MyBatis 3

❑ **About**

- 본 문서는 MyBatis 사용 가이드를 위해 작성되었음.
- MyBatis는 Apache 재단의 iBatis 라이선스를 google이 인수하면서 버전업되었음.
- iBatis 는 Persistence framework, ORM tool, Sql Mapper, Data Mapper 등으로 불리나 본 문서에서는 Data Mapper 로 통칭하겠음.
- 본 문서는 myBatis 에 익숙치않은 초급자용 교재로 작성됐으며, 자세한 사항은 메뉴얼을 참고하기 바람.

❑ **목차**

❑ **참고**

- http://mybatis.github.io/mybatis-3/ko/(myBatis 메뉴얼)

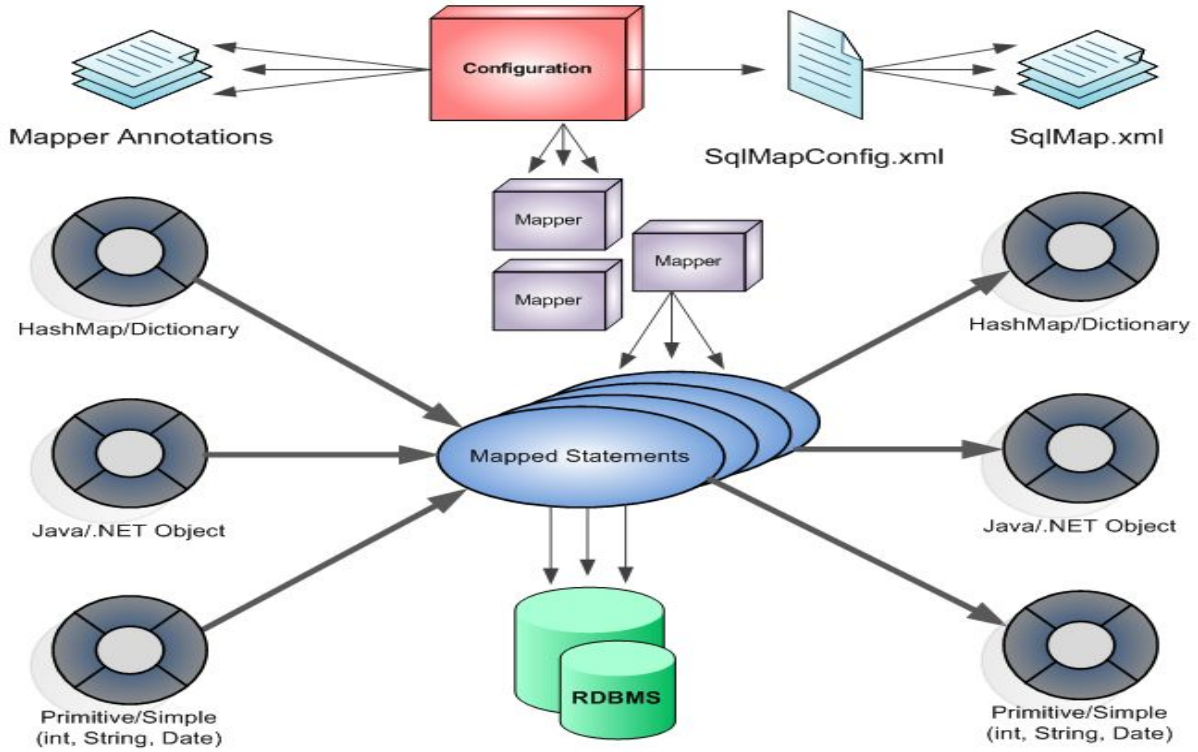
대덕인재개발원 : 최희연

❑ **MyBatis Intro & Architecture (**Mybatis란?)

- Intro
  - myBatis Data Mapper Framework는 JDBC 프로그래밍 코드를 현저하게 줄여줌.
  - myBatis 는 다른 프레임워크, ORM 에 비해 쉽고 가벼움.
  - XML기술을 사용해 Sql 과 자바객체간ㅊ의 매핑을 걸어줌.
  - PreparedStatement ←→ 자바객체
  - ResultSet ←→ 자바객체 상호간 데이터 변환을 지원해줌.
- History
  - 개발자인 Clinton Begin에 의해 java기반의 iBatis 프로젝트 시작.
  - 2002년 ASF(Apache software Foundation) Java 기반 1.0 발표 .Net 기반 프로젝트 시작
  - 2004년 ASF의 Top Level project로 승인 .Net 기반 1.0 발표
  - 2010년 6월 Google Code 로 프로젝트 이전 MyBatis로 개칭
- 특징
  - 추상화된 접근 방식 제공
    JDBC 프로그래밍에 추상화된 접근 방식을 적용 간편하고, 쉬운 API, 자원 연결/해제, 공통 에러 처리 등을 통합 지원함.
  - 코드로부터 SQL 분리 지원
    소스코드로부터 SQL문을 분리하여 별도의 repository(의미있는 문법의 XML)에 유지하고 이에 대한 빠른 참조 구조를 내부적으로 구현하여 관리/유지보수/튜닝의 용이성을 보장함.
  - 쿼리 실행의 입/출력 객체 바인딩/매핑 지원
    쿼리문의 입력파라미터에 대한 바인딩과 실행결과 resultset의 가공(매핑)처리시 객체(Vo, Map, List)수준의 자동화를 지원함.
  - Dynamic Sql 지원
    코드작성, API 직접 사용 없이 입력 조건에 따른 동적인 쿼리문 변경을 지원함.
  - 다양한 DB 처리 지원
    기본 질의 외에 Batch SQL, Paging, Callable Statement, BLOB/CLOB처리를 지원함.
- iBatis 와의 주요 차이점

iBatis	MyBatis
쿼리 등록 및 관리를 위한 설정에 XML 우선	매퍼 설정에 XML 과 어노테이션 모두 지원
namespace의 의미가 상대적으로 단순	namespace를 통해 매퍼객체가 생성되므로 namespace 선언이 필수사항이며 반드시 매퍼 인터페이스의 qualified name사용
SqlMapClient의 질의 실행 API에 의존	매퍼 인터페이스의 메소드를 직접 사용하므로 type safety 보장
thread safety 한 SqlMapClient 사용	메소드 스코프의 SqlSession 사용
inline parameter : #param#, replaceText : $text$	inline parameter : #{param}, replaceText : ${text}
dynamic 엘리먼트를 이용한 동적 쿼리	if, where 등의 직관적 엘리먼트 사용

- 아키텍쳐 및 구성요소



Architecture 구 성 요 소	설 명
설정파일 (SqlMapConfig.xml)	DataSource, Data Mapper, Transaction Manager 등 데이터베이스 사용을 위한 기본 설정들을 가진 설정 파일이다.
매퍼 인터페이스	SQL을 정의한 XML이나 어노테이션으로 정의된 SQL을 가진 인터페이스로 이를 통해 매퍼 객체가 생성된다.
결과 매핑과 매핑 구문	질의 실행 결과 집합을 자바 객체로 매핑하기 위한 규칙을 정의한 XML이나 어노테이션 설정을 의미한다.
매핑된 구문맵	XML이나 어노테이션으로 등록된 질의문이 관리되는 맵
파라미터	질의 실행을 위해 필요한 파라미터 데이터로 일반적인 자바객체 형태로 표현되면 myBatis에 설정된 타입핸들러에 의해 데이터베이스 데이터타입의 파라미터로 전환된다.
결과 객체	Cursor 형태의 결과집합으로 조회된 결과가 myBatis의 타입핸들러에 의해 자바 객체 형태로 전환된다.

- 역할

어플리케이션과 관계형 데이터베이스간에 데이터를 송수신할 수 있는 연결 객체 생성 및 관리와 양쪽에서 서로 다른 형태로 표현되는 데이터를 변환하기 위한 절차가 생략될 수 있도록 고도의 추상화가 제공되는 프레임워크.

- 쿼리를 실행할 수 있도록 연결 설정을 위한 기반 코드의 제공.
- 등록된 쿼리를 유지하기 위한 쿼리 맵의 관리
- 쿼리를 실행하기 위해 어플리케이션에서 넘기는 파라미터의 바인딩
- 쿼리의 실행
- 쿼리를 실행한 결과 객체를 어플리케이션으로 넘기기 위한 결과 객체 매핑

❑ MyBatis 시작하기

1) 이클립스에 Mybatis 플러그인 설치 : 이클립스 마켓에서 Mybatipse 설치

2) Mybatis 관련 의존성 추가 : 메이븐 dependency스크립트를 추가하거나,
http://blog.mybatis.org/p/products.html 에서 다운로드

```
<dependency>
	<groupId>org.mybatis</groupId>
	<artifactId>mybatis</artifactId>
	<version>3.2.8</version>
</dependency>
```

3) 결과 매핑 객체 작성

```
public class DataBasePropertiesVO {
	private String property_name;
	private String property_value;
	private String description;
	// getter/setter..
}
```

4) 매퍼 XML 혹은 매퍼 인터페이스 작성

① XML config

kr.or.ddit.chap18.ICommonDAO

```
public interface ICommonDAO {
	 public List<DataBasePropertiesVO> retrieveProps();
}
```

kr/or/ddit/mybatis/mappers/commons.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace="kr.or.ddit.chap18.ICommonDAO">
	<select id="retrieveProps" resultType="dbPropertiesVO">
		SELECT PROPERTY_NAME, PROPERTY_VALUE, DESCRIPTION
		FROM DATABASE_PROPERTIES
	</select>
</mapper>
```

② Java config

```
public interface ICommonDAO {
	@Select({
		"SELECT PROPERTY_NAME, PROPERTY_VALUE, DESCRIPTION",
		"FROM DATABASE_PROPERTIES" })
	public List<DataBasePropertiesVO> retrieveProps(String textParam);
}
```

Namespace 는 iBatis 까지는 선택사항이었지만, Mybatis 에서는 각 구문 그룹을 구분할 수 있는 식별자로서 namespace가 필수 설정사항이다. 그런데 단순히 그룹 식별자 역할 보다 더 중요한 기능은, Mybatis가 매퍼 인터페이스를 바인딩하고, 매퍼 프록시 객체를 생성하기 위한 조건으로 바로 이 namespace가 사용된다는 것이다. 때문에 단지 관행적으로 뿐만아니라 분석의 용이성을 위해 XMLConfig 나 Java Config에서 모두 패키지 경로를 포함한 매퍼 인터페이스의 qualified name을 namespace로 사용하는 것이 일반적이다.

5) 매퍼 등록 설정 및 SqlSessionFactory 빌드하기

① XML Config

kr/or/ddit/mybatis/SqlMapConfig.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE configuration PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
"http://mybatis.org/dtd/mybatis-3-config.dtd" >
<configuration>
	<properties resource="kr/or/ddit/database_info.properties" />
	<typeAliases>
		<typeAlias type="kr.or.ddit.chap18.DataBasePropertiesVO"
			alias="dbPropertiesVO"/>
	</typeAliases>
	<environments default="development">
		<environment id="development">
			<transactionManager type="JDBC" />
			<dataSource type="POOLED">
				<property name="driver" value="${maindb.driverClassName}"/>
				<property name="url" value="${maindb.url}"/>
				<property name="username" value="${maindb.username}"/>
				<property name="password" value="${maindb.password}"/>
				<property name="poolMaximumActiveConnections"
					value="${maindb.maxActive}"/>
				<property name="poolTimeToWait" value="${maindb.maxWait}"/>
			</dataSource>
		</environment>
	</environments>
	<mappers>
		<mapper resource="kr/or/ddit/mybatis/mappers/commons.xml"/>
		<!-- <mapper class="kr.or.ddit.chap18.ICommonDAO" /> -->
	</mappers>
</configuration>
```

kr.or.ddit.mybatis.CustomSqlSessionFactoryBuilderXMLConfig

```
public class CustomSqlSessionFactoryBuilderXMLConfig {
	private static SqlSessionFactory sessionFactory;
	static{
		Reader reader;
		try {
			reader = Resources.getResourceAsReader
					("kr/or/ddit/mybatis/SqlMapConfig.xml");
			sessionFactory = new SqlSessionFactoryBuilder()
					.build(reader, "development");
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
	public static SqlSessionFactory getSessionFactory() {
		return sessionFactory;
	}
}
```

② Java Config

kr.or.ddit.mybatis.CustomSqlSessionFactoryBuilderJavaConfig

```
public class CustomSqlSessionFactoryBuilderJavaConfig {
	private static SqlSessionFactory sessionFactory;
	static{
		ResourceBundle databaseInfo = ResourceBundle
					.getBundle("kr.or.ddit.database_info");
		// driver, url, username, password, maxActive, maxWait 등의 확보
		TransactionFactory transactionFactory = new JdbcTransactionFactory();
		PooledDataSource dataSource =
				new PooledDataSource(driver, url, username, password);
		dataSource.setPoolMaximumActiveConnections(poolMaximumActiveConnections);
		dataSource.setPoolTimeToWait(poolTimeToWait);
		Environment environment =
				new Environment("development",
					transactionFactory, dataSource);
		Configuration configuration = new Configuration(environment);
		configuration.addMapper(ICommonDAO.class);
		sessionFactory = new SqlSessionFactoryBuilder().build(configuration);
	}
	public static SqlSessionFactory getSessionFactory() {
		return sessionFactory;
	}
}
```

6) SqlSession을 통한 데이터 조회

```
SqlSessionFactory factory =  CustomSqlSessionFactoryBuilderXMLConfig.getSessionFactory();
SqlSession sqlSession = factory.openSession();
try{
	// iBatis 에서 주로 사용하던 방식으로 type 안정성이 보장되지 않고, 오타에 취약함.
	// List<DataBasePropertiesVO> propsList
	//= sqlSession.selectList("kr.or.ddit.chap18.ICommonDAO.retrieveProps");
	// Mybatis 에서 주로 사용하는 방식으로, 매퍼 인터페이스를 사용하는 이유.
	ICommonDAO commonDAO = sqlSession.getMapper(ICommonDAO.class);
	List<DataBasePropertiesVO> propsList = commonDAO.retrieveProps("ss");
	for(DataBasePropertiesVO prop : propsList){
		// prop 에 대한 개별 처리
	}
}finally {
	sqlSession.close();
}
```

- Mybatis에 구문을 등록하고, Factory를 통해 SqlSession을 사용하는 주의할 부분이 Scope와 LifeCycle 부분이다.
  - SqlSessionFactoryBuiilder : SqlSessionFactory 객체 생성 이후 유지할 필요가 없어 메소드 스코프 사용한다.
  - SqlSessionFacotry : 어플리케이션 스코프로 유지하고, 삭제나 재생성은 필요없다. 데이터베이스 서버가 여러 개라면 하나의 서버당 하나의 SqlSessionFactory를 생성한다.
  - SqlSession 및 매퍼 인스턴스 : 각 쓰레드별로 자체적인 SqlSession 인스턴스를 가져야하고, SqlSession 인스턴스는 서로 공유되지 않아야하기 때문에 쓰레드에 안전하지도 않다. 따라서 가장 좋은 스코프는 메소드 스코프라 할 수 있고, 절대 정적 필드나 전역필드로 설정하면 안된다. 또는 서블릿의 HttpSession과 같은 스코프에서 관리하는 것도 위험하다. 어떤 종류의 웹 프레임워크를 사용한다면 Http 요청과 유사한 스코프에 두는 것을 고려해야 한다. 다시 말해 요청이 들어올 때 만들고 응답을 리턴하기 전에 SqlSession을 닫는 방법을 고려해야 하고, 언제나 finally 블록에서 close를 해야지만, 정상적으로 커넥션이 풀에 반환될수 있다.

❑ MyBatis 매퍼 설정(SqlMapConfig.xml)

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration   PUBLIC "-//mybatis.org//DTD Config 3.1//EN“
  "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
	<properties resource="" />
	<settings>
		<setting name="" value="" />
	</settings>
	<typeAliases>
		<typeAlias type="" alias="" />
	</typeAliases>
	<typeHandlers>
		<typeHandler handler="" javaType="" />
	</typeHandlers>
	<objectFactory type="“>
		<property name="" value="" />
	</objectFactory>
	<plugins>
		<plugin interceptor="">
			<property name="" value="" />
		</plugin>
	</plugins>
	<environments default="">
		<environment id="">
		<transactionManager type="" />
		<dataSource type="">
			<property name="" value="" />
		</dataSource>
		</environment>
	</environments>
	<mappers>
		<mapper resource="" />
		<mapper class="" />
		<package name="" />
	</mappers>
</configuration>
```

properties	외부 설정을 로딩하거나 전역으로 접근할 프로퍼티를 설정하기 위한 요소
settings	런타임에 Mybatis의 행위를 조정하기 위한 설정값들로 SqlSession 관련 설정 최적화를 위한 요소들로 구성되며, 모든 요소는 optional 설정
typeAliases	파라미터나 결과객체의 class qualified name 대신 사용할 수 있는 별칭 집합
typeHandlers	파라미터 바인딩이나 결과 객체 매핑시 타입 변환에 사용될 타입 핸들러 집합, 기본으로 제공되는 타입핸들러 이외의 커스텀 핸들러 등록에 사용
objectFactory	구문 실행 결과를 매핑할 결과객체를 생성하기 위한 factory 클래스 지정 결과 객체를 매핑할 타입에 파라미터가 있는 생성자를 사용하는 경우 설정
plugins	매핑 구문 실행의 특정 시점에 호출을 가로채기 위한 플러그인으로 Mybatis 기반 클래스나 메소드를 수정할 때 사용
environments	transactionManager 나 dataSource 등에 대한 설정을 가진 environment 들의 집합
databaseIdprovider	DatabaseMetaData#getDatabaseProductName() 이 돌려주는 db 서버 별칭 등록시 사용
mappers	구문이 작성된 매퍼xml 이나 어노테이션으로 구문이 작성된 매퍼 인터페이스의 등록에 사용

- settings 의 주요 설정 종류(Mybatis Settings 문서를 참고)

cacheEnabled	기본 true, 캐시 사용 여부, 각 매퍼별로 캐시 사용여부를 제어할때는 false로 셋팅.
lazyLoadingEnabled	기본 false, 성능 개선을 위한 Lazy Loading 사용 여부. 조회시 데이터를 한꺼번에 가져오지 않고, 필요한 시점에 가져오도록 동작하므로 시스템의 부하를 분산시켜 성능의 향상을 꾀할 수 있다. 특히 기본키를 기준으로 데이터를 가져오는 경우 lazy Loading 이 아닌 것처럼 동작해 빠른 처리가 가능하다.
aggressiveLazyLoading	기본 true, lazy loading 객체는 호출에 따라 점진적으로 요청하고, 일반 객체는 요청시 한번에 모두 로드된다.
defaultStatementTimeout	데이터베이스 요청을 처리하는 도중 처리 지연이 발생시 자동으로 중지하도록 하기 위한 지연시간 설정. 기본값은 JDBC 드라이버에 설정된 값을 따른다.
mapUnderscoreToCamelCase	기본값 false, 데이터베이스 테이블 컬럼명에 사용되는 언더바(_)를 자바의 코딩 규칙인 카멜표기법으로 고쳐서 표현할지 여부 ex) co_author => coAuthor
localCacheScope	기본 SESSION, 캐시데이터 저장 범위 설정 SESSION : SqlSession을 기준으로 캐시 저장 STATEMENT : 구문별로 캐시 저장, SqlSession에서 두 개의 다른 호출 사이 데이터 공유는 불가능하다.
useGeneratedKeys	기본값 false, 생성키 사용여부, MySQL의 auto_increment, Oracle의 Sequence, SQL Server의 identify등이 생성키로 제공된다.

- environments

```
<environments default="development">
  <environment id="development">
    <transactionManager type="JDBC">
      <property name="..." value="..."/>
    </transactionManager>
    <dataSource type="POOLED">
      <property name="driver" value="${driver}"/>
      <property name="url" value="${url}"/>
      <property name="username" value="${username}"/>
      <property name="password" value="${password}"/>
    </dataSource>
  </environment>
</environments>
```

  - default 속성 : SqlSessionFactory 객체를 생성하는 과정에서 등록된 여러개의 environment 들 중 특정 하나가 지정되지 않았다면 기본적으로 사용될 environment의 ID.

```
sessionFactory = new SqlSessionFactoryBuilder().build(reader, "development");
```

  - environment의 하위요소 transactionManager 타입
    a) JDBC : JDBC 커밋과 롤백을 직접 처리하기 위해 사용되며, 트랜잭션의 스코프를 관리하기 위해 dataSource로부터 커넥션을 가져온다.
    b) MANAGED : 커밋이나 롤백을 직접 처리하지 않고, 컨테이너가 트랜잭션의 모든 생명주기를 관리하는 경우 사용된다. 참고로 스프링의 경우는 어떠한 트랜잭션관리자를 설정하더라도 스프링 모듈 자체를 구성하기 때문에 설정이 무시된다.
  - environment의 하위요소 dataSource 타입
    a) UNPOOLED : 매번 요청이 발생하는 순간 커넥션을 열고 닫는 Simple DataSource로 ibatis 의 "SIMPLE"과 유사한 타입
    b) POOLED : DataSource에 풀링이 적용된 JDBC 커넥션을 위한 구현체(PooledDataSource)로, 새로운 Connection 인스턴스를 생성하기 위해 매번 초기화하는 것을 피할 수 있기 때문에 빠른 응답속도를 요하는 웹 어플리케이션에서 가장 흔히 사용된다.
    c) JNDI : 컨테이너가 JNDI 컨텍스트에 등록한 데이터베이스 풀링 객체를 lookup을 통해 사용하기 위한 타입으로 풀링 데이터소스 객체의 JNDI 등록에 관한 자세한 설명은 톰캣7 JNDI 문서를 참고하기 바란다.

d) dataSource 기본 속성들

driver	JDBC 드라이버의 패키지 경로를 포함한 자바 클래스명
url	데이터베이스 인스턴스에 대한 JDBC URL
username/password	데이터베이스 접속 계정
defaultTransactionIsolationLevel	커넥션에 대한 기본 트랜잭션 격리 레벨
"driver." prefix	선택적인 프로퍼티 설정. ex) driver.encoding=UTF8

- mappers 등록
  - 매퍼 XML 인 경우.

```
<!-- Using classpath relative resources -->
<mappers>
  <mapper resource="org/mybatis/builder/AuthorMapper.xml"/>
  <mapper resource="org/mybatis/builder/BlogMapper.xml"/>
  <mapper resource="org/mybatis/builder/PostMapper.xml"/>
</mappers>
```

  - 매퍼 인터페이스에 어노테이션을 사용하는 경우.

```
<!-- Using mapper interface classes -->
<mappers>
  <mapper class="org.mybatis.builder.AuthorMapper"/>
  <mapper class="org.mybatis.builder.BlogMapper"/>
  <mapper class="org.mybatis.builder.PostMapper"/>
</mappers>
```

  - 특정 패키지 내의 모든 클래스를 대상으로 타입별칭이나 매퍼 인터페이스로 등록하는 경우

```
<typeAliases>
  <package name="domain.blog"/>
</typeAliases>
```

```
@Alias("author")
public class Author {
    ...
}
```

```
<!-- Register all interfaces in a package as mappers -->
<mappers>
  <package name="org.mybatis.builder"/>
</mappers>
```

CoC 에 충실한 프레임워크들의 경향에 따라 Mybatis도 다양한 방식으로 CoC를 지원하는데, 이를 통해 설정을 간소화할 수 도 있다. 예를 들어, 특정 패키지의 빈들에 대해 축약형 별칭을 자동등록하고 싶거나 특정 패키지의 인터페이스를 매퍼로 자동 등록할 때는 상위 그림과 같은 형태의 설정을 통해 가능하다. 타입별칭이 등록될 때는 CoC에 따라 타입명의 첫문자가 소문자로 바뀌어 등록되는데, 이를 명시적으로 바꾸고 싶다면, @Alias 어노테이션을 사용하면 가능하다. Mybatis는 이외에도 다양한 프로퍼티들을 사용하여 설정 외부화나 타입별칭 지정 혹은 타입변환이나 런타임시 수행성능 향상을 위한 캐시 설정 등 다양한 환경들을 제어할 수 있으니 자세한 사항은 Mybatis configuration 문서를 통해 확인하기 바란다.

❑ MyBatis 매퍼 XML, 매퍼 인터페이스 작성

- 매퍼 XML 을 이용한 매핑 구문 작성
  - select

```
<select id="selectPerson" parameterType="int" resultType="hashmap">
  SELECT * FROM PERSON WHERE ID = #{id}
</select>
```

id	구문을 찾기 위해 사용될 수 있는 namespace 내의 유일한 식별자, 일반적으로 해당 구문을 사용할 매퍼 인터페이스의 메소드명과 일치시킨다.
parameterType	구문에 전달될 파라미터의 패키지 경로를 포함한 전체 클래스명이나 별칭
resultType	결과 객체 매핑 타입별칭이나 클래스명.
resultMap	외부 resultMap 의 참조명. 조회구문에는 반드시 결과 데이터 매핑을 위한 resultType 이나 resultMap이 필요하다.
flushCache	구문이 호출시마다 캐시를 지울지(flush) 여부. 디폴트는 false 이다.
useCache	구문의 결과가 캐싱 여부. 디폴트는 true 이다.
timeout	데이터베이스 요청이 발생하고 예외가 던져지기 전까지 최대 대기 시간으로, 디폴트는 셋팅하지 않는 것이고 드라이버에 따라 지원되지 않을 수 있다.
fetchSize	지정된 수만큼의 결과를 리턴하도록 하는 드라이버 힌트 형태의 값이다. 디폴트는 셋팅하지 않는 것이고 드라이버에 따라 지원되지 않을 수 있다.
statementType	구문 객체의 종류 설정(STATEMENT, PREPARED, CALLABLE), 디폴트는 PREPARED

```
<insert id="insertAuthor">
  insert into Author (id,username,password,email,bio)
  values (#{id},#{username},#{password},#{email},#{bio})
</insert>

<update id="updateAuthor">
  update Author set
    username = #{username},
    password = #{password},
    email = #{email},
    bio = #{bio}
  where id = #{id}
</update>

<delete id="deleteAuthor">
  delete from Author where id = #{id}
</delete>
```

useGeneratedKeys	(입력(insert, update)에만 적용) 데이터베이스에서 내부적으로 생성한 키를 받는 JDBC getGeneratedKeys 메서드를 사용하도록 설정하다. 디폴트는 false 이다.
keyProperty	(입력(insert, update)에만 적용) getGeneratedKeys 메서드나 insert 구문의 selectKey 하위 요소에 의해 리턴된 키를 셋팅할 프로퍼티를 지정. 디폴트는 셋팅하지 않는 것이다.
keyColumn	(입력(insert, update)에만 적용) 생성키를 가진 테이블의 칼럼명을 셋팅. 키 칼럼이 테이블이 첫 번째 칼럼이 아닌 데이터베이스(PostgreSQL 처럼)에서만 필요하다.

- selectKey – 자동 키 생성을 지원하지 않는 데이터베이스를 위한 대안으로 제공되면, insert 구문에서 사용됨.

```
<insert id="insertAuthor">
  <selectKey keyProperty="id" resultType="int" order="BEFORE">
    select CAST(RANDOM()*1000000 as INTEGER) a from SYSIBM.SYSDUMMY1
  </selectKey>
  insert into Author
    (id, username, password, email,bio, favourite_section)
  values
    (#{id}, #{username}, #{password}, #{email}, #{bio}, #{favouriteSection,jdbcType=VARCHAR})
</insert>
```

keyProperty	selectKey 구문의 결과가 셋팅될 대상 프로퍼티.
keyColumn	리턴되는 결과셋의 칼럼명은 프로퍼티에 일치한다. 여러개의 칼럼을 사용한다면 칼럼명의 목록은 콤마를 사용해서 구분한다.
resultType	결과의 타입. MyBatis 는 이 기능을 제거할 수 있지만 추가하는게 문제가 되지는 않을것이다. MyBatis 는 String 을 포함하여 키로 사용될 수 있는 간단한 타입을 허용한다.
order	BEFORE 또는 AFTER 를 셋팅할 수 있다. BEFORE 로 셋팅하면, 키를 먼저 조회하고 그 값을 keyProperty 에 셋팅한 뒤 insert 구문을 실행한다. AFTER 로 셋팅하면, insert 구문을 실행한 뒤 selectKey 구문을 실행한다. Oracle 과 같은 데이터베이스에서는 insert 구문 내부에서 일관된 호출 형태로 처리한다.
statementType	구문객체의 종류.

위의 예에서 selectKey 구문이 먼저 실행되고, Author.id에 그 값이 셋팅된 다음, insert 구문이 실행되기 때문에, 자동키 생성 컬럼을 지원하지 않는 데이터베이스에서도 비슷한 효과를 볼수 있다.

- sql : 다른 구문에서 재사용 가능한 SQL 구문을 정의할 때 사용.

```
<sql id="userColumns"> id,username,password </sql>
```

```
<select id="selectUsers" resultType="map">
  select <include refid="userColumns"/>
  from some_table
  where id = #{id}
</select>
```

- 인라인 파라미터와 대체 구문

  특정 컬럼에 null 값이 전달되는 경우라면, jdbcType과 javaType 에 대한 기술이 반드시 필요하다 (PreparedStatement.setNull 참고).

```
#{property,javaType=int,jdbcType=NUMERIC}
```

enum으로 지원되는 JDBC 타입

BIT	FLOAT	CHAR	TIMESTAMP	OTHER	UNDEFINED
TINYINT	REAL	VARCHAR	BINARY	BLOG	NVARCHAR
SMALLINT	DOUBLE	LONGVARCHAR	VARBINARY	CLOB	NCHAR
INTEGER	NUMERIC	DATE	LONGVARBINARY	BOOLEAN	NCLOB
BIGINT	DECIMAL	TIME	NULL	CURSOR	ARRAY

동적 구문 작성을 위한 대체 구문 형식 : Sql Injection 에 취약하므로 지양하는 것이 좋다.

```
ORDER BY ${columnName}
```

- Result Maps

  결과 객체 매핑에 자바빈이나 POJO 가 사용되는 경우, 컬럼명과 프로퍼티명이 같으면 자동 매핑이라는 규칙성이 적용되는데, 이는 Mybatis가 내부적으로 자바빈의 프로퍼티명을 기준으로 ResultMap을 생성하여 매핑 처리를 하는 것이다. 이런 매핑 처리에 다른 규칙성을 적용하고 싶을때, column alias 를 사용하거나 커스텀

```
<select id="selectUsers" resultType="User">
  select
    user_id             as "id",
    user_name           as "userName",
    hashed_password     as "hashedPassword"
  from some_table
  where id = #{id}
</select>
```

```
<resultMap id="userResultMap" type="User">
  <id property="id" column="user_id" />
  <result property="username" column="username"/>
  <result property="password" column="password"/>
</resultMap>
```

```
<select id="selectUsers" resultMap="userResultMap">
  select user_id, user_name, hashed_password
  from some_table
  where id = #{id}
</select>
```

1) id, result : 결과 매핑의 가장 기본적인 형태로, 둘다 모두 하나의 컬럼을 하나의 프로퍼티에 매핑하기 위한 설정이며, id는기본키 매핑에 별도의 엘리먼트를 사용함으로써 객체 인스턴스 비교에 사용하여 성능을 향상을 꾀하기 위해 사용한다.

property	컬럼 매핑을 위한 결과 객체의 프로퍼티 명
column	데이터베이스의 칼럼명이나 칼럼 라벨.
javaType	패키지 경로를 포함한 클래스 전체명이거나 타입 별칭.
jdbcType	지원되는 타입 목록에서 설명하는 JDBC 타입. JDBC 타입은 insert, update 또는 delete 하는 null 입력이 가능한 칼럼에서만 필요하다. JDBC 의 요구사항이지 MyBatis 의 요구사항이 아니다.
typeHandler	이 프로퍼티를 사용하면, 디폴트 타입 핸들러를 오버라이드 할 수 있다. 이 값은 TypeHandler 구현체의 패키지를 포함한 전체 클래스명이나 타입 별칭이다.

2) consutructor
   setter 대신 생성자 주입으로 컬럼 데이터를 결과 객체에 매핑하는 경우에 사용하며, idArg는 생성자 호출 파라미터 중 키 역할을 하는 데이터를 주입할 때 사용하는 엘리먼트이다.

```
public class User {
   //...
   public User(int id, String username) {
     //...
  }
//...
}
```

```
<constructor>
   <idArg column="id" javaType="int"/>
   <arg column="username" javaType="String"/>
</constructor>
```

3) association : has-one 관계 매핑을 위한 resultMap에 사용하는 엘리먼트
결과 집합을 매핑할 객체와 객체간의 관계 혹은 join 의 대상이 되는 테이블과 테이블간의 관계를 형성하여 직접조인 구문이나 Mybatis 설정을 통한 다중 테이블 조회 구문의 결과집합을 객체에 매핑하기 위한 resultMap 설정에 사용된다. 그중 has-one 관계에 사용되는 설정이 association, has-many 관계에 사용되는 설정이 collection 엘리먼트이다.
다중 테이블 조회 결과를 결과객체로 매핑하는 두가지 방식이 지원된다.

① nested select

```
<resultMap id="blogResult" type="Blog">
  <association property="author" column="author_id" javaType="Author" select="selectAuthor"/>
</resultMap>

<select id="selectBlog" resultMap="blogResult">
  SELECT * FROM BLOG WHERE ID = #{id}
</select>
```

```
<select id="selectAuthor" resultType="Author">
  SELECT * FROM AUTHOR WHERE ID = #{id}
</select>
```

이 방법은 간단한 반면, "N+1 Selects 문제" 로 알려진 문제점을 야기할 수 있다. N+1 Selects 문제는 처리과정의 특이성으로 인해 야기된다.
- 레코드의 목록을 가져오기 위해 하나의 SQL 구문을 실행한다. ("+1").
- 리턴된 레코드별로, 각각의 상세 데이터를 로드하기 위해 select 구문을 실행한다. ("N").
이 문제는 수백 또는 수천의 SQL 구문 실행이라는 결과를 야기할 수 있다. 목록을 로드하고 내포된 데이터에 접근하기 위해 즉시 반복적으로 처리한다면, settings에서 lazyLoadingEnabled를 true 로 설정하고 사용하더라도 one 에 해당하는 객체를 대상으로 proxy 객체가 만들어지기 때문에 많은 데이터가 조회되는 경우, 성능이 떨어질 수 밖에 없다.

② nested results

```
<select id="selectBlog" resultMap="blogResult">
  select
    B.id            as blog_id,
    B.title         as blog_title,
    B.author_id     as blog_author_id,
    A.id            as author_id,
    A.username      as author_username,
    A.password      as author_password,
    A.email         as author_email,
    A.bio           as author_bio
  from Blog B left outer join Author A on B.author_id = A.id
  where B.id = #{id}
</select>
```

이 방법은 BLOG 테이블과 AUTHOR 테이블의 데이터를 하나의 구문으로 조회하고 있기 때문에 association 프로퍼티를 채우기 위해 별도의 select 구문을 사용할 필요가 없어진다.

```
<resultMap id="blogResult" type="Blog">
  <id property="id" column="blog_id" />
  <result property="title" column="blog_title"/>
  <association property="author" column="blog_author_id" javaType="Author" resultMap="authorResult"/>
</resultMap>
```

```
<resultMap id="authorResult" type="Author">
  <id property="id" column="author_id"/>
  <result property="username" column="author_username"/>
  <result property="password" column="author_password"/>
  <result property="email" column="author_email"/>
  <result property="bio" column="author_bio"/>
</resultMap>
```

4) collection : has-many 관계 정의에 사용

① nested select

```
<resultMap id="blogResult" type="Blog">
  <collection property="posts" column="id" ofType="Post" select="selectPostsForBlog"/>
</resultMap>

<select id="selectBlog" resultMap="blogResult">
  SELECT * FROM BLOG WHERE ID = #{id}
</select>

<select id="selectPostsForBlog" resultType="Blog">
  SELECT * FROM POST WHERE BLOG_ID = #{id}
</select>
```

② nested results

```
<select id="selectBlog" resultMap="blogResult">
  select
  B.id as blog_id,
  B.title as blog_title,
  B.author_id as blog_author_id,
  P.id as post_id,
  P.subject as post_subject,
  P.body as post_body,
  from Blog B
  left outer join Post P on B.id = P.blog_id
  where B.id = #{id}
</select>
```

```
<resultMap id="blogResult" type="Blog">
  <id property="id" column="blog_id" />
  <result property="title" column="blog_title"/>
  <collection property="posts" ofType="Post">
    <id property="id" column="post_id"/>
    <result property="subject" column="post_subject"/>
    <result property="body" column="post_body"/>
  </collection>
</resultMap>
```

5) discriminator : 동적 결과 매핑을 위한 요소

```
<resultMap id="vehicleResult" type="Vehicle">
  <id property="id" column="id" />
  <result property="vin" column="vin"/>
  <result property="year" column="year"/>
  <result property="make" column="make"/>
  <result property="model" column="model"/>
  <result property="color" column="color"/>
  <discriminator javaType="int" column="vehicle_type">
    <case value="1" resultMap="carResult"/>
    <case value="2" resultMap="truckResult"/>
    <case value="3" resultMap="vanResult"/>
    <case value="4" resultMap="suvResult"/>
  </discriminator>
</resultMap>
```

discriminator 정의는 colume 과 javaType 속성을 명시한다. colume 은 MyBatis 로 하여금 비교할 값을 찾을 것이다. javaType 은 동일성 테스트와 같은 것을 실행하기 위해 필요하다.
이 예제에서, MyBatis 는 결과데이터에서 각각의 레코드를 가져와서 vehicle_type 값과 비교한다. 만약 discriminator 비교값과 같은 경우가 생기면, 이 경우에 명시된 resultMap 을 사용할 것이다. 해당되는 경우가 없다면 무시된다. 만약 일치하는 경우가 하나도 없다면, MyBatis 는 discriminator 블럭 밖에 정의된 resultMap 을 사용할 것이다. 이렇게 결과 집합 중 일부 데이터 따라 동적인 매핑을 처리하고 싶을 때 discriminator가 유용하게 쓰일수 있다.

- cache

```
<cache
  eviction="FIFO"
  flushInterval="60000"
  size="512"
  readOnly="true"/>
```

- cache

```
<cache
  eviction="FIFO"
  flushInterval="60000"
  size="512"
  readOnly="true"/>
```

- 매핑 구문 파일내 select 구문의 모든 결과가 캐싱된다.
- 매핑 구문 파일내 insert, update 그리고 delete 구문은 캐시 데이터를 지운다(flush).
- 기본적으로 캐싱에는 Least Recently Used (LRU) 알고리즘을 사용된다.
- 어플리케이션이 실행되는 캐시데이터가 유지되기 때문에 특정 시점에 사라지거나 하지않고, 시간 순서대로 지워진다는 보장도 없다.
- 캐시는 리스트나 객체에 대해 쿼리 메소드가 실행될때마다 최대 1024개까지 저장한다.
- 캐시는 읽기/쓰기가 모두 가능한다.

① eviction : 캐시 데이터 교체 전략, LRU, FIFO, SOFT, WEAK 등이 있다.
② flushInterval : 캐시데이터 유지 기간 , ms 단위
③ size : 캐싱할 객체 개수, 기본 1024개
④ readOnly : 캐시데이터의 read/write 속성 지정

Mybatis 가 가진 캐시 기능은 여러가지 제약이 따르기 때문에 최근에는 EhCache 등의 전문 캐시프레임워크들과 연동하여 사용하는 경우가 많은데, 연동 방법은 아주 간단하다. 먼저 해당 캐시 프레임워크와 Mybatis의 연동 모듈 라이브러리의 의존성을 추가한 후 캐시 엘리먼트에 type 속성을 사용하면 된다.

ex) <cache type="org.mybatis.caches.ehcache.EhcacheCache" />

– 매퍼 인터페이스에 어노테이션 을 이용한 매핑 구문 작성

Annotation	target	XML element	description
@CacheNamespace	class	<cache>	
@CacheNamespaceRef	class	<cache-ref>	
@ConstructorArgs	Method	<constructor>	조회 결과를 자바 객체에 매핑할 때 생성자 주입방식을 사용한다.
@Arg	Method	<arg> <idArg>	<constructor> 의 하위 요소들인 <arg> 나 <idArg> 와 동일 id, column, javaType, jdbcType등의 속성을 사용
@TypeDiscriminator	Method	<disriminator>	결과 매핑 객체의 타입을 동적으로 선택.
@Case	Method	<case>	@TypeDiscriminator와 함께 사용되며, <discriminator>의 하위인 <case>와 동일
@Results	Method	<resultMap>	value로 @Result 어노테이션 배열을 갖는다.
@Result	Method	<result> <id>	one 속성은 has-one 관계 정의를 위한 <association> 기능을 하고, many 속성은 has-many 관계 정의를 위한 <collection> 기능을 한다.
@One	Method	<association>	공통적으로 select 속성을 통해 one/many 역할의 데이터를 조회하는 구문을 지정할 수 있으나, 직접 조인을 통한 resultMap 매핑은 순환 참조를 허용하지 않는 자바 어노테이션의 제약으로 인해 지원되지 않는다.
@Many	Method	<collection>	
@MapKey	Method		리턴타입이 Map 메소드에 사용되며, 결과 객체의 list를 객체의 프로퍼티중 하나를 키로 하는 Map으로 변환하기 위해 사용된다.

– 매퍼 인터페이스에 어노테이션 을 이용한 매핑 구문 작성 (계속)

Annotation	target	XML element	description
@Options	Method	매핑구문 속성들	이 애노테이션은 매핑된 구문에 속성으로 존재하는 많은 분기(switch)와 설정 옵션에 접근할 수 있다. 각 구문을 복잡하게 만들기 보다, Options 애노테이션으로 일관되고 깔끔한 방법으로 설정 할 수 있게 한다. 사용가능한 속성들 : useCache=true, flushCache=false, resultSetType=FORWARD_ONLY, statementType=PREPARED, fetchSize=-1, timeout=-1, useGeneratedKeys=false, keyProperty="id", keyColumn="". 자바 애노테이션을 이해하는 것이 중요하다. 자바 애노테이션은 "null"을 셋팅 할 수 없다. 그래서 일단 Options 애노테이션을 사용하면 각각의 속성은 디폴트 값을 사용하게 된다. 디폴트 값이 기대하지 않은 결과를 만들지 않도록 주의해야 한다. keyColumn 은 키 칼럼이 테이블의 첫번째 칼럼이 아닌 특정 데이터베이스에서만(PostgreSQL 같은) 필요하다.
@Insert @Update @Delete @Select	Method	<insert> <update> <delete> <select>	각각의 애노테이션은 실행하고자 하는 SQL 을 표현한다. 각각 문자열의 배열(또는 한개의 문자열)을 가진다. 문자열의 배열이 전달되면, 각각 공백을 두고 하나로 합친다. 자바 코드에서 SQL 을 만들때 발행할 수 있는 "공백 누락" 문제를 해결하도록 도와준다.
@InsertProvider @UpdateProvider @DeleteProvider @SelectProvider	Method	<insert> <update> <delete> <select>	실행시 SQL 을 리턴할 클래스명과 메서드명을 명시하도록 해주는 대체수단의 애노테이션이다. 매핑된 구문을 실행할 때 MyBatis 는 클래스의 인스턴스를 만들고, 메서드를 실행한다. 메서드는 파라미터 객체를 받을 수도 있다. type : 클래스의 패키지 경로를 포함한 전체 이름 method: 클래스의 메서드
@Param	Parameter		매퍼 메서드가 여러개의 파라미터를 가진다면, 이 애노테이션은 이름에 일치하는 매퍼 메서드 파라미터에 적용된다. 반면에 여러개의 파라미터는 순서대로 명명된다. 예를 들어, #{param1}, #{param2} 등이 디폴트다. @Param("person") 를 사용하면, 파라미터는 #{person} 로 명명된다.
@SelectKey	Method	<selectKey>	
@ResultMap	Method		이 애노테이션은 @Select 또는 @SelectProvider 애노테이션을 위해 XML 매퍼의 <resultMap> 요소의 id 를 제공하기 위해 사용된다. XML 에 정의된 결과 매핑을 재사용하도록 해준다. 이 애노테이션은 @Results 나 @ConstructorArgs 를 오버라이드 할 것이다.
@ResultType	Method		이 애노테이션은 결과 핸들러를 사용할때 사용한다. 이 경우 리턴타입은 void이고 마이바티스는 각각의 레코드 정보를 가지는 객체의 타입을 결정하는 방법을 가져야만 한다. XML 결과매핑이 있다면 @ResultMap 애노테이션을 사용하자. 결과타입이 XML에서 <select> 엘리먼트에 명시되어 있다면 다른 애노테이션이 필요하지 않다. 결과타입이 XML에서 <select> 엘리먼트에 명시되어 있지 않은 경우에 이 애노테이션을 사용하자. 예를들어, @Select 애노테이션이 선언되어 있다면 메소드는 결과 핸들러를 사용할 것이다. 결과 타입은 void여야만 하고 이 애노테이션(이나 @ResultMap)을 반드시 사용해야 한다. 이 애노테이션은 메소드 리턴타입이 void가 아니라면 무시한다.

❑ 동적 SQL

– 매퍼 XML 에서 동적 쿼리 지원 엘리먼트의 사용

* OGNL(Object Graph Navigation Language) 문법

문법 파트	예시
객체의 프로퍼티 접근, comment 객체의 userId 필드	comment.userId
객체의 메소드 호출, 현재 객체의 hashcode()호출	hashcode()
배열의 특정 요소 접근, comments 배열의 첫번째 요소	comments[0]

- if 엘리먼트

```
<select id="findActiveBlogWithTitleLike"
     resultType="Blog">
  SELECT * FROM BLOG
  WHERE state = 'ACTIVE'
  <if test="title != null">
    AND title like #{title}
  </if>
</select>
```

if 구문을 사용하여 동적인 조건절을 완성하는 경우, 흔히 매퍼 파라미터의 존재 유무에 따라 다른 조건절이 완성되어야 하는 경우가 있다. 그런데, 매퍼 파라미터가 문자열인 경우 유/무를 판단하기 위해서는 null 체크와 trim 이후의 length 까지 확인해야 하고, 이를 간단하게 처리하기 위한 유틸리티 메소드들을 만들어 사용하기도 한다. 이때 두가지 문제가 발생할 수 있는데, 첫번째는 if구문에서 OGNL 표기법을 사용하여 클래스의 정적 메소드에 접근할 수 있어야 하고, 두번째는 매퍼 파라미터의 특정 프로퍼티에 접근하는 것이 아니라, 파라미터 자체의 유무를 확인하기 위해 매퍼 파라미터 자체를 참조할 수 있는 식별자가 필요하게 된다. 이러한 문제는 다음과 같은 방식으로 처리할 수 있다.

먼저, 정적 메소드의 접근은 OGNL 표기법에 따라 '@'기호를 사용하여 처리한다.

```
public List<Blog> findActiveBlogWithTitleLike(Blog blog);

<select id="findActiveBlogWithTitleLike" resultType="Blog">
  SELECT * FROM BLOG
  WHERE state = 'ACTIVE'
  <if test="@org.apache.commons.lang3.StringUtils@isNotEmpty(title)">
    AND title like #{title}
  </if>
</select>
```

매퍼 파라미터 자체에 대한 접근이 필요한 경우는 @Param 어노테이션을 매퍼 메소드의 파라미터에 선언하여 파라미터맵의 키를 만들어서 해결할 수 있다.

```
public List<Blog> findActiveBlogWithTitleLike(@Param("title") String title);

<select id="findActiveBlogWithTitleLike" parameterType="hashmap"
     resultType="Blog">
  SELECT * FROM BLOG
  WHERE state = 'ACTIVE'
  <if test="@org.apache.commons.lang3.StringUtils@isNotEmpty(title)">
    AND title like #{title}
  </if>
</select>
```

상위의 예제에서 title 은 findActiveBlogWithTitleLike 메소드로 전달된 Blog 객체의 프로퍼티나 혹은 맵의 키가 되는데, 만일 title자체가 findActiveBlogWithTitleLike 메소드의 파라미터로 넘어온다면 그때는 다음과 같이 @Param 어노테이션을 사용하여 매퍼 파라미터의 정확한 이름을 결정해줘야 한다.

- choose, when, otherwise 엘리먼트 : 다중조건형

```
<select id="findActiveBlogLike"
     resultType="Blog">
  SELECT * FROM BLOG WHERE state = 'ACTIVE'
  <choose>
    <when test="title != null">
      AND title like #{title}
    </when>
    <when test="author != null and author.name != null">
      AND author_name like #{author.name}
    </when>
    <otherwise>
      AND featured = 1
    </otherwise>
  </choose>
</select>
```

- trim, where, set 엘리먼트

  왼쪽의 코드처럼 if 문을 사용하여 where 절을 완성하는 경우, 불완전한 조건절이나 피연산자가 부족한 조건식이 만들어 질수도 있다. 그래서 대신 where 엘리먼트로 조건절을 완성하면 오른쪽과 같은 구문이 완성된다.

```
<select id="findActiveBlogLike"
     resultType="Blog">
  SELECT * FROM BLOG
  WHERE
  <if test="state != null">
    state = #{state}
  </if>
  <if test="title != null">
    AND title like #{title}
  </if>
  <if test="author != null and author.name != null">
    AND author_name like #{author.name}
  </if>
</select>
```

```
<select id="findActiveBlogLike"
     resultType="Blog">
  SELECT * FROM BLOG
  <where>
    <if test="state != null">
         state = #{state}
    </if>
    <if test="title != null">
        AND title like #{title}
    </if>
    <if test="author != null and author.name != null">
        AND author_name like #{author.name}
    </if>
  </where>
</select>
```

  비슷하게 update 구문에서 값의 유/무에 따라 컬럼의 수정여부를 결정해야 하는 경우에는 set 엘리먼트를 사용한다.

```
<update id="updateAuthorIfNecessary">
  update Author
    <set>
      <if test="username != null">username=#{username},</if>
      <if test="password != null">password=#{password},</if>
      <if test="email != null">email=#{email},</if>
      <if test="bio != null">bio=#{bio}</if>
    </set>
  where id=#{id}
</update>
```

  where나 set 엘리먼트는 모두 trim 엘리먼트로 좀더 세부적인 제거 혹은 붙임 문자를 지정할 수 있다.

```
<trim prefix="WHERE" prefixOverrides="AND |OR ">
  ...
</trim>
```

- foreach 엘리먼트 : 집합객체의 요소에 대해 반복처리를 위한 엘리먼트

```
<select id="selectPostIn" resultType="domain.blog.Post">
      SELECT *
      FROM POST P
      <where>
            <if test="list != null">
                  ID in
                  <foreach item="postId" index="index" collection="idList"
                              open="(" separator="," close=")">
                        #{postId}
                  </foreach>
            </if>
      </where>
</select>
```

  - collection : 값을 가지고 있는 객체에 대한 참조명. ex) 매퍼 파라미터명
  - item : 컬렉션의 요소 하나를 참조하기 위한 이름.
  - index : 컬렉션의 인덱스 참조
  - open : 컬렉션에서 값을 가져와서 구문을 완성할 때 가장 앞에 붙일 문자
  - close : 가장 마지막에 붙일 문자
  - separator : 컬렉션의 요소와 요소 사이에 붙일 문자

이외에도 자주 사용하진 않지만, Sql Provider 어노테이션들과 SqlBuilder 등을 이용해 java config로 동적 구문을 등록하는 방법들도 있으니 Mybatis 구문빌더 문서를 참고하도록 한다.

❑ User Defined Type(UDT) 처리를 위한 TypeHandler 의 이용

싱글 벨류를 갖는 스칼라 데이터 타입이 아닌 복합적인 데이터 타입이나 컬렉션 객체가 필요할때 UDT(User Defined Type)을 정의해 활용하게 되는데, 이때 Mybatis 의 TypeHandler API 를 활용한다.

- 커스텀 타입의 데이터를 매개로 한 함수나 프로시저의 활용 1.
  - 테스트용 테이블과 해당 테이블 대상 프로시저에서 활용할 UDT 정의
    특정 테이블을 대상으로 단순 DML 이 아닌 절차적인 작업을 수행해야 하는 경우, 해당 테이블의 레코드를 대상으로 UDT 를 정의하고, UDT 의 집합 객체의 형태로 해당 테이블의 데이터를 바인드할 수 있다.
    한건의 레코드를 매개변수로 받아 등록하고, 한건을 조회하여 호출자쪽으로 전달하는 단순 프로시저의 IN/OUT 파라미터 타입으로 사용할 UDT 를 정의한다.

```
CREATE TABLE SAMPLETABLE(
        COL1 NUMBER(4),
        COL2 VARCHAR2(20)
);

CREATE OR REPLACE TYPE SAMPLETYPE IS OBJECT(
        COL1 NUMBER(4),
        COL2 VARCHAR2(20)
);
```

  - UDT 타입의 매개변수를 갖는 프로시저

```
CREATE OR REPLACE PROCEDURE SAMPLEPROC(
        VO IN SAMPLETYPE,
        OUTVO OUT SAMPLETYPE,
        RESULTVAL OUT VARCHAR2
)
IS
        CNT NUMBER;
BEGIN
        INSERT INTO SAMPLETABLE(COL1, COL2)
        VALUES(VO.COL1, VO.COL2);
        SELECT SAMPLETYPE(COL1, COL2) INTO OUTVO FROM SAMPLETABLE
        WHERE ROWNUM = 1
        ORDER BY ROWID DESC;
        SELECT COUNT(*) INTO CNT FROM SAMPLETABLE;
        RESULTVAL := 'SUCCESS, TOTALCOUNT : '||CNT;

        EXCEPTION
        WHEN OTHERS THEN
                ROLLBACK;
                RESULTVAL := SQLCODE||':'||SQLERRM;
END;
```

JDBC 스펙에 따르면 일반적인 데이터베이스 객체를 자바 객체로 매핑할 때, 즉 ResultSet.getObject(index) 를 호출할 때 구체적인 타입을 정의하고 있는 별도의 type Map 을 제공하지 않았다면, JDBC 드라이버는 java.sql.Struct 의 구현체로 데이터베이스 객체를 매핑한다. 예를 들어, 오라클이라면 oracle.sql.STRUCT 클래스의 인스턴스로 데이터베이스 객체를 구체화한다. 경우에 따라 특정한 데이터베이스 객체 하나에 매핑될 수 있는 커스텀 객체의 타입이 필요하다면, SQLData 의 구현체를 정의하거나, 오라클의 경우 ORAData 와 ORADataFactory 의 구현체를 정의해 활용한다. 즉 각 구현체를 정의하고 해당 타입을 드라이버에 typeMap으로 등록해야 한다.

- UDT 하나에 매핑될 SQLData 구현체 정의

```
public class SampleVO implements SQLData{
	private int col1;
	private String col2;

	// 생성자 및 getter/setter

	@Override
	public String getSQLTypeName() throws SQLException {
		return "SAMPLETYPE";
	}

	//take from ResultSet
	@Override
	public void readSQL(SQLInput stream, String typeName) throws SQLException {
		// SQLInput : 데이터베이스의 UDT 객체로부터 값을 읽어올때 사용할 스트림
		// 본 메소드는 ResultSet.getObject  를 호출할때 JDBC 드라이버에 의해 실행됨
		// UDT 내의 프로퍼티 정의 순서대로 읽어옴.
		 col1 = stream.readInt();
		 col2 = stream.readString();
	}

	//pass parameter to Statement
	@Override
	public void writeSQL(SQLOutput stream) throws SQLException {
		//  SQLOutput : 데이터베이스의 UDT 타입( SQLType) 객체에 기록시 사용 스트림
		stream.writeInt(col1);
		stream.writeString(col2);
	}
}
```

- 매퍼 설정

```
public interface IUDTExampleDAO {
	public void callProc(Map<String, Object> paramMap);
}
```

SAMPLEPROC 프로시저를 대상으로 IN 바운드 매개변수 SampleVO 객체를 넘기고 OUT 바운드 매개변수로 SAMPLETYPE 의 데이터베이스 객체를 SampleVO 타입의 인스턴스로 매핑할 목적의 코드 샘플이다.
JDBC 드라이버는 데이터베이스 객체를 getObject 로 읽어올때 typemap 설정이 없는 한 java.sql.Struct 구현체의 인스턴스를 생성한다. 그러나 type map 에 특정 UDT 에 대한 SQLData 구현타입이 등록되어있다면, 해당 타입의 인스턴스를 생성하고 readSQL 메소드를 호출하여 데이터베이스 객체를 SQLData 구현체로 매핑하게 된다.따라서 mybatis 를 이용해 이러한 과정이 처리되기 위해서는 데이터베이스의 데이터를 자바 객체로 매핑하는 과정에서 동작하는 typeHandler 를 추가로 등록해주어야 한다.

```
public class CustomSampleVOTypeHandler extends BaseTypeHandler<SampleVO> {
	private void setTypeMap(Statement stmt, String jdbcTypeName, Class<?> mappedType)
	throws SQLException {
		Connection conn = stmt.getConnection();
		// 기본적으로 비어있는 typeMap 이 돌아옴.
		Map<String, Class<?>> typeMap = conn.getTypeMap();
		// UDT 에 대해 커스텀으로 타입 매핑 정보를 추가하고, 반드시 setTypeMap 호출해야 함.
		typeMap.put(jdbcTypeName, mappedType);
		conn.setTypeMap(typeMap);
	}
```

```
        @Override
        public void setNonNullParameter(PreparedStatement ps, int i,
                                                SampleVO parameter, JdbcType jdbcType)
        throws SQLException {
                setTypeMap(ps, "SAMPLETYPE", SampleVO.class);
                ps.setObject(i, parameter);
        }
        @Override
        public SampleVO getNullableResult(ResultSet rs, String columnName) throws SQLException {
                setTypeMap(rs.getStatement(), "SAMPLETYPE", SampleVO.class);
                return (SampleVO) rs.getObject(columnName);
        }
        @Override
        public SampleVO getNullableResult(ResultSet rs, int columnIndex) throws SQLException {
                setTypeMap(rs.getStatement(), "SAMPLETYPE", SampleVO.class);
                return (SampleVO) rs.getObject(columnIndex);
        }
        @Override
        public SampleVO getNullableResult(CallableStatement cs, int columnIndex) throws
        SQLException {
                setTypeMap(cs, "SAMPLETYPE", SampleVO.class);
                return (SampleVO) cs.getObject(columnIndex);
        }
}
```

구체적인 "SAMPLETYPE" 이라는 SQL UDT에 대해 SQLData를 구현하고 있는 SampleVO 를 매핑 타입으로 설정한 typeMap 을 드라이버에 직접 설정하여 getObject 나 setObject 에서 Struct 객체가 사용되지 않고, SQLData 의 구현체 SampleVO의 인스턴스가 매핑되도록 하고 있다.

```
<mapper namespace="com.tester.udt.example.IUDTExampleDAO"
<!-- 결과 데이터 혹은 out bind 값을 받아올 때는 jdbcType, jdbcTypeName, typeHandler 등에 대한
구체적인 설정이 필요함. -->
<!-- java.sql.SQLException: ORA-03115: unsupported network datatype or representation -->
<!-- "jdbcTypeName=SAMPLETYPE" 을 설정 -->
<select id="callProc" statementType="CALLABLE">
        {
                CALL SAMPLEPROC(
                        #{paramVO,mode=IN},
                        #{outVO,mode=OUT,jdbcType=STRUCT,jdbcTypeName=SAMPLETYPE,
                                typeHandler=com.tester.udt.CustomSampleVOTypeHandler},
                        #{resultval,mode=OUT,jdbcType=VARCHAR,javaType=string}
                )
        }
</select>
```

IN바운드 매개변수의 경우, jdbcType이나 jdbcTypeName, typeHandler 등의 설정이 없어도, SQLData 구현체의 getSQLTypeName 과 writeSQL 메소드 호출을 통해 데이터베이스 객체 생성 및 프로퍼티 값 설정에 아무런 문제가 없으나, 반드시 null 이 아닌 객체를 전달하여야 한다.

```
IUDTExampleDAO exampleDAO = sqlSession.getMapper(IUDTExampleDAO.class);
SampleVO vo = new SampleVO(23, "testValue");
paramMap.put("paramVO", vo);
exampleDAO.callProc(paramMap);
// sqlSession.selectOne("com.tester.udt.example.IUDTExampleDAO.callProc", paramMap);
```

- 커스텀 타입의 데이터를 매개로 한 함수나 프로시저의 활용 2. (Array / Collection)
  - 테스트용 테이블과 해당 테이블 대상 프로시저에서 활용할 UDT 정의
    여러건 데이터를 매개변수로 받아 등록하고, 여러 레코드들을 조회하여 호출자쪽으로 전달하는 집합 객체 활용 프로시저의 IN/OUT 파라미터 타입으로 사용할 UDT 를 정의한다.

```
CREATE OR REPLACE TYPE SAMPLEARRAY IS TABLE OF SAMPLETYPE;
```

  - UDT 타입의 매개변수를 갖는 프로시저

```
CREATE OR REPLACE PROCEDURE  SAMPLEARRAYPROC(
        VOARRAY IN SAMPLEARRAY,
        OUTVOARRAY OUT SAMPLEARRAY,
        RESULTVAL OUT VARCHAR2
)
IS
        CNT NUMBER;
BEGIN
        FORALL I IN 1..VOARRAY.COUNT
                INSERT INTO SAMPLETABLE(COL1, COL2)
                VALUES(VOARRAY(I).COL1, VOARRAY(I).COL2);

        SELECT SAMPLETYPE(COL1, COL2) BULK COLLECT INTO OUTVOARRAY
        FROM SAMPLETABLE;
        SELECT COUNT(*) INTO CNT FROM SAMPLETABLE;
        RESULTVAL := 'SUCCESS, TOTALCOUNT : '||CNT;

        EXCEPTION
        WHEN OTHERS THEN
                ROLLBACK;
                RESULTVAL := SQLCODE||':'||SQLERRM;
END;
```

  - 매퍼 설정

```
public void callArrayProc(Map<String, Object> paramMap);
```

    table 이나 array 등의 데이터베이스 집합 객체를 대상으로 전달되는 매개변수나 반환 객체는 java.sql.Array의 구현체를 이용하게 되는데, 이 경우, 집합을 구성하고 있는 요소객체의 매핑 타입을 typeMap 에 등록하고, 해당 타입의 객체를 수집하여 java.sql.Array 의 구현체를 생성하는 typeHandler 가 필요하다.

```
public class CustomSampleVOArrayTypeHandler extends BaseTypeHandler<SampleVO[]>{
        private void setTypeMap(Statement stmt, String jdbcTypeName, Class<?> mappedType)
                throws SQLException {
                Connection conn = stmt.getConnection();
                Map<String, Class<?>> typeMap = conn.getTypeMap();
                typeMap.put(jdbcTypeName, mappedType);
                conn.setTypeMap(typeMap);
        }
        @Override
        public void setNonNullParameter(PreparedStatement ps, int i,
                SampleVO[] parameter, JdbcType jdbcType) throws SQLException {
                OracleConnection conn = (OracleConnection) ps.getConnection();
                Array array = conn.createARRAY("SAMPLEARRAY", parameter);
                //java.sql.Connection.createArrayOf(elementType, elements) 메소드 미지원(Oracle)
                ps.setArray(i, array);
        }
```

```
	@Override
	public SampleVO[] getNullableResult(ResultSet originRs, String columnName)
			throws SQLException {
			setTypeMap(originRs.getStatement(), "SAMPLETYPE", SampleVO.class);
			Array array = originRs.getArray(columnName);
			// array 에 각 요소에 대해 접근하기 위한 set , 첫번째 컬럼:인덱스, 두번째 컬럼:요소값
			ResultSet rs = array.getResultSet();
			List<SampleVO> tmpList = new ArrayList<>();
			while(rs.next()){
				// typeMap 설정이 필요함.
				tmpList.add((SampleVO)rs.getObject(2));
			}
			return (SampleVO[]) tmpList.toArray(new SampleVO[tmpList.size()]);
	}
	@Override
	public SampleVO[] getNullableResult(ResultSet originRs, int columnIndex)
			throws SQLException {
			setTypeMap(originRs.getStatement(), "SAMPLETYPE", SampleVO.class);
			Array array = originRs.getArray(columnIndex);
			ResultSet rs = array.getResultSet();
			List<SampleVO> tmpList = new ArrayList<>();
			while(rs.next()){
				tmpList.add((SampleVO)rs.getObject(2));
			}
			return (SampleVO[]) tmpList.toArray(new SampleVO[tmpList.size()]);
	}
	@Override
	public SampleVO[] getNullableResult(CallableStatement cs, int columnIndex)
			throws SQLException {
			setTypeMap(cs, "SAMPLETYPE", SampleVO.class);
			Array array = cs.getArray(columnIndex);
			ResultSet rs = array.getResultSet();
			List<SampleVO> tmpList = new ArrayList<>();
			while(rs.next()){
					tmpList.add((SampleVO)rs.getObject(2));
			}
			return (SampleVO[]) tmpList.toArray(new SampleVO[tmpList.size()]);
	}
}
```

```
<select id="callArrayProc" statementType="CALLABLE">
	{
	    CALL
		SAMPLEARRAYPROC(
			#{paramArray,mode=IN,
				typeHandler=com.tester.udt.CustomSampleVOArrayTypeHandler},
			#{outArray,mode=OUT,jdbcType=ARRAY,jdbcTypeName=SAMPLEARRAY,
				typeHandler=com.tester.udt.CustomSampleVOArrayTypeHandler},
			#{resultval,mode=OUT,jdbcType=VARCHAR,javaType=string}
		)
	}
</select>
```

집합객체를 데이터베이스로 전달하거나 집합객체를 조회할 때, setArray / getArray 메소드를 직접적으로 사용할 typeHandler 와 데이터베이스 객체 유형(jdbcType)과 데이터베이스 객체 타입명(jdbcTypeName)을 반드시 명시해야한다.

- 커스텀 타입의 데이터를 매개로 한 함수나 프로시저의 활용 3. (Function)
  - 기 정의된 UDT 를 활용한 함수

    한건 데이터를 매개변수로 받아 등록하고, 한건의 레코드들을 조회하여 호출자쪽으로 반환하는 함수.

```
CREATE OR REPLACE FUNCTION SAMPLEFUNCARRAY(
        INARRAY SAMPLEARRAY
)
RETURN SAMPLEARRAY
IS
        -- 선언문
BEGIN
        -- 실행문
 END;
```

    이 함수를 실행하는 방법은 세가지가 있다.

```
<select id="callFunc" statementType="CALLABLE">
        {
                CALL
                #{outArray,mode=OUT,jdbcType=ARRAY,jdbcTypeName=SAMPLEARRAY,
                        typeHandler=com.tester.udt.CustomSampleVOArrayTypeHandler}
                :=SAMPLEFUNCARRAY(#{paramArray,
                        typeHandler=com.tester.udt.CustomSampleVOArrayTypeHandler})
        }
</select>
```

    익명 블록 내에서 호출하는 형태의 위 코드는 mybatis api 메소드 호출시 전달되는 파라미터맵을 통해 함수의 반환값을 획득하게 된다. 반면, 다음과 같은 형태의 코드는 일반적인 DML 문을 실행했을때와 동일한 방식으로 처리되는데, 함수의 반환값이 alias 를 이용하여 하나의 컬럼 형태로 조회되기 때문에 해당 컬럼에 대한 typeHandler 를 명시하기 위해 resultMap 을 활용하게 된다.

```
<resultMap type="hashmap" id="sampleMap">
        <result column="SAMPLEARRAY" property="outArray"
                typeHandler="com.tester.udt.CustomSampleVOArrayTypeHandler"/>
</resultMap>

<select id="selectCallFuncArray" resultMap="sampleMap">
        SELECT SAMPLEFUNCARRAY(#{paramArray,
                typeHandler=com.tester.udt.CustomSampleVOArrayTypeHandler}) SAMPLEARRAY
        FROM DUAL
</select>
```

    TABLE() 함수를 호출하여 함수의 반환객체를 TABLE 형태로 전환하여 조회할 수 있다. 이때, 결과 데이터를 조회하는 방법은 일반적인 DML 조회 구문한 동일한 방식을 사용하게 된다.

```
<select id="selectCallFuncArrayWithTable" resultType="com.tester.udt.SampleVO">
        SELECT *
        FROM TABLE(
                SAMPLEFUNCARRAY(#{paramArray,
                        typeHandler=com.tester.udt.CustomSampleVOArrayTypeHandler})
        )
</select>
```