the object network.

- Granularity(세분성)
  - 경우에 따라 데이터베이스에 있는 해당 테이블 수보다 더 많은 클래스를 가진 객체 모델을 가질 수 있다.
  - 예를 들어, "사용자 세부 사항"에 대해 생각해보자.
    - 코드 재사용과 유지보수를 위해 "Person"과 "Address"라는 **두 개의 클래스**로 나눌 수 있다.
    - 그러나 데이터베이스에는 person이라는 **하나의 테이블**에 "사용자 세부 사항"을 저장할 수 있다.
    - 이렇게 Object 2개와 Table 1개로 두 개의 갯수가 다를 수 있다.
  - 1) Coarse Granularity(굵은/거친): PersonDetails Class
  - 2) Fine Granularity(가는/세밀한): Persion Class, Address Class
- Inheritance(상속)

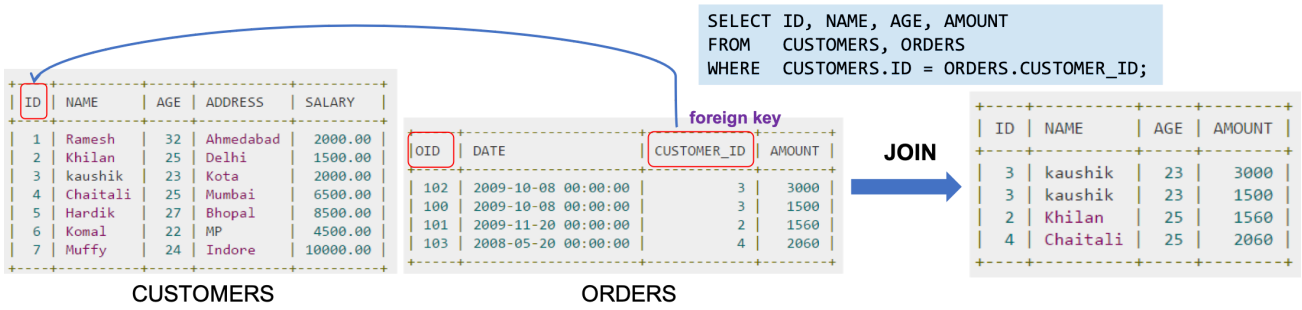
    - RDBMS는 객체지향 프로그래밍 언어의 자연적 패러다임인 상속과 유사한 것을 정의하지 않는다.
    - 즉, 상속의 개념이 없다.
- Identity(일치)
    - RDBMS는 'sameness'라는 하나의 개념을 정확히 정의하는데, 바로 '기본키(primary key)'이다.
    - 그러나 자바에서는 객체 식별(a==b)과 객체 동일성(a.equales(b))을 모두 정의한다.
    - RDBMS에서는 PK가 같으면 서로 동일한 record로 정의하지만, Java에서는 주솟값이 같거나 내용이 같은 경우를 구분하여 정의한다.
- Associations(연관성)
    - 객체지향 언어는 객체 참조(reference)를 사용하는 연관성을 나타내는 반면, RDBMS는 연관성을 '외래키(foreign key)'로 나타낸다.
    - ***아래 참고***
- Navigation(탐색/순회)
    - Java 및 RDBMS에서 객체에 액세스하는 방법은 근본적으로 다르다.
    - Java에서는 하나의 연결에서 다른 연결로 이동하면서 탐색/순회한다. (그래프 형태)
        - 예를 들어, aUser.getBillingDetails().getAccountNumber()
        - 이는 RDBMS에서 데이터를 검색하는 효율적인 방법이 아니다.
    - RDBMS에서는 일반적으로 SQL 쿼리 수를 최소화하고 JOIN을 통해 여러 엔터티를 로드하고 원하는 대상 엔터티를 선택(select)한다.

Association(연관성)

- Java에서의 객체 참조(Object References)
    - 방향성이 있다. (Directional)

```
public class Employee {
    private int id;
    private String first_name;
    …
    private Department department; // Employee -> Department
    …
}
```

    - Java에서 양방향 관계가 필요한 경우 연관을 두 번 정의해야 한다.
        - 즉, 서로 Reference를 가지고 있어야 한다.
- RDBMS의 외래키(Foreign Key)
    - FK와 테이블 Join은 관계형 데이터베이스 연결을 자연스럽게 만든다.
    -



ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

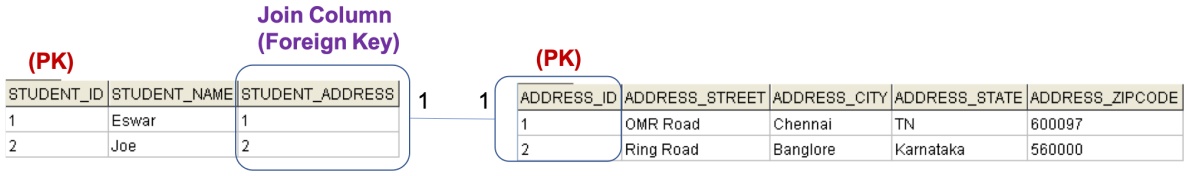
OID	DATE	CUSTOMER_ID	AMOUNT
102	2009-10-08 00:00:00	3	3000
100	2009-10-08 00:00:00	3	1500
101	2009-11-20 00:00:00	2	1560
103	2008-05-20 00:00:00	4	2060

ID	NAME	AGE	AMOUNT
3	kaushik	23	3000
3	kaushik	23	1500
2	Khilan	25	1560
4	Chaitali	25	2060

- 방향성이 없다. (Direction-Less)

```
INSERT INTO
EMPLOYEE(id, first_name, … ,department_id) // FK
VALUES …
```

1. **One-To-One Relationship**
   - 예를 들어, 각 학생은 고유한 주소를 가지고 있다고 하자.
   - RDBMS (방향성이 없다.)



STUDENT_ID	STUDENT_NAME	STUDENT_ADDRESS
1	Eswar	1
2	Joe	2

ADDRESS_ID	ADDRESS_STREET	ADDRESS_CITY	ADDRESS_STATE	ADDRESS_ZIPCODE
1	OMR Road	Chennai	TN	600097
2	Ring Road	Banglore	Karnataka	560000

     - 각 Student의 record는 서로 다른 Address record를 가리키고 이것은 일대일 매핑을 보여준다.
   - Java Object (방향성이 있다.)

```
public class Student {
 private long studentId;
 private String studentName;
 private Address studentAddress; // Student -> Address
 …
}
public class Address {
 private long addressId;
 private String street;
 private String city;
 private String state;
 private String zipcode;
 …
}
```

2. **One-To-Many Relationship**
    - 예를 들어, 각 학생은 여러 개의 핸드폰을 가질 수 있다고 하자.
    - RDBMS (방향성이 없다.)
        - 각 Student의 record는 여러 개의 Phone record를 가리킬 수 있다. (일대다 매핑)
        - 이 관계를 하나의 다른 Table(Relational Model)로 만들 수 있다.
        - One-To-Many를 구성하는 방법: 1) Join Table, 2) Join Column
    - Java Object (방향성이 있다.)

```
public class Student {
 private long studentId;
 private String studentName;
 private Set<Phone> studentPhoneNumbers; // Student -> Some Phones
 …
}
public class Phone {
 private long phoneId;
 private String phoneType;
 private String phoneNumber;
 …
}
```

관련된 Post

- JDBC, JPA/Hibernate, Mybatis의 차이에 대해 알고 싶으시면 JDBC, JPA/Hibernate, Mybatis의 차이를 참고하시기 바랍니다.
- Spring Hibernate에 대해 알고 싶으시면 Spring Hibernate 이해하기를 참고하시기 바랍니다.

References

- ORM의 장단점