# MyBatis 3

*사용자 가이드*

번역자 : 이동국(fromm0@gmail.com, http://openframework.or.kr)

최신버전은 http://kldp.net/projects/fwko/

# Contents

# MyBatis 는 무엇인가?

MyBatis 는 개발자가 지정한 SQL, 저장프로시저 그리고 몇가지 고급 매핑을 지원하는 퍼시스턴스 프레임워크이다. MyBatis 는 JDBC 코드와 수동으로 셋팅하는 파라미터와 결과 매핑을 제거한다. MyBatis 는 데이터베이스 레코드에 원시타입과 Map 인터페이스 그리고 자바 POJO 를 설정하고 매핑하기 위해 XML 과 애노테이션을 사용할 수 있다.

# 시작하기

모든 MyBatis 애플리케이션은 SqlSessionFactory 인스턴스를 사용한다. SqlSessionFactory 인스턴스는 SqlSessionFactoryBuilder 를 사용하여 만들수 있다. SqlSessionFactoryBuilder 는 XML 설정파일에서 SqlSessionFactory 인스턴스를 빌드할 수 있다.

## XML 에서 SqlSessionFactory 빌드하기

XML 파일에서 SqlSessionFactory 인스턴스를 빌드하는 것은 매우 간단한다. 설정을 위해 클래스패스 자원을 사용하는 것을 추천하나, 파일 경로나 file:// URL 로부터 만들어진 Reader 인스턴스를 사용할 수도 있다. MyBatis 는 클래스패스와 다른 위치에서 자원을 로드하는 것으로 좀더 쉽게 해주는 Resources 라는 유틸성 클래스를 가지고 있다.

```
String resource = "org/mybatis/example/Configuration.xml";
Reader reader = Resources.getResourceAsReader(resource);
sqlMapper = new SqlSessionFactoryBuilder().build(reader);
```

XML 설정파일에서 지정하는 MyBatis 의 핵심이 되는 설정은 트랜잭션을 제어하기 위한 TransactionManager 과 함께 데이터베이스 Connection 인스턴스를 가져오기 위한 DataSource 를 포함한다. 세부적인 설정은 조금 뒤에 보고 간단한 예제를 먼저보자.

```
<?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>
 <environments default="development">
   <environment id="development">
   <transactionManager type="JDBC"/>
   <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>
 <mappers>
  <mapper resource="org/mybatis/example/BlogMapper.xml"/>
 </mappers>
</configuration>
```

좀 더 많은 XML 설정방법이 있지만, 위 예제에서는 가장 핵심적인 부분만을 보여주고 있다. XML 가장 위부분에서는 XML 문서의 유효성체크를 위해 필요하다. environment 요소는 트랜잭션 관리와 커넥션 풀링을 위한 환경적인 설정을 나타낸다. mappers 요소는 SQL 코드와 매핑 정의를 가지는 XML 파일인 mapper 의 목록을 지정한다.

## XML 을 사용하지 않고 SqlSessionFactory 빌드하기

XML 보다 자바를 사용해서 직접 설정하길 원한다면, XML 파일과 같은 모든 설정을 제공하는 Configuration 클래스를 사용하면 된다.

```
DataSource dataSource = BlogDataSourceFactory.getBlogDataSource();
TransactionFactory transactionFactory = new JdbcTransactionFactory();
Environment environment =
        new Environment("development", transactionFactory, dataSource);
Configuration configuration = new Configuration(environment);
configuration.addMapper(BlogMapper.class);
SqlSessionFactory sqlSessionFactory =
        new SqlSessionFactoryBuilder().build(configuration);
```

이 설정에서 추가로 해야 할 일은 Mapper 클래스를 추가하는 것이다. Mapper 클래스는 SQL 매핑 애노테이션을 가진 자바 클래스이다. 어쨌든 자바 애노테이션의 몇가지 제약과 몇가지 설정방법의 목잡함에도 불구하고, XML 매핑은 세부적인 매핑을 위해 언제든 필요하다. 좀더 세부적인 내용은 조금 뒤 다시 보도록 하자.

## SqlSessionFactory 에서 SqlSession 만들기

SqlSessionFactory 이름에서 보듯이, SqlSession 인스턴스를 만들수 있다. SqlSession 은 데이터베이스에 대해 SQL 명령어를 실행하기 위해 필요한 모든 메서드를 가지고 있다. 그래서 SqlSession 인스턴스를 통해 직접 SQL 구문을 실행할 수 있다. 예를 들면 :

```
SqlSession session = sqlMapper.openSession();
try {
  Blog blog = (Blog) session.selectOne(
    "org.mybatis.example.BlogMapper.selectBlog", 101);
} finally {
  session.close();
}
```

이 방법이 MyBatis 의 이전버전을 사용한 사람이라면 굉장히 친숙할 것이다. 하지만 좀더 좋은 방법이 생겼다. 주어진 SQL 구문의 파라미터와 리턴값을 설명하는 인터페이스(예를 들면, BlogMapper.class )를 사용하여, 문자열 처리 오류나 타입 캐스팅 오류 없이 좀더 타입에 안전하고 깔끔하게 실행할 수 있다.

예를 들면:

```
SqlSession session = sqlSessionFactory.openSession();
try {
```

```
  BlogMapper mapper = session.getMapper(BlogMapper.class);
  Blog blog = mapper.selectBlog(101);
} finally {
  session.close();
}
```

그럼 좀더 자세히 살펴보자.

## 매핑된 SQL 구문 살펴보기

이 시점에 SqlSession 이나 Mapper 클래스가 정확히 어떻게 실행되는지 궁금할 것이다. 매핑된 SQL 구문에 대한 내용이 가장 중요하다. 그래서 이 문서 전반에서 가장 자주 다루어진다. 하지만 다음의 두가지 예제를 통해 정확히 어떻게 작동하는지에 대해서는 충분히 이해하게 될 것이다.

위 예제처럼, 구문은 XML 이나 애노테이션을 사용해서 정의할 수 있다. 그럼 먼저 XML 을 보자. MyBatis 가 제공하는 대부분의 기능은 XML 을 통해 매핑 기법을 사용한다. 이전에 MyBatis 를 사용했었다면 쉽게 이해되겠지만, XML 매핑 문서에 이전보다 많은 기능이 추가되었다. SqlSession 을 호출하는 XML 기반의 매핑 구문이다.

```
<?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="org.mybatis.example.BlogMapper">
 <select id="selectBlog" parameterType="int" resultType="Blog">
   select * from Blog where id = #{id}
 </select>
</mapper>
```

간단한 예제치고는 조금 복잡해 보이지만 실제로는 굉장히 간단하다. 한 개의 매퍼 XML 파일에는 많은 수의 매핑 구문을 정의할 수 있다. XML 도입무의 헤더와 doctype 을 제외하면, 나머지는 쉽게 이해되는 구문의 형태이다. 여기선 org.mybatis.example.BlogMapper 명명공간에서 selectBlog 라는 매핑 구문을 정의했고, 이는 결과적으로 org.mybatis.example.BlogMapper.selectBlog 형태로 실제 명시되게 된다. 그래서 다음처럼 사용하게 되는 셈이다.

```
Blog blog = (Blog) session.selectOne(
   "org.mybatis.example.BlogMapper.selectBlog", 101);
```

이건 마치 패키지를 포함한 전체 경로의 클래스내 메서드를 호출하는 것과 비슷한 형태이다. 이 이름은 매핑된 select 구문의 이름과 파라미터 그리고 리턴타입을 가진 명명공간과 같은 이름의 Mapper 클래스와 직접 매핑될 수 있다. 이건 위에서 본것과 같은 Mapper 인터페이스의 메서드를 간단히 호출하도록 허용한다. 위 예제에 대응되는 형태는 아래와 같다.

```
BlogMapper mapper = session.getMapper(BlogMapper.class);
Blog blog = mapper.selectBlog(101);
```

두번째 방법은 많은 장점을 가진다. 먼저 문자열에 의존하지 않는다는 것이다. 이는 애플리케이션을 좀더 안전하게 만든다. 두번째는 개발자가 IDE 를 사용할 때, 매핑된 SQL 구문을 사용할때의 수고를 덜어준다. 세번째는 리턴타입에 대해 타입 캐스팅을 하지 않아도 된다. 그래서 BlogMapper 인터페이스는 깔끔하고 리턴 타입에 대해 타입에 안전하며 이는 파라미터에도 그대로 적용된다.

## 명명공간(Namespaces)에 대한 설명

➔ **명명공간(Namespaces)**이 이전버전에서는 사실 선택사항이었다. 하지만 이제는 패키지경로를 포함한 전체 이름을 가진 구문을 구분하기 위해 필수로 사용해야 한다.

명명공간은 인터페이스 바인딩을 가능하게 한다. 명명공간을 사용하고 자바 패키지의 명명공간을 두면 코드가 깔끔해지고 MyBatis 의 사용성이 크게 향상될 것이다.

➔ **이름 분석(Name Resolution)**: 타이핑을 줄이기 위해, MyBatis 는 구문과 결과맵, 캐시등의 모든 설정요소를 위한 이름 분석 규칙을 사용한다.

- “com.mypackage.MyMapper.selectAllThings” 과 같은 패키지를 포함한 전체 경로명(Fully qualified names)은 같은 형태의 경로가 있다면 그 경로내에서 직접 찾는다.
- “selectAllThings” 과 같은 짧은 형태의 이름은 모호하지 않은 엔트리를 참고하기 위해 사용될 수 있다. 그래서 짧은 이름은 모호해서 에러를 자주 보게 되니 되도록 이면 전체 경로를 사용해야 할 것이다.

BlogMapper 와 같은 Mapper 클래스에는 몇가지 트릭이 있다. 매핑된 구문은 XML 에 전혀 매핑될 필요가 없다. 대신 자바 애노테이션을 사용할 수 있다. 예를 들어, 위 XML 예제는 다음과 같은 형태로 대체될 수 있다.

```
package org.mybatis.example;
public interface BlogMapper {
  @Select("SELECT * FROM blog WHERE id = #{id}")
  Blog selectBlog(int id);
}
```

간단한 구문에서는 애노테이션이 좀더 깔끔하다. 어쨌든 자바 애노테이션은 좀더 복잡한 구문에서는 제한적이고 코드를 엉망으로 만들 수 있다. 그러므로, 복잡하게 사용하고자 한다면, XML 매핑 구문을 사용하는 것이 좀더 나을 것이다.

## 스코프(Scope) 와 생명주기(Lifecycle)

이제부터 다룰 스코프와 생명주기에 대해서 이해하는 것은 매우 중요하다. 스코프와 생명주기를 잘못 사용하는 것은 다양한 동시성 문제를 야기할 수 있다..

### *SqlSessionFactoryBuilder*

이 클래스는 인스턴스회되어 사용되고 던져질 수 있다. SqlSessionFactory 를 생성한 후 유지할 필요는 없다. 그러므로 SqlSessionFactoryBuilder 인스턴스의 가장 좋은 스코프는 메서드 스코프(예를 들면, 메서드 지역변수)이다. 여러 개의

SqlSessionFactory 인스턴스를 빌드하기 위해 SqlSessionFactoryBuilder 를 재사용할 수도 있지만 유지하지 않는 것이 가장 좋다..

## *SqlSessionFactory*

한번 만든뒤, SqlSessionFactory 는 애플리케이션을 실행하는 동안 존재해야만 한다. 그래서 삭제하거나 재생성할 필요가 없다. 애플리케이션이 실행되는 동안 여러 차례 SqlSessionFactory 를 다시 빌드하지 않는 것이 가장 좋은 형태이다. 재빌드하는 형태는 결과적으로 “나쁜 냄새” 가 나도록 한다. 그러므로 SqlSessionFactory 의 가장 좋은 스코프는 애플리케이션 스코프이다. 애플리케이션 스코프로 유지하기 위한 다양한 방법이 존재한다. 가장 간단한 방법은 싱글턴 패턴이나 static 싱글턴 패턴을 사용하는 것이다. 또는 구글 Guice 나 Spring 과 같은 의존성 삽입 컨테이너를 선호할 수 도 있다. 이러한 프레임워크는 SqlSessionFactory 의 생명주기를 싱글턴으로 관리할 것이다.

## *SqlSession*

각각의 쓰레드는 자체적으로 SqlSession 인스턴스를 가져야 한다. SqlSession 인스턴스는 공유되지 않고 쓰레드에 안전하지도 않다. 그러므로 가장 좋은 스코프는 요청 또는 메서드 스코프이다. SqlSession 을 static 필드나 클래스의 인스턴스 필드로 지정해서는 안된다. 그리고 서블릿 프레임워크의 HttpSession 과 같은 관리 스코프에 둬서도 안된다. 어떠한 종류의 웹 프레임워크를 사용한다면, HTTP 요청과 유사한 스코프에 두는 것으로 고려해야 한다. 달리 말해서, HTTP 요청을 받을?때마다 만들고, 응답을 리턴할때마다 SqlSession 을 닫을 수 있다. SqlSession 을 닫는 것은 중요하다. 언제나 finally 블록에서 닫아야만 한다. 다음은 SqlSession 을 닫는 것을 확인하는 표준적인 형태다.

```
SqlSession session = sqlSessionFactory.openSession();
try {
  // do work
} finally {
  session.close();
}
```

## *Mapper 인스턴스*

Mapper 는 매핑된 구문을 바인딩 하기 위해 만들어야 할 인터페이스이다. mapper 인터페이스의 인스턴스는 SqlSession 에서 생성한다. 그래서 mapper 인스턴스의 가장 좋은 스코프는 SqlSession 과 동일하다. 어쨌든 mapper 인스턴스의 가장 좋은 스코프는 메서드 스코프이다. 사용할 메서드가 호출되면 생성되고 끝난다. 명시적으로 닫을 필요는 없다.

```
SqlSession session = sqlSessionFactory.openSession();
try {
  BlogMapper mapper = session.getMapper(BlogMapper.class);
  // do work
} finally {
  session.close();
}
```

# 매퍼 설정 XML

MyBatis XML 설정파일은 다양한 셋팅과 프로퍼티를 가진다. 문서의 구조는 다음과 같다.:

- configuration
  - properties
  - settings
  - typeAliases
  - typeHandlers
  - objectFactory
  - plugins
  - environments
    - environment
      - transactionManager
      - dataSource
  - mappers

## properties

이 설정은 외부에 옮길 수 있다. 자바 프로퍼티 파일 인스턴스에 설정할 수도 있고, properties 요소의 하위 요소에 둘수도 있다. 예를 들면:

```
<properties resource="org/mybatis/example/config.properties">
  <property name="username" value="dev_user"/>
  <property name="password" value="F2Fa3!33TYyg"/>
</properties>
```

속성들은 파일 도처에 둘수도 있다. 예를 들면:

```
<dataSource type="POOLED">
  <property name="driver" value="${driver}"/>
  <property name="url" value="${url}"/>
  <property name="username" value="${username}"/>
  <property name="password" value="${password}"/>
</dataSource>
```

이 에제에서 username 과 password 는 properties 요소의 설정된 값으로 대체될 수 있다. driver 와 url 속성은 config.properties 파일에 포함된 값으로 대체될 수도 있다. 이것은 설정에 대한 다양한 옵션을 제공하는 셈이다.

속성은 SqlSessionBuilder.build() 메서드에 전달될 수 있다. 예를 들면:

```
SqlSessionFactory factory =
        sqlSessionFactoryBuilder.build(reader, props);

// ... or ...

SqlSessionFactory factory =
```

```
sqlSessionFactoryBuilder.build(reader, environment, props);
```

속성이 한개 이상 존재한다면, MyBatis 는 일정한 순서로 로드한다.:

- properties 요소에 명시된 속성을 가장 먼저 읽는다.
- properties 요소의 클래스패스 자원이나 url 속성으로 부터 로드된 속성을 두번재로 읽는다. 그래서 이미 읽은 값이 있다면 덮어쓴다.
- 마지막으로 메서드 파라미터로 전달된 속성을 읽는다. 앞서 로드된 값을 덮어쓴다.

그래서 가장 우선순위가 높은 속성은 메서드의 파라미터로 전달된 값이고 그 다음은 자원및 url 속성이고 마지막은 properties 요소에 명시된 값이다.

## settings

런타임시 MyBatis 의 행위를 조정하기 위한 중요한 값들이다. 다음표는 셋팅과 그 의미 그리고 디폴트 값을 설명한다.

| 셋팅 | 설명 | 사용가능한 값들 | 디폴트 |
|---|---|---|---|
| cacheEnabled | 설정에서 각 mapper 에 설정된 캐시를 전역적으로 사용할지 말지에 대한 여부 | true \| false | true |
| lazyLoadingEnabled | 늦은 로딩을 사용할지에 대한 여부. 사용하지 않는다면 모두 즉시 로딩할 것이다. | true \| false | true |
| aggressiveLazyLoading | 활성화 상태로 두게 되면 늦은(lazy) 로딩 프로퍼티를 가진 객체는 호출에 따라 로드될 것이다. 반면에 개별 프로퍼티는 요청할때 로드된다. | true \| false | true |
| multipleResultSetsEnabled | 한개의 구문에서 여러개의 ResultSet 을 허용할지의 여부(드라이버가 해당 기능을 지원해야 함) | true \| false | true |
| useColumnLabel | 칼럼명 대신에 칼럼라벨을 사용. 드라이버마다 조금 다르게 작동한다. 문서와 간단한 테스트를 통해 실제 기대하는 것처럼 작동하는지 확인해야 한다. | true \| false | true |
| useGeneratedKeys | 생성키에 대한 JDBC 지원을 허용. 지원하는 드라이버가 필요하다. true 로 설정하면 생성키를 강제로 생성한다. 일부 드라이버(예를들면, Derby)에서는 이 설정을 무시한다. | true \| false | False |
| autoMappingBehavior | MyBatis 가 칼럼을 필드/프로퍼티에 자동으로 매핑할지와 방법에 대해 명시. PARTIAL 은 간단한 자동매핑만 할뿐, 내포된 결과에 대해서는 처리하지 않는다. FULL 은 처리가능한 모든 자동매핑을 처리한다. | NONE, PARTIAL, FULL | PARTIAL |
| defaultExecutorType | 디폴트 실행자(executor) 설정. SIMPLE 실행자는 특별히 하는 것이 없다. REUSE 실행자는 PreparedStatement 를 재사용한다. BATCH 실행자는 구문을 재사용하고 수정을 배치처리한다. | SIMPLE REUSE BATCH | SIMPLE |
| defaultStatementTimeout | 데이터베이스로의 응답을 얼마나 오래 기다릴지를 판단하는 타임아웃을 셋팅 | 양수 | 셋팅되지 않음(null) |

위 설정을 모두 사용한 setting 요소의 예제이다:

```
<settings>
  <setting name="cacheEnabled" value="true"/>
  <setting name="lazyLoadingEnabled" value="true"/>
  <setting name="multipleResultSetsEnabled" value="true"/>
  <setting name="useColumnLabel" value="true"/>
  <setting name="useGeneratedKeys" value="false"/>
  <setting name="enhancementEnabled" value="false"/>
  <setting name="defaultExecutorType" value="SIMPLE"/>
  <setting name="defaultStatementTimeout" value="25000"/>
</settings>
```

## typeAliases

타입 별칭은 자바 타입에 대한 좀더 짧은 이름이다. 오직 XML 설정에서만 사용되며, 타이핑을 줄이기 위해 존재한다. 예를들면:

```
<typeAliases>
  <typeAlias alias="Author" type="domain.blog.Author"/>
  <typeAlias alias="Blog" type="domain.blog.Blog"/>
  <typeAlias alias="Comment" type="domain.blog.Comment"/>
  <typeAlias alias="Post" type="domain.blog.Post"/>
  <typeAlias alias="Section" type="domain.blog.Section"/>
  <typeAlias alias="Tag" type="domain.blog.Tag"/>
</typeAliases>
```

이 설정에서, “Blog” 는 도처에서 “domain.blog.Blog” 대신 사용될 수 있다.

공통의 자바타입에 대해서는 내장된 타입 별칭이 있다. 이 모두 대소문자를 가린다..

| 별칭 | 매핑된 타입 |
|---|---|
| _byte | Byte |
| _long | Long |
| _short | Short |
| _int | Int |
| _integer | int |
| _double | double |
| _float | float |
| _boolean | boolean |
| string | String |
| byte | Byte |
| long | Long |
| short | Short |
| int | Integer |
| integer | Integer |
| double | Double |

| 별칭 | 매핑된 타입 |
|---|---|
| float | Float |
| boolean | Boolean |
| date | Date |
| decimal | BigDecimal |
| bigdecimal | BigDecimal |
| object | Object |
| map | Map |
| hashmap | HashMap |
| list | List |
| arraylist | ArrayList |
| collection | Collection |
| iterator | Iterator |

## typeHandlers

MyBatis 가 PreparedStatement 에 파라미터를 셋팅하고 ResultSet 에서 값을 가져올때마다, TypeHandler 는 적절한 자바 타입의 값을 가져오기 위해 사용된다. 다음의 표는 디폴트 TypeHandlers 를 설명한다..

| 타입 핸들러 | 자바 타입 | JDBC 타입 |
|---|---|---|
| BooleanTypeHandler | Boolean, boolean | 어떤 호환가능한 BOOLEAN |
| ByteTypeHandler | Byte, byte | 어떤 호환가능한 NUMERIC 또는 BYTE |
| ShortTypeHandler | Short, short | 어떤 호환가능한 NUMERIC 또는 SHORT INTEGER |
| IntegerTypeHandler | Integer, int | 어떤 호환가능한 NUMERIC 또는 INTEGER |
| LongTypeHandler | Long, long | 어떤 호환가능한 NUMERIC 또는 LONG INTEGER |
| FloatTypeHandler | Float, float | 어떤 호환가능한 NUMERIC 또는 FLOAT |
| DoubleTypeHandler | Double, double | 어떤 호환가능한 NUMERIC 또는 DOUBLE |
| BigDecimalTypeHandler | BigDecimal | 어떤 호환가능한 NUMERIC 또는 DECIMAL |
| StringTypeHandler | String | CHAR, VARCHAR |
| ClobTypeHandler | String | CLOB, LONGVARCHAR |
| NStringTypeHandler | String | NVARCHAR, NCHAR |
| NClobTypeHandler | String | NCLOB |
| ByteArrayTypeHandler | byte[] | 어떤 호환가능한 byte 스트림 타입 |
| BlobTypeHandler | byte[] | BLOB, LONGVARBINARY |
| DateTypeHandler | Date (java.util) | TIMESTAMP |
| DateOnlyTypeHandler | Date (java.util) | DATE |
| TimeOnlyTypeHandler | Date (java.util) | TIME |
| SqlTimestampTypeHandler | Timestamp (java.sql) | TIMESTAMP |
| SqlDateTypeHadler | Date (java.sql) | DATE |
| SqlTimeTypeHandler | Time (java.sql) | TIME |
| ObjectTypeHandler | Any | OTHER, 또는 명시하지 않는 |
| EnumTypeHandler | Enumeration Type | VARCHAR - 문자열 호환타입. |

지원하지 않거나 비표준인 타입에 대해서는 당신 스스로 만들어서 타입핸들러를 오버라이드할 수 있다. 그러기 위해서는 TypeHandler 인터페이스를 구현하고 자바 타입에 TypeHandler 를 매핑하면 된다. 예를 들면:

```
// ExampleTypeHandler.java
public class ExampleTypeHandler implements TypeHandler {
  public void setParameter(
          PreparedStatement ps, int i, Object parameter,JdbcType jdbcType)
                  throws SQLException {
    ps.setString(i, (String) parameter);
  }
  public Object getResult(
                  ResultSet rs, String columnName)
                          throws SQLException {
    return rs.getString(columnName);
  }
  public Object getResult(
                  CallableStatement cs, int columnIndex)
                          throws SQLException {
    return cs.getString(columnIndex);
  }
}

// MapperConfig.xml
<typeHandlers>
          <typeHandler javaType="String" jdbcType="VARCHAR"
                  handler="org.mybatis.example.ExampleTypeHandler"/>
</typeHandlers>
```

이러한 TypeHandler 를 사용하는 것은 자바 String 프로퍼티와 VARCHAR 파라미터 및 결과를 위해 이미 존재하는 핸들러를 오버라이드하게 될 것이다. MyBatis 는 타입을 판단하기 위해 데이터베이스의 메타데이터를 보지 않는다. 그래서 파라미터와 결과에 정확한 타입 핸들러를 매핑해야 한다. MyBatis 가 구문이 실행될때까지는 데이터 타입에 대해 모르기 때문이다..

## objectFactory

매번 MyBatis 는 결과 객체의 인스턴스를 만들기 위해 ObjectFactory 를 사용한다. 디폴트 ObjectFactory 는 디폴트 생성자를 가진 대상 클래스를 인스턴스화하는 것보다 조금 더 많은 작업을 한다. ObjectFactory 의 디폴트 행위를 오버라이드하고자 한다면, 만들 수 있다. 예를 들면:

```
// ExampleObjectFactory.java
public class ExampleObjectFactory extends DefaultObjectFactory {
  public Object create(Class type) {
    return super.create(type);
  }
  public Object create(
                          Class type,
                          List<Class> constructorArgTypes,
                          List<Object> constructorArgs) {
    return super.create(type, constructorArgTypes, constructorArgs);
```

```
  }
  public void setProperties(Properties properties) {
    super.setProperties(properties);
  }
}

// MapperConfig.xml
<objectFactory type="org.mybatis.example.ExampleObjectFactory">
  <property name="someProperty" value="100"/>
</objectFactory>
```

ObjectFactory 인터페이스는 매우 간단한다. 두개의 create 메서드를 가지고 있으며, 하나는 디폴트 생성자를 처리하고 다른 하나는 파라미터를 가진 생성자를 처리한다. 마지막으로 setProperties 메서드는 ObjectFactory 를 설정하기 위해 사용될 수 있다. objectFactory 요소에 정의된 프로퍼티는 ObjectFactory 인스턴스가 초기화된 후 setProperties 에 전달될 것이다.

## plugins

MyBatis 는 매핑 구문을 실행하는 어떤 시점에 호출을 가로챈다. 기본적으로 MyBatis 는 메서드 호출을 가로채기 위한 플러그인을 허용한다.

- Executor
  (update, query, flushStatements, commit, rollback, getTransaction, close, isClosed)
- ParameterHandler
  (getParameterObject, setParameters)
- ResultSetHandler
  (handleResultSets, handleOutputParameters)
- StatementHandler
  (prepare, parameterize, batch, update, query)

이 클래스들의 메서드는 각각 메서드 시그니처를 통해 찾을 수 있고 소스코드는 MyBatis 릴리즈 파일에서 찾을 수 있다. 오버라이드할 메서드의 행위를 이해해야만 한다. 주어진 메서드의 행위를 변경하거나 오버라이드하고자 한다면, MyBatis 의 핵심기능에 악영향을 줄수도 있다. 이러한 로우레벨 클래스와 메서드들은 주의를 해서 사용해야 한다.

플러그인을 사용하는 것을 제공하는 기능에 다소 간단하다. Interceptor 인터페이스를 간단히 구현해서, 가로채고(intercept) 싶은 시그니처를 명시해야 한다.

```
// ExamplePlugin.java
@Intercepts({@Signature(
        type= Executor.class,
        method = "update",
        args = {MappedStatement.class,Object.class})})
public class ExamplePlugin implements Interceptor {
```

```
  public Object intercept(Invocation invocation) throws Throwable {
    return invocation.proceed();
  }
  public Object plugin(Object target) {
    return Plugin.wrap(target, this);
  }
  public void setProperties(Properties properties) {
  }
}

// MapperConfig.xml
<plugins>
  <plugin interceptor="org.mybatis.example.ExamplePlugin">
    <property name="someProperty" value="100"/>
  </plugin>
</plugins>
```

위 플러그인은 매핑된 구문의 로우레벨 실행을 책임지는 내부 객체인 Executor 인스턴스의 “update” 메서드 호출을 모두 가로챌것이다.

**설정파일 오버라이드하기**

플러그인을 사용해서 MyBatis 핵심 행위를 변경하기 위해, Configuration 클래스 전체를 오버라이드 할 수 있다. 이 클래스를 확장하고 내부 메서드를 오버라이드하고, sqlSessionFactoryBuilder.build(myConfig) 메서드에 그 객체를 넣어주면 된다. 다시 얘기하지만 이 작업은 MyBatis 에 큰 영향을 줄수 있으니 주의해서 해야 한다.

## environments

MyBatis 는 여러개의 환경으로 설정될 수 있다. 여러가지 이유로 여러개의 데이터베이스에 SQL Map 을 적용하는데 도움이 된다. 예를들어, 개발, 테스트, 리얼 환경을 위해 별도의 설정을 가지거나, 같은 스키마를 여러개의 DBMS 제품을 사용할 경우들이다. 그외에도 많은 경우가 있을 수 있다.

**중요하게 기억해야 할 것은, 다중 환경을 설정할 수는 있지만, SqlSessionFactory 인스턴스마다 한개만 사용할 수 있다는 것이다.**

두개의 데이터베이스에 연결하고 싶다면, SqlSessionFactory 인스턴스를 두개 만들 필요가 있다. 세개의 데이터베이스를 사용한다면, 역시 3 개의 인스턴스를 필요로 한다. 기억하기 쉽게

- ⇨ **데이터베이스별로 하나의 SqlSessionFactory**

환경을 명시하기 위해, SqlSessionFactoryBuilder 에 옵션으로 추가 파라미터를 주면 된다. 환경을 선택하는 두가지 시그니처는

```
SqlSessionFactory  factory = sqlSessionFactoryBuilder.build(reader, environment);
SqlSessionFactory  factory = sqlSessionFactoryBuilder.build(reader, environment,properties);
```

environment 파라미터가 없으면, 디폴트 환경이 로드된다.

```
SqlSessionFactory  factory = sqlSessionFactoryBuilder.build(reader);
SqlSessionFactory  factory = sqlSessionFactoryBuilder.build(reader,properties);
```

environments 요소는 환경을 설정하는 방법을 정의한다.

```
<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>
```

중요한 부분을 살펴보면

- 디폴트 환경(Environment) ID (예를 들면. default=”development”).
- 각각의 환경을 정의한 환경(Environment) ID (예를 들면. id=”development”).
- TransactionManager 설정 (예를 들면. type=”JDBC”)
- DataSource 설정 (예를 들면. type=”POOLED”)

디폴트 환경(environment)과 환경(environment) ID 는 용어 자체가 역할을 설명한다.

## transactionManager

MyBatis 는 두가지 타입의 TransactionManager 를 제공한다.

- **JDBC** - 이 설정은 간단하게 JDBC 커밋과 롤백을 처리하기 위해 사용된다. 트랜잭션의 스코프를 관리하기 위해 dataSource 로 부터 커넥션을 가져온다.
- **MANAGED** - 이 설정은 어떤것도 하지 않는다. 결코 커밋이나 롤백을 하지 않는다. 대신 컨테이너가 트랜잭션의 모든 생명주기를 관리한다. 디플트로 커넥션을 닫긴 한다. 하지만 몇몇 컨테이너는 커넥션을 닫는 것 또한 기대하지 않기 때문에, 커넥션 닫는 것으로 멈추고자 한다면, “closeConnection” 프로퍼티를 false 로 설정하라. 예를 들면:

```
<transactionManager type="MANAGED">
    <property name="closeConnection" value="false"/>
</transactionManager>
```

TransactionManager 타입은 어떠한 프로퍼티도 필요하지 않다. 어쨌든 둘다 타입 별칭이 있다. 즉 TransactionFactory 를 위한 클래스 명이나 타입 별칭 중 하나를 사용할 수 있다.

```
public interface TransactionFactory {
  void setProperties(Properties props);
  Transaction newTransaction(Connection conn, boolean autoCommit);
}
```

XML 에 설정된 프로퍼티는 인스턴스를 만든 뒤 setProperties() 메서드에 전달할 것이다. 당신의 구현체가 Transaction 구현체를 만들 필요가 있을 것이다.:

```
public interface Transaction {
  Connection getConnection();
  void commit() throws SQLException;
  void rollback() throws SQLException;
  void close() throws SQLException;
}
```

이 두개의 인터페이스를 사용하여, MyBatis 가 트랜잭션을 처리하는 방법을 완벽하게 정의할 수 있다..

## dataSource

dataSource 요소는 표준 JDBC DataSource 인터페이스를 사용하여 JDBC Connection 객체의 소스를 설정한다.

⇨ 대부분의 MyBatis 애플리케이션은 예제처럼 dataSource 를 설정할 것이다.

여기엔 3 가지의 내장된 dataSource 타입이 있다.

**UNPOOLED** - 이 구현체는 매번 요청에 대해 커넥션을 열고 닫는 간단한 DataSource 이다. 조금 늦긴 하지만 성능을 크게 필요로 하지 않는 간단한 애플리케이션을 위해서는 괜찮은 선택이다. UNPOOLED DataSource 는 5 개의 프로퍼티만으로 설정된다.

- **driver** - JDBC 드라이버의 패키지 경로를 포함한 결제 자바 클래스명
- **url** - 데이터베이스 인스턴스에 대한 JDBC URL.
- **username** - 데이터베이스에 로그인 할 때 사용할 사용자명
- **password** - 데이터베이스에 로그인 할 때 사용할 패스워드
- **defaultTransactionIsolationLevel** - 커넥션에 대한 디폴트 트랜잭션 격리 레벨

필수는 아니지만 선택적으로 데이터베이스 드라이버에 프로퍼티를 전달할 수 도 있다. 그러기 위해서는 다음 예제처럼 “driver.” 로 시작하는 접두어로 프로퍼티를 명시하면 된다.

- **driver.encoding=UTF8**

이 설정은 “encoding” 프로퍼티를 “UTF8”로 설정하게 된다. 이 방법외에도 DriverManager.getConnection(url, driverProperties) 메서드를 통해서도 프로퍼티를 설정할 수 있다.

**POOLED** - DataSource 에 풀링이 적용된 JDBC 커넥션을 위한 구현체이다. 이는 새로운 Connection 인스턴스를 생성하기 위해 매번 초기화하는 것을 피하게 해준다. 그래서 빠른 응답을 요구하는 웹 애플리케이션에서는 가장 흔히 사용되고 있다.

UNPOOLED DataSource 에 비해, 많은 프로퍼티를 설정할 수 있다.

- **poolMaximumActiveConnections** - 주어진 시간에 존재할 수 있는 활성화된(사용중인) 커넥션의 수. 디폴트는 10 이다.
- **poolMaximumIdleConnections** - 주어진 시간에 존재할 수 있는 유휴 커넥션의 수
- **poolMaximumCheckoutTime** - 강제로 리턴되기 전에 풀에서 “체크아웃” 될 수 있는 커넥션의 시간. 디폴트는 20000ms(20 초)
- **poolTimeToWait** - 풀이 로그 상태를 출력하고 비정상적으로 긴 경우 커넥션을 다시 얻을려고 시도하는 로우 레벨 셋팅. 디폴트는 20000ms(20 초)
- **poolPingQuery** - 커넥션이 작업하기 좋은 상태이고 요청을 받아서 처리할 준비가 되었는지 체크하기 위해 데이터베이스에 던지는 핑쿼리(Ping Query). 디폴트는 “핑 쿼리가 없음” 이다. 이 설정은 대부분의 데이터베이스로 하여금 에러메시지를 보게 할수도 있다.
- **poolPingEnabled** - 핑쿼리를 사용할지 말지를 결정. 사용한다면, 오류가 없는(그리고 빠른) SQL 을 사용하여 poolPingQuery 프로퍼티를 셋팅해야 한다. 디폴트는 false 이다.
- **poolPingConnectionsNotUsedFor** - poolPingQuery 가 얼마나 자주 사용될지 설정한다. 필요이상의 핑을 피하기 위해 데이터베이스의 타임아웃 값과 같을 수 있다. 디폴트는 0 이다. 디폴트 값은 poolPingEnabled 가 true 일 경우에만, 모든 커넥션이 매번 핑을 던지는 값이다.

**JNDI** - 이 DataSource 구현체는 컨테이너에 따라 설정이 변경되며, JNDI 컨텍스트를 참조한다. 이 DataSource 는 오직 두개의 프로퍼티만을 요구한다.

- **initial_context** - 이 프로퍼티는 InitialContext 에서 컨텍스트를 찾기(예를 들어, initialContext.lookup(initial_context)) 위해 사용된다. 이 프로퍼티는 선택적인 값이다. 이 설정을 생략하면, data_source 프로퍼티가 InitialContext 에서 직접 찾을 것이다.
- **data_source** - DataSource 인스턴스의 참조를 찾을 수 있는 컨텍스트 경로이다. initial_context 룩업을 통해 리턴된 컨텍스트에서 찾을 것이다. initial_context 가 지원되지 않는다면, InitialContext 에서 직접 찾을 것이다.

다른 DataSource 설정과 유사하게, 다음처럼 “env.” 를 접두어로 프로퍼티를 전달할 수 있다.

- env.encoding=UTF8

이 설정은 인스턴스화할 때 InitialContext 생성자에 “encoding” 프로퍼티를 “UTF8” 로 전달한다.

### mappers

이제 우리는 매핑된 SQL 구문을 정의할 시간이다. 하지만 먼저, 설정을 어디에 둘지 결정해야 한다. 자바는 자동으로 리소스를 찾기 위한 좋은 방법을 제공하지 않는다. 그래서 가장 좋은 건 어디서 찾으라고 지정하는 것이다. 클래스패스에 상대적으로 리소스를 지정할 수도 있고, url 을 통해서 지정할 수 도 있다. 예를 들면

```
// 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>

// Using url fully qualified paths
<mappers>
  <mapper url="file:///var/sqlmaps/AuthorMapper.xml"/>
  <mapper url="file:///var/sqlmaps/BlogMapper.xml"/>
  <mapper url="file:///var/sqlmaps/PostMapper.xml"/>
</mappers>
```

SQL 매핑 파일에 대해서 좀더 자세히 알아보자.

## SQL Map XML 파일

MyBatis 의 가장 큰 장점은 매핑된 구문이다. 이건 간혹 마법을 부리는 것처럼 보일 수 있다. SQL Map XML 파일은 상대적으로 간단하다. 더군다나 동일한 기능의 JDBC 코드와 비교하면 아마도 95% 이상 코드수가 감소하기도 한다. MyBatis 는 SQL 을 작성하는데 집중하도록 만들어졌다.

SQL Map XML 파일은 첫번째(first class)요소만을 가진다.

- **cache** - 해당 명명공간을 위한 캐시 설정
- **cache-ref** - 다른 명명공간의 캐시 설정에 대한 참조
- **resultMap** - 데이터베이스 결과데이터를 객체에 로드하는 방법을 정의하는 요소
- ~~**parameterMap** - 비권장됨! 예전에 파라미터를 매핑하기 위해 사용되었으나 현재는 사용하지 않음~~
- **sql** - 다른 구문에서 재사용하기 위한 SQL 조각
- **insert** - 매핑된 INSERT 구문
- **update** - 매핑된 UPDATE 구문.
- **delete** - 매핑된 DELEETE 구문.
- **select** - 매핑된 SELECT 구문.

다음 섹션에서는 각각에 대해 좀더 세부적으로 살펴볼 것이다.

## select

select 구문은 MyBatis 에서 가장 흔히 사용할 요소이다. 데이터베이스에서 데이터를 가져온다. 아마도 대부분의 애플리케이션은 데이터를 수정하기보다는 조회하는 기능을 많이 가진다. 그래서 MyBatis 는 데이터를 조회하고 그 결과를 매핑하는데 집중하고 있다. Select 는 다음 예처럼 단순한 경우에는 단순하게 설정된다.

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

이 구문의 이름은 selectPerson 이고 int 타입의 파라미터를 가진다. 그리고 결과 데이터는 HashMap 에 저장된다. 파라미터 표기법을 보자.

```
#{id}
```

이 표기법은 MyBatis 에게 PreparedStatement 파라미터를 만들도록 지시한다. JDBC 를 사용할 때 PreparedStatement 에는 "?" 형태로 파라미터가 전달된다. 즉 결과적으로 위 설정은 아래와 같이 작동하게 되는 셈이다.

```
// Similar JDBC code, NOT MyBatis…
String selectPerson = "SELECT * FROM PERSON WHERE ID=?";
PreparedStatement ps = conn.prepareStatement(selectPerson);
ps.setInt(1,id);
```

물론 JDBC 를 사용하면, 결과를 가져와서 객체의 인스턴스에 매핑하기 위한 많은 코드가 필요하겠지만, MyBatis 는 그 코드들을 작성하지 않아도 되게 해준다.
select 요소는 각각의 구문이 처리하는 방식에 대해 좀더 세부적으로 설정하도록 많은 속성을 설정할 수 있다.

```
<select
        id="selectPerson"
        parameterType="int"
        parameterMap="deprecated"
        resultType="hashmap"
        resultMap="personResultMap"
        flushCache="false"
        useCache="true"
        timeout="10000"
        fetchSize="256"
        statementType="PREPARED"
        resultSetType="FORWARD_ONLY"
        >
```

| 속성 | 설명 |
|---|---|
| id | 구문을 찾기 위해 사용될 수 있는 명명공간내 유일한 구분자 |
| parameterType | 구문에 전달될 파라미터의 패키지 경로를 포함한 전체 클래스명이나 별칭 |
| parameterMap | 외부 parameterMap 을 찾기 위한 비권장된 접근방법. 인라인 파라미터 매핑과 parameterType 을 대신 사용하라. |

| 속성 | 설명 |
|---|---|
| resultType | 이 구문에 의해 리턴되는 기대타입의 패키지 경로를 포함한 전체 클래스명이나 별칭. collection 이 경우, collection 타입 자체가 아닌 collection 이 포함된 타입이 될 수 있다. resultType 이나 resultMap 을 사용하라. |
| resultMap | 외부 resultMap 의 참조명. 결과맵은 MyBatis 의 가장 강력한 기능이다. resultType 이나 resultMap 을 사용하라. |
| flushCache | 이 값을 true 로 셋팅하면, 구문이 호출될때마다 캐시가 지원질것이다(flush). 디폴트는 false 이다. |
| useCache | 이 값을 true 로 셋팅하면, 구문의 결과가 캐시될 것이다. 디폴트는 true 이다. |
| timeout | 예외가 던져지기 전에 데이터베이스의 요청 결과를 기다리는 최대시간을 설정한다. 디폴트는 셋팅하지 않는 것이고 드라이버에 따라 다소 지원되지 않을 수 있다. |
| fetchSize | 지정된 수만큼의 결과를 리턴하도록 하는 드라이버 힌트 형태의 값이다. 디폴트는 셋팅하지 않는 것이고 드라이버에 따라 다소 지원되지 않을 수 있다. |
| statementType | STATEMENT, PREPARED 또는 CALLABLE 중 하나를 선택할 수 있다. MyBatis 에게 Statement, PreparedStatement 또는 CallableStatement 를 사용하게 한다. 디폴트는 PREPARED 이다. |
| resultSetType | FORWARD_ONLY\|SCROLL_SENSITIVE\|SCROLL_INSENSITIVE 중 하나를 선택할 수 있다. 디폴트는 셋팅하지 않는 것이고 드라이버에 다라 다소 지원되지 않을 수 있다. |

insert, update, delete

데이터를 변경하는 구문인 insert, update, delete 는 매우 간단하다.

```
<insert
        id="insertAuthor"
        parameterType="domain.blog.Author"
        flushCache="true"
        statementType="PREPARED"
        keyProperty=""
        keyColumn=""
        useGeneratedKeys=""
        timeout="20000">

<update
        id="insertAuthor"
        parameterType="domain.blog.Author"
        flushCache="true"
        statementType="PREPARED"
        timeout="20000">

<delete
        id="insertAuthor"
        parameterType="domain.blog.Author"
        flushCache="true"
        statementType="PREPARED"
        timeout="20000">
```

| 속성 | 설명 |
|---|---|
| Id | 구문을 찾기 위해 사용될 수 있는 명명공간내 유일한 구분자 |
| parameterType | 구문에 전달될 파라미터의 패키지 경로를 포함한 전체 클래스명이나 별칭 |

| 속성 | 설명 |
|---|---|
| parameterMap | 외부 parameterMap 을 찾기 위한 비권장된 접근방법. 인라인 파라미터 매핑과 parameterType 을 대신 사용하라. |
| flushCache | 이 값을 true 로 셋팅하면, 구문이 호출될때마다 캐시가 지원질것이다(flush). 디폴트는 false 이다. |
| Timeout | 예외가 던져지기 전에 데이터베이스의 요청 결과를 기다리는 최대시간을 설정한다. 디폴트는 셋팅하지 않는 것이고 드라이버에 따라 다소 지원되지 않을 수 있다. |
| statementType | STATEMENT, PREPARED 또는 CALLABLE 중 하나를 선택할 수 있다. MyBatis 에게 Statement, PreparedStatement 또는 CallableStatement 를 사용하게 한다. 디폴트는 PREPARED 이다. |
| useGeneratedKeys | (입력(insert)에만 적용) 데이터베이스에서 내부적으로 생성한 키(예를 들어, MySQL 또는 SQL Server 와 같은 RDBMS 의 자동 증가 필드)를 받는 JDBC getGeneratedKeys 메서드를 사용하도록 설정하다. 디폴트는 false 이다. |
| keyProperty | (입력(insert)에만 적용) getGeneratedKeys 메서드나 insert 구문의 selectKey 하위 요소에 의해 리턴된 키를 셋팅할 프로퍼티를 지정. 디폴트는 셋팅하지 않는 것이다. |
| keyColumn | (입력(insert)에만 적용) 생성키를 가진 테이블의 칼럼명을 셋팅. 키 칼럼이 테이블이 첫번째 칼럼이 아닌 데이터베이스(PostgreSQL 처럼)에서만 필요하다. |

Insert, update, delete 구문의 예제이다.

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

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

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

앞서 설명했지만, insert 는 key 생성과 같은 기능을 위해 몇가지 추가 속성과 하위 요소를 가진다.
먼저, 사용하는 데이터베이스가 자동생성키(예를 들면, MySQL 과 SQL 서버)를 지원한다면, useGeneratedKeys="true" 로 설정하고 대상 프로퍼티에 keyProperty 를 셋팅할 수 있다. 예를 들어, Author 테이블이 id 칼럼에 자동생성키를 적용했다고 하면, 구문은 아래와 같은 형태일 것이다.

```
<insert id="insertAuthor" parameterType="domain.blog.Author"
        useGeneratedKeys="true" keyProperty="id">
  insert into Author (username,password,email,bio)
  values (#{username},#{password},#{email},#{bio})
</insert>
```

MyBatis 는 자동생성키 칼럼을 지원하지 않는 다른 데이터베이스를 위해 다른 방법 또한 제공한다.
이 예제는 랜덤 ID 를 생성하고 있다.

```
<insert id="insertAuthor" parameterType="domain.blog.Author">
  <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>
```

위 예제에서, selectKey 구문이 먼저 실행되고, Author id 프로퍼티에 셋팅된다. 그리고 나서 insert 구문이 실행된다. 이건 복잡한 자바코드 없이도 데이터베이스에 자동생성키의 행위와 비슷한 효과를 가지도록 해준다.
selectKey 요소는 다음처럼 설정가능하다.

```
<selectKey
        keyProperty="id"
        resultType="int"
        order="BEFORE"
        statementType="PREPARED">
```

| 속성 | 설명 |
|---|---|
| keyProperty | selectKey 구문의 결과가 셋팅될 대상 프로퍼티 |
| 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 | 위 내용과 같다. MyBatis 는 Statement, PreparedStatement 그리고 CallableStatement 을 매핑하기 위해 STATEMENT, PREPARED 그리고 CALLABLE 구문타입을 지원한다. |

## sql

이 요소는 다른 구문에서 재사용가능한 SQL 구문을 정의할 때 사용된다.

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

SQL 조각은 다른 구문에 포함시킬수 있다.

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

```
</select>
```

## Parameters

앞서 본 구문들에서, 간단한 파라미터들의 예를 보았을 것이다. Parameters 는 MyBatis 에서 매우 중요한 요소이다. 대략 90% 정도의 간단한 경우, 이러한 형태로 설정할 것이다.

```
<select id="selectUsers" parameterType="int" resultType="User">
        select id, username, password
        from users
        where id = #{id}
</select>
```

위 예제는 매우 간단한 명명된 파라미터 매핑을 보여준다. parameterType 은 "int" 로 설정되어 있다. Integer 과 String 과 같은 원시타입 이나 간단한 데이터 타입은 프로퍼티를 가지지 않는다. 그래서 파라미터 전체가 값을 대체하게 된다. 하지만 복잡한 객체를 전달하고자 한다면, 다음의 예제처럼 상황은 조금 다르게 된다.

```
<insert id="insertUser" parameterType="User" >
        insert into users (id, username, password)
        values (#{id}, #{username}, #{password})
</insert>
```

User 타입의 파라미터 객체가 구문으로 전달되면, id, username, password 프로퍼티는 찾아서 PreparedStatement 파라미터로 전달된다.
비록 파라미터들을 구문에 전달하는 괜찮은 예제이지만, 파라미터 매핑을 위한 다른 기능 또한 더 있다.

먼저, 파라미터에 데이터 타입을 명시할 수 있다.

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

javaType 은 파라미터 객체의 타입을 판단하는 기준이 된다. javaType 은 TypeHandler 를 사용하여 정의할 수도 있다.

➔ Note: 만약 특정 칼럼에 null 이 전달되면, JDBC Type 은 null 가능한 칼럼을 위해 필요하다. 처리 방법에 대해서는 PreparedStatement.setNull() 메서드의 JavaDoc 을 보라.

좀더 다양하게 타입 핸들링하기 위해서는, TypeHandler 클래스를 명시할 수 있다.

```
#{age,javaType=int,jdbcType=NUMERIC,typeHandler=MyTypeHandler}
```

숫자 타입을 위해서, 크기를 판단하기 위한 numericScale 속성도 있다.

```
#{height,javaType=double,jdbcType=NUMERIC,numericScale=2}
```

마지막으로, mode 속성은 IN, OUT 또는 INOUT 파라미터를 명시하기 위해 사용한다. 파라미터가 OUT 또는 INOUT 이라면, 파라미터의 실제 값은 변경될 것이다. mode=OUT(또는 INOUT) 이고 jdbcType=CURSOR(예를 들어, Oracle REFCURSOR) 라면, 파라미터의 타입에 ResultSet 를 매핑하기 위해 resultMap 을 명시해야만 한다.

```
#{department,
```

```
        mode=OUT,
        jdbcType=CURSOR,
        javaType=ResultSet,
        resultMap=departmentResultMap}
```

MyBatis 는 structs 와 같은 좀더 향상된 데이터 타입을 지원하지만, 파라미터를 등록할 때, 타입명을 구문에 전달해야 한다. 예를 들면,

```
#{middleInitial,
        mode=OUT,
        jdbcType=STRUCT,
        jdbcTypeName=MY_TYPE,
        resultMap=departmentResultMap}
```

이런 강력한 옵션들에도 불구하고, 대부분은 프로퍼티명만 명시하거나 null 가능한 칼럼을 위해 jdbcType 정도만 명시할 것이다.

```
#{firstName}
#{middleInitial,jdbcType=VARCHAR}
#{lastName}
```

### 문자열 대체(String Substitution)

#{} 문법은 MyBatis 로 하여금 PreparedStatement 프로퍼티를 만들어서 PreparedStatement 파라미터(예를 들면, ?)에 값을 셋팅하도록 할 것이다. 이 방법이 안전하기는 하지만, 좀더 빠른 방법이 선호되기도 한다. 가끔은 SQL 구문에 변하지 않는 값으로 삽입하길 원하기도 한다. 예를 들면, ORDER BY 와 같은 구문들이다.

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

여기서 MyBatis 는 문자열을 변경하거나 이스케이프 처리하지 않는다.

➔ 중요 : 사용자로부터 받은 값을 이 방법으로 변경하지 않고 구문에 전달하는 건 안전하지 않다. 이건 잠재적으로 SQL 주입 공격에 노출된다. 그러므로 사용자 입력값에 대해서는 이 방법을 사용하면 안된다. 사용자 입력값에 대해서는 언제나 자체적으로 이스케이프 처리하고 체크해야 한다.

## resultMap

resultMap 요소는 MyBatis 에서 가장 중요하고 강력한 요소이다. ResultSet 에서 데이터를 가져올때 작성되는 JDBC 코드를 대부분 줄여주는 역할을 담당한다. 사실 join 매핑과 같은 복잡한 코드는 굉장히 많은 코드가 필요하다. ResultMap 은 간단한 구문에서는 매핑이 필요하지 않고, 좀더 복잡한 구문에서 관계를 서술하기 위해 필요하다.

이미 앞에서 명시적인 resultMap 을 가지지 않는 간단한 매핑 구문은 봤을 것이다.

```
<select id="selectUsers" parameterType="int" resultType="hashmap">
        select id, username, hashedPassword
        from some_table
        where id = #{id}
```

```
</sql>
```

모든 칼럼의 값이 결과가 되는 간단한 구문에서는 HashMap 에서 키 형태로 자동으로 매핑된다. 하지만 대부분의 경우, HashMap 은 매우 좋은 도메인 모델이 되지는 못한다. 그래서 대부분 도메인 모델로는 자바빈이나 POJO 를 사용할 것이다. MyBatis 는 둘다 지원한다. 자바빈의 경우를 보자.

```
package com.someapp.model;
public class User {
  private int id;
  private String username;
  private String hashedPassword;

  public int getId() {
        return id;
  }
  public void setId(int id) {
        this.id = id;
  }
  public String getUsername() {
        return username;
  }
  public void setUsername(String username) {
        this.username = username;
  }
  public String getHashedPassword() {
        return hashedPassword;
  }
  public void setHashedPassword(String hashedPassword) {
        this.hashedPassword = hashedPassword;
  }
}
```

자바빈 스펙에 기반하여, 위 클래스는 3 개의 프로퍼티(id, username, hashedPassword)를 가진다. 이 프로퍼티는 select 구문에서 칼럼명과 정확히 일치한다.

그래서 자바빈은 HashMap 과 마찬가지로 매우 쉽게 ResultSet 에 매핑될 수 있다.

```
<select id="selectUsers" parameterType="int"
        resultType="com.someapp.model.User">
        select id, username, hashedPassword
        from some_table
        where id = #{id}
</sql>
```

그리고 TypeAliases 가 편리한 기능임을 기억해두는게 좋다. TypeAliases 를 사용하면 타이핑 수를 줄일 수 있다. 예를 들면,

```
<!-- In Config XML file -->
<typeAlias type="com.someapp.model.User" alias="User"/>
```

```
<!-- In SQL Mapping XML file -->
<select id="selectUsers" parameterType="int"
        resultType="User">
        select id, username, hashedPassword
        from some_table
        where id = #{id}
</sql>
```

이 경우 MyBatis 는 칼럼을 자바빈에 이름 기준으로 매핑하여 ResultMap 을 자동으로 생성할 것이다. 만약 칼럼명이 프로퍼티명과 다르다면, SQL 구문에 별칭을 지정할 수 있다. 예를 들면.

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

ResultMap 에 대한 중요한 내용은 다 보았다. 하지만 다 본건 아니다. 칼럼명과 프로퍼티명이 다른 경우에 대해 데이터베이스 별칭을 사용하는 것과 다른 방법으로 명시적인 resultMap 을 선언하는 방법이 있다.

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

구문에서는 resultMap 속성에 이를 지정하여 참조한다. 예를 들면.

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

## Advanced Result Mapping

이런 복잡한 구문은 어떻게 매핑할까.?

```
<!-- Very Complex Statement -->
<select id="selectBlogDetails" parameterType="int" resultMap="detailedBlogResultMap">
  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,
        A.favourite_section as author_favourite_section,
        P.id as post_id,
        P.blog_id as post_blog_id,
        P.author_id as post_author_id,
        P.created_on as post_created_on,
        P.section as post_section,
        P.subject as post_subject,
        P.draft as draft,
        P.body as post_body,
        C.id as comment_id,
        C.post_id as comment_post_id,
        C.name as comment_name,
        C.comment as comment_text,
        T.id as tag_id,
        T.name as tag_name
  from Blog B
        left outer join Author A on B.author_id = A.id
        left outer join Post P on B.id = P.blog_id
        left outer join Comment C on P.id = C.post_id
        left outer join Post_Tag PT on PT.post_id = P.id
        left outer join Tag T on PT.tag_id = T.id
  where B.id = #{id}
</select>
```

아마 Author 에 의해 작성되고 Comments 이나 태그를 가지는 많은 포스트를 가진 Blog 를 구성하는 괜찮은 객체 모델에 매핑하고 싶을 것이다. 이건 복잡한 ResultMap 으로 충분한 예제이다. 복잡해보이지만 단계별로 살펴보면 지극히 간단하다.

```
<!-- Very Complex Result Map -->
<resultMap id="detailedBlogResultMap" type="Blog">
  <constructor>
    <idArg column="blog_id" javaType="int"/>
  </constructor>
  <result property="title" column="blog_title"/>
  <association property="author" column="blog_author_id" javaType=" 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"/>
    <result property="favouriteSection" column="author_favourite_section"/>
  </association>
  <collection property="posts" ofType="Post">
```

```
  <id property="id" column="post_id"/>
  <result property="subject" column="post_subject"/>
  <association property="author" column="post_author_id" javaType="Author"/>
  <collection property="comments" column="post_id" ofType=" Comment">
   <id property="id" column="comment_id"/>
  </collection>
  <collection property="tags" column="post_id" ofType=" Tag" >
   <id property="id" column="tag_id"/>
  </collection>
  <discriminator javaType="int" column="draft">
   <case value="1" resultType="DraftPost"/>
  </discriminator>
 </collection>
</resultMap>
```

resultMap 요소는 많은 하위 요소를 가진다. 다음은 resultMap 요소의 개념적인 뷰(conceptual view)이다.

resultMap

- constructor - 인스턴스화되는 클래스의 생성자에 결과를 삽입하기 위해 사용됨
  - idArg - ID 인자 ; ID 와 같은 결과는 전반적으로 성능을 향상시킨다.
  - arg - 생성자에 삽입되는 일반적인 결과
- id - ID 결과; ID 와 같은 결과는 전반적으로 성능을 향상시킨다.
- result - 필드나 자바빈 프로퍼티에 삽입되는 일반적인 결과
- association - 복잡한 타입의 연관관계; 많은 결과는 타입으로 나타난다.
  - nested result mappings - resultMap 스스로의 연관관계
- collection - 복잡한 타입의 컬렉션
  - nested result mappings - resultMap 스스로의 연관관계
- discriminator - 사용할 resultMap 을 판단하기 위한 결과값을 사용
  - case - 몇가지 값에 기초한 결과 매핑
    - nested result mappings - a case is also a result map itself, and thus can contain many of these same elements, or it can refer to an external resultMap.

➔ 가장 좋은 형태 : 매번 ResultMap 을 추가해서 빌드한다. 이 경우 단위 테스트가 도움이 될 수 있다. 한번에 모든 resultMap 을 빌드하면, 작업하기 어려울 것이다. 간단히 시작해서 단계별로 처리하는 것이 좋다. 프레임워크를 사용하는 것은 종종 블랙박스와 같다. 가장 좋은 방법은 단위 테스트를 통해 기대하는 행위를 달성하는 것이다. 이건 버그가 발견되었을때 디버깅을 위해서도 좋은 방법이다.

다음 섹션은 각각의 요소에 대해 좀더 상세하게 살펴볼 것이다.

## id, result

```
<id property="id" column="post_id"/>
<result property="subject" column="post_subject"/>
```

이건 결과 매핑의 가장 기본적인 형태이다. id 와 result 모두 한개의 칼럼을 한개의 프로퍼티나 간단한 데이터 타입의 필드에 매핑한다.

둘 사이의 차이점은 id 값은 객체 인스턴스를 비교할 때 사용되는 구분자 프로퍼티로 처리되는 점이다. 이 점은 일반적으로 성능을 향상시키지만 특히 캐시와 내포된(nested) 결과 매핑(조인 매핑)의 경우에 더 그렇다.

둘다 다수의 속성을 가진다.

| 속성 | 설명 |
|---|---|
| property | 결과 칼럼에 매핑하기 위한 필드나 프로퍼티. 자바빈 프로퍼티가 해당 이름과 일치한다면, 그 프로퍼티가 사용될 것이다. 반면에 MyBatis 는 해당 이름이 필드를 찾을 것이다. 점 표기를 사용하여 복잡한 프로퍼티 검색을 사용할 수 있다. 예를 들어, “username”과 같이 간단하게 매핑될 수 있거나 “address.street.number” 처럼 좀더 복잡하게 매핑될수도 있다. |
| column | 데이터베이스의 칼럼명이나 별칭된 칼럼 라벨. resultSet.getString(columnName) 에 전달되는 같은 문자열이다. |
| javaType | 패키지 경로를 포함한 클래스 전체명이거나 타입 별칭. 자바빈을 사용한다면 MyBatis 는 타입을 찾아낼 수 있다. 반면에 HashMap 으로 매핑한다면, 기대하는 처리를 명확히 하기 위해 javaType 을 명시해야 한다. |
| 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. JDBC 로 직접 코딩을 하다보면, null 이 가능한 값에 이 타입을 지정할 필요가 있을 것이다. |
| typeHandler | 이 문서 앞에서 이미 타입 핸들러에 대해 설명했다. 이 프로퍼티를 사용하면, 디폴트 타입 핸들러를 오버라이드 할 수 있다. 이 값은 TypeHandler 구현체의 패키지를 포함한 전체 클래스명이나 타입 별칭이다. |

## 지원되는 JDBC 타입

좀더 상세한 설명전에, MyBatis 는 jdbcType 열거를 통해 다음의 JDBC 타입들을 지원한다.

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

## constructor

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

프로퍼티가 데이터 전송 객체(DTO) 타입 클래스로 작동한다. 변하지 않는 클래스를 사용하고자 하는 경우가 있다. 거의 변하지 않는 데이터를 가진 테이블은 종종 이 변하지 않는 클래스에 적합하다. 생성자 주입은 public 메서드가 없어도 인스턴스화할 때 값을 셋팅하도록 해준다. MyBatis 는 private 프로퍼티와 private 자바빈 프로퍼티를 지원하지만 많은 사람들은 생성자 주입을 선호한다. constructor 요소는 이러한 처리를 가능하게 한다.
다음의 생성자를 보자.

```
public class User {
   //…
```

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

결과를 생성자에 주입하기 위해, MyBatis 는 파라미터 타입에 일치하는 생성자를 찾을 필요가 있다. 자바는 파라미터 이름에서 타입을 체크할 방법이 없다. 그래서 constructor 요소를 생성할 때, 인자의 순서에 주의하고 데이터 타입을 명시해야 한다.

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

나머지 속성과 규칙은 id 와 result 요소와 동일하다.

| 속성 | 설명 |
|---|---|
| column | 데이터베이스의 칼럼명이나 별칭된 칼럼 라벨. resultSet.getString(columnName) 에 전달되는 같은 문자열이다. |
| javaType | 패키지 경로를 포함한 클래스 전체명이거나 타입 별칭. 자바빈을 사용한다면 MyBatis 는 타입을 찾아낼 수 있다. 반면에 HashMap 으로 매핑한다면, 기대하는 처리를 명확히 하기 위해 javaType 을 명시해야 한다. |
| 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. JDBC 로 직접 코딩을 하다보면, null 이 가능한 값에 이 타입을 지정할 필요가 있을 것이다. |
| typeHandler | 이 문서 앞에서 이미 타입 핸들러에 대해 설명했다. 이 프로퍼티를 사용하면, 디폴트 타입 핸들러를 오버라이드 할 수 있다. 이 값은 TypeHandler 구현체의 패키지를 포함한 전체 클래스명이나 타입 별칭이다. |
| select | 다른 매핑된 구문의 ID 는 이 프로퍼티 매핑이 필요로 하는 복잡한 타입을 로드할 것이다. column 속성의 칼럼으로 부터 가져온 값은 대상 select 구문에 파라미터로 전달될 것이다. 좀더 세부적인 설명은 association 요소를 보라. |
| resultMap | 이 인자의 내포된 결과를 적절한 객체로 매핑할 수 있는 ResultMap 의 ID 이다. 다른 select 구문을 호출하기 위한 대체방법이다. 여러개의 테이블을 조인하는 것을 하나의 ResultSet 으로 매핑하도록 해준다. ResultSet 은 사본을 포함할 수 있고, 데이터를 반복할 수도 있다. 가능하게 하기 위해서, 내포된 결과를 다루도록 결과맵을 "연결"하자. 좀더 자세히 알기 위해서는 association 요소를 보라. |

## association

```
<association property="author" column="blog_author_id" javaType=" Author">
        <id property="id" column="author_id"/>
        <result property="username" column="author_username"/>
</association>
```

association 요소는 "has-one" 타입의 관계를 다룬다. 예를 들어, Blog 는 하나의 Author 를 가진다. association 매핑은 다른 결과와 작동한다. 값을 가져오기 위해 대상 프로퍼티를 명시한다.

MyBatis 는 관계를 정의하는 두가지 방법을 제공한다.

- **내포된(Nested) Select**: 복잡한 타입을 리턴하는 다른 매핑된 SQL 구문을 실행하는 방법.
- **내포된(Nested) Results**: 조인된 결과물을 반복적으로 사용하여 내포된 결과 매핑을 사용하는 방법.

먼저 요소내 프로퍼티들을 보자. 보이는 것처럼, select 와 resultMap 속성만을 사용하는 간단한 결과 매핑과는 다르다.

| 속성 | 설명 |
|---|---|
| property | 결과 칼럼에 매핑하기 위한 필드나 프로퍼티. 자바빈 프로퍼티가 해당 이름과 일치한다면, 그 프로퍼티가 사용될 것이다. 반면에 MyBatis 는 해당 이름이 필드를 찾을 것이다. 점 표기를 사용하여 복잡한 프로퍼티 검색을 사용할 수 있다. 예를 들어, “username”과 같이 간단하게 매핑될 수 있거나 “address.street.number” 처럼 좀더 복잡하게 매핑될수도 있다. |
| column | 데이터베이스의 칼럼명이나 별칭된 칼럼 라벨. resultSet.getString(columnName) 에 전달되는 같은 문자열이다.<br>Note: 복합키를 다루기 위해서, column=”{prop1=col1,prop2=col2}” 문법을 사용해서 여러개의 칼럼명을 내포된 select 구문에 명시할 수 있다. 이것은 대상의 내포된 select 구문의 파라미터 객체에 prop1, prop2 형태로 셋팅하게 될 것이다. |
| javaType | 패키지 경로를 포함한 클래스 전체명이거나 타입 별칭. 자바빈을 사용한다면 MyBatis 는 타입을 찾아낼 수 있다. 반면에 HashMap 으로 매핑한다면, 기대하는 처리를 명확히 하기 위해 javaType 을 명시해야 한다. |
| 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. JDBC 로 직접 코딩을 하다보면, null 이 가능한 값에 이 타입을 지정할 필요가 있을 것이다. |
| typeHandler | 이 문서 앞에서 이미 타입 핸들러에 대해 설명했다. 이 프로퍼티를 사용하면, 디폴트 타입 핸들러를 오버라이드 할 수 있다. 이 값은 TypeHandler 구현체의 패키지를 포함한 전체 클래스명이나 타입 별칭이다. |

관계를 위한 내포된 select

| | |
|---|---|
| select | 다른 매핑된 구문의 ID 는 이 프로퍼티 매핑이 필요로 하는 복잡한 타입을 로드할 것이다.<br>column 속성의 칼럼으로 부터 가져온 값은 대상 select 구문에 파라미터로 전달될 것이다.<br>Note: 복합키를 다루기 위해서, column=”{prop1=col1,prop2=col2}” 문법을 사용해서 여러개의 칼럼명을 내포된 select 구문에 명시할 수 있다. 이것은 대상의 내포된 select 구문의 파라미터 객체에 prop1, prop2 형태로 셋팅하게 될 것이다. |

예를 들면.

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

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

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

```
</select>
```

여기엔 두개의 select 구문이 있다. 하나는 Blog 를 로드하고 다른 하나는 Author 를 로드한다. 그리고 Blog 의 resultMap 은 author 프로퍼티를 로드하기 위해 "selectAuthor" 구문을 사용한다.

다른 프로퍼티들은 칼럼과 프로퍼티명에 일치하는 것들로 자동으로 로드 될 것이다.

이 방법은 간단한 반면에, 큰 데이터나 목록에는 제대로 작동하지 않을 것이다. 이 방법은 "N+1 Selects 문제" 으로 알려진 문제점을 가진다. N+1 Selects 문제는 처리과정의 특이성으로 인해 야기된다.

- 레코드의 목록을 가져오기 위해 하나의 SQL 구문을 실행한다. ("+1" 에 해당).
- 리턴된 레코드별로, 각각의 상세 데이터를 로드하기 위해 select 구문을 실행한다. ("N" 에 해당).

이 문제는 수백  또는 수천의 SQL 구문 실행이라는 결과를 야기할 수 있다. 아마도 언제나 바라는 형태의 처리가 아닐 것이다.

목록을 로드하고 내포된 데이터에 접근하기 위해 즉시 반복적으로 처리한다면, 늦은 로딩으로 호출하고 게다가 성능은 많이 나빠질 것이다.

그래서 다른 방법이 있다.

## 관계를 위한 내포된 결과(Nested Results)

| | |
|---|---|
| resultMap | 이 인자의 내포된 결과를 적절한 객체로 매핑할 수 있는 ResultMap 의 ID 이다. 다른 select 구문을 호출하기 위한 대체방법이다. 여러개의 테이블을 조인하는 것을 하나의 ResultSet 으로 매핑하도록 해준다. ResultSet 은 사본을 포함할 수 있고, 데이터를 반복할 수도 있다. 가능하게 하기 위해서, 내포된 결과를 다루도록 결과맵을 "연결"하자. 좀더 자세히 알기 위해서는 association 요소를 보라. |

위에서 내포된 관계의 매우 복잡한 예제를 보았을 것이다. 다음은 작동하는 것을 보기 위한 좀더 간단한 예제이다. 개별 구문을 실행하는 것 대신에, Blog 와 Author 테이블을 함께 조인했다.

```
<select id="selectBlog" parameterType="int" 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>
```

조인을 사용할 때, 결과의 값들이 유일하거나 좀더 명확한 이름이 되도록 별칭을 사용하는 것이 좋다. 이제 결과를 매핑할 수 있다.

```
<resultMap id="blogResult" type="Blog">
```

```
  <id property="blog_id" column="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>
```

위 예제에서, Author 인스턴스를 로드하기 위한 "authorResult" resultMap 으로 위임된 Blog 의 "author" 관계를 볼 수 있을 것이다.

매우 중요 : id 요소는 내포된 결과 매핑에서 매우 중요한 역할을 담당한다. 결과 중 유일한 것을 찾아내기 위한 한개 이상의 프로퍼티를 명시해야만 한다. 가능하면 결과 중 유일한 것을 찾아낼 수 있는 프로퍼티들을 선택하라. 기본키가 가장 좋은 선택이 될 수 있다.

이제, 위 예제는 관계를 매핑하기 위해 외부의 resultMap 요소를 사용했다. 이 외부 resultMap 은 Author resultMap 을 재사용가능하도록 해준다. 어쨌든, 재사용할 필요가 있거나, 한개의 resultMap 에 결과 매핑을 함께 위치시키고자 한다면, association 결과 매핑을 내포시킬수 있다. 다음은 이 방법을 사용한 예제이다.

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

지금까지 "has one" 관계를 다루는 방법을 보았다. 하지만 "has many" 는 어떻게 처리할까? 그건 다음 섹션에서 다루어보자.

## collection

```
<collection property="posts" ofType="domain.blog.Post">
  <id property="id" column="post_id"/>
  <result property="subject" column="post_subject"/>
  <result property="body" column="post_body"/>
</collection>
```

collection 요소는 관계를 파악하기 위해 작동한다. 사실 이 내용이 중복되는 내용으로, 차이점에 대해서만 주로 살펴보자.

위 예제를 계속 진행하기 위해, Blog 는 오직 하나의 Author 를 가진다. 하지만 Blog 는 많은 Post 를 가진다. Blog 클래스에 다음처럼 처리될 것이다.

```
private List<Post> posts;
```

List 에 내포된 결과를 매핑하기 위해, collection 요소를 사용한다. association 요소와는 달리, 조인에서 내포된 select 나 내포된 결과를 사용할 수 있다.

## Collection 을 위한 내포된(Nested) Select

먼저, Blog 의 Post 를 로드하기 위한 내포된 select 를 사용해보자.

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

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

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

바로 눈치챌 수 있는 몇가지가 있지만, 대부분 앞서 배운 association 요소와 매우 유사하다. 먼저, collection 요소를 사용한 것이 보일 것이다. 그리고 나서 새로운 "ofType" 속성을 사용한 것을 알아차렸을 것이다. 이 속성은 자바빈 프로퍼티 타입과 collection 의 타입을 구분하기 위해 필요하다.

```
<collection property="posts" javaType="ArrayList" column="blog_id"
        ofType="Post" select="selectPostsForBlog"/>
```

➔ 독자 왈 : "Post 의 ArrayList 타입의 글 목록"

javaType 속성은 그다지 필요하지 않다. MyBatis 는 대부분의 경우 이 속성을 사용하지 않을 것이다. 그래서 좀더 간단하게 설정할 수 있다.

```
<collection property="posts" column="blog_id" ofType="Post"
        select="selectPostsForBlog"/>
```

## Collection 을 위한 내포된(Nested) Results

이 시점에, collection 을 위한 내포된 결과가 어떻게 작동하는지 짐작할 수 있을 것이다. 왜냐하면 association 와 정확히 일치하기 때문이다. 하지만 "***ofType***" 속성이 추가로 적용되었다.

먼저, SQL 을 보자.

```
<select id="selectBlog" parameterType="int" 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>
```

다시 보면, Blog 와 Post 테이블을 조인했고 간단한 매핑을 위해 칼럼명에 적절한 별칭을 주었다. 이제 Post 의 Collection 을 가진 Blog 의 매핑은 다음처럼 간단해졌다.

```
<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>
```

다시, 여기서 *id* 요소의 중요성을 기억해두거나 기억이 나지 않으면 *association* 섹션에서 다시 읽어둬라.

혹시, 결과 매핑의 재사용성을 위해 좀더 긴 형태를 선호한다면, 다음과 같은 형태로도 가능하다.

```
<resultMap id="blogResult" type="Blog">
  <id property="id" column="blog_id" />
  <result property="title" column="blog_title"/>
  <collection property="posts" ofType="Post" resultMap="blogPostResult"/>
</resultMap>

<resultMap id="blogPostResult" type="Post">
  <id property="id" column="post_id"/>
  <result property="subject" column="post_subject"/>
  <result property="body" column="post_body"/>
</resultMap>
```

➔ **Note:** associations 과 collections 에서 내포의 단계 혹은 조합에는 제한이 없다. 매핑할때는 성능을 생각해야 한다. 단위 테스트와 성능 테스트는 애플리케이션에서 가장 좋은 방법을 찾도록 지속해야 한다. MyBatis 는 이에 수정비용을 최대한 줄이도록 해줄 것이다.

## discriminator

```
<discriminator javaType="int" column="draft">
  <case value="1" resultType="DraftPost"/>
</discriminator>
```

종종 하나의 데이터베이스 쿼리는 많고 다양한 데이터 타입의 결과를 리턴한다. discriminator 요소는 클래스 상속 관계를 포함하여 이러한 상황을 위해 고안되었다. discriminator 는 자바의 switch 와 같이 작동하기 때문에 이해하기 쉽다.

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. 예를 들어

```
<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>
```

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 사용한다. carResult 가 다음처럼 정의된다면,

```
<resultMap id="carResult" type="Car">
        <result property=”doorCount” column="door_count" />
</resultMap>
```

doorCount 프로퍼티만이 로드될 것이다. discriminator 경우들의 독립적인 결과를 만들어준다. 이 경우 우리는 물론 car 와 vehicle 간의 관계를 알 수 있다. 그러므로, 나머지 프로퍼티들도 로드하길 원하게 된다. 그러기 위해서는 간단하게 하나만 변경하면 된다.

```
<resultMap id="carResult" type="Car" extends=”vehicleResult”>
        <result property=”doorCount” column="door_count" />
</resultMap>
```

vehicleResult 와 carResult 의 모든 프로퍼티들이 로드 될 것이다.

한가지 더, 도처에 설정된 외부 정의를 찾게 될지도 모른다. 그러므로 좀더 간결한 매핑 스타일의 문법이 있다. 예를 들면

```
<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" resultType="carResult">
            <result property=”doorCount” column="door_count" />
          </case>
          <case value="2" resultType="truckResult">
            <result property=”boxSize” column="box_size" />
            <result property=”extendedCab” column="extended_cab" />
          </case>
          <case value="3" resultType="vanResult">
            <result property=”powerSlidingDoor” column="power_sliding_door" />
          </case>
          <case value="4" resultType="suvResult">
            <result property=”allWheelDrive” column="all_wheel_drive" />
          </case>
        </discriminator>
</resultMap>
```

➔ 모든 결과 매핑이 있고, 모두 명시하고 싶지 않다면, MyBatis 는 칼럼과 프로퍼티 명으로 자동으로 매핑할 것이다. 이 예제는 실제로 필요한 내용보다 좀더 많이 서술되어 있다.

## cache

MyBatis 는 쉽게 설정가능하고 변경가능한 쿼리 캐싱 기능을 가지고 있다. MyBatis 3 캐시 구현체는 좀더 강력하고 쉽게 설정할 수 있도록 많은 부분이 수정되었다.

성능을 개선하고 순환하는 의존성을 해결하기 위해 필요한 로컬 세션 캐싱을 제외하고 기본적으로 캐시가 작동하지 않는다. 캐싱을 활성화하기 위해서, SQL 매핑 파일에 한줄을 추가하면 된다.

```
<cache/>
```

하나의 간단한 구문에 다음과 같은 순서로 영향을 준다.

- 매핑 구문 파일내 **select** 구문의 모든 결과가 캐시 될 것이다.
- 매핑 구문 파일내 **insert, update 그리고 delete** 구문은 캐시를 지울(flush) 것이다.

- 캐시는 **Least Recently Used (LRU)** 알고리즘을 사용할 것이다.
- 캐시는 스케줄링 기반으로 시간순서대로 지워지지는 않는다. (예를 들면. **no Flush Interval**)
- 캐시는 리스트나 객체에 대해 **1024 개의 참조**를 저장할 것이다. (쿼리 메서드가 실행될때마다)
- 캐시는 읽기/쓰기 캐시처럼 처리될 것이다. 이것은 가져올 객체는 공유되지 않고 호출자에 의해 안전하게 변경된다는 것을 의미한다.

모든 프로퍼티는 cache 요소의 속성을 통해 변경가능하다. 예를 들면:

```
〈cache
        eviction="FIFO"
        flushInterval="60000"
        size="512"
        readOnly="true"/〉
```

좀더 많은 프로퍼티가 셋팅된 이 설정은 60 초마다 캐시를 지우는 FIFO 캐시를 생성한다. 이 캐시는 결과 객체 또는 결과 리스트를 512 개까지 저장하고 각 객체는 읽기 전용이다. 캐시 데이터를 변경하는 것은 개별 쓰레드에서 호출자간의 충돌을 야기할 수 있다.

사용가능한 캐시 전략은 4 가지이다.

- **LRU** - Least Recently Used: 가장 오랜시간 사용하지 않는 객체를 제거
- **FIFO** - First In First Out: 캐시에 들어온 순서대로 객체를 제거
- **SOFT** - Soft Reference: 가비지 컬렉터의 상태와 강하지 않은 참조(Soft References )의 규칙에 기초하여 객체를 제거
- **WEAK** - Weak Reference: 가비지 컬렉터의 상태와 약한 참조(Weak References)의 규칙에 기초하여 점진적으로 객체 제거

디폴트 값은 LRU 이다.

**flushInterval** 은 양수로 셋팅할 수 있고, 밀리세컨드로 명시되어야 한다. 디폴트는 셋팅되지 않으나, 플러시(flush) 주기를 사용하지 않으면, 캐시는 오직 구문이 호출될때마다 캐시를 지운다.

**size** 는 양수로 셋팅할 수 있고 캐시에 객체의 크기를 유지하지만 메모리 자원이 충분해야 한다. 디폴트 값은 1024 이다.

**readOnly** 속성은 true 또는 false 로 설정 할 수 있다. 읽기 전용 캐시는 모든 호출자에게 캐시된 객체의 같은 인스턴스를 리턴 할 것이다. 게다가 그 객체는 변경할 수 없다. 이건 종종 성능에 잇점을 준다. 읽고 쓰는 캐시는 캐시된 객체의 복사본을 리턴 할 것이다. 이건 조금 더 늦긴 하지만 안전하다. 디폴트는 false 이다..

## 사용자 지정 캐시 사용하기

앞서 본 다양한 설정방법에 더해, 자체적으로 개발하거나 써드파티 캐시 솔루션을 사용하여 캐시 처리를 할 수 있다.

```
<cache type="com.domain.something.MyCustomCache"/>
```

이 예제는 사용자 지정 캐시 구현체를 사용하는 방법을 보여준다. type 속성에 명시된 클래스는 org.mybatis.cache.Cache 인터페이스를 반드시 구현해야 한다. 이 인터페이스는 MyBatis 프레임워크의 가장 복잡한 구성요소 중 하나이다. 하지만 하는 일은 간단하다.

```
public interface Cache {
  String getId();
  int getSize();
  void putObject(Object key, Object value);
  Object getObject(Object key);
  boolean hasKey(Object key);
  Object removeObject(Object key);
  void clear();
  ReadWriteLock getReadWriteLock();
}
```

캐시를 설정하기 위해, 캐시 구현체에 public 자바빈 프로퍼티를 추가하고 cache 요소를 통해 프로퍼티를 전달한다. 예를 들어, 다음 예제는 캐시 구현체에서 "setCacheFile(String file)" 를 호출하여 메서드를 호출할 것이다.

```
<cache type="com.domain.something.MyCustomCache">
  <property name="cacheFile" value="/tmp/my-custom-cache.tmp"/>
</cache>
```

모든 간단한 타입의 자바빈 프로퍼티를 사용할 수 있다. MyBatis 는 변환할 것이다.

캐시 설정과 캐시 인스턴스가 SQL Map 파일의 명명공간에 묶여지는(bound) 것을 기억하는게 중요하다. 게다가, 같은 명명공간내 모든 구문은 묶여진다. 구문별로 캐시와 상호작동하는 방법을 변경할 수 있거나 두개의 간단한 속성을 통해 완전히 배제될 수 도 있다. 디폴트로 구문은 아래와 같이 설정된다.

```
<select ... flushCache="false" useCache="true"/>
<insert ... flushCache="true"/>
<update ... flushCache="true"/>
<delete ... flushCache="true"/>
```

이러한 방법으로 구문을 설정하는 하는 방법은 굉장히 좋지 않다. 대신 디폴트 행위를 변경해야 할 경우에만 flushCache 와 useCache 속성을 변경하는 것이 좋다. 예를 들어, 캐시에서 특정 select 구문의 결과를 제외하고 싶거나 캐시를 지우기 위한 select 구문을 원할 것이다. 유사하게 실행할때마다 캐시를 지울 필요가 없는 update 구문도 있을 것이다.

### cache-ref

이전 섹션 내용을 돌이켜보면서, 특정 명명공간을 위한 캐시는 오직 하나만 사용되거나 같은 명명공간내에서는 구문마다 캐시를 지울수 있다. 명명공간간의 캐시 설정과 인스턴스를 공유하고자 할때가 있을 것이다. 이 경우 cache-ref 요소를 사용해서 다른 캐시를 참조할 수 있다.

```
<cache-ref namespace="com.someone.application.data.SomeMapper"/>
```

## 동적 SQL

MyBatis 의 가장 강력한 기능 중 하나는 동적 SQL 기능이다. JDBC 나 다른 유사한 프레임워크를 사용해본 경험이 있다면, 동적으로 SQL 을 구성하는 것이 얼마나 힘든 작업인지 이해할 것이다. 간혹 공백이나 콤마를 붙이는 것을 잊어본 적도 있을 것이다. 동적 SQL 은 그만큼 어려운 것이다.

동적 SQL 을 사용하는 것은 결코 파티가 될 수 없을 것이다. MyBatis 는 강력한 동적 SQL 언어로 이 상황은 개선한다.

동적 SQL 요소들은 JSTL 이나 XML 기반의 텍스트 프로세서를 사용해 본 사람에게는 친숙할 것이다. MyBatis 의 이전 버전에서는, 알고 이해해야 할 요소가 많았다. MyBatis 3 에서는 이를 크게 개선했고 실제 사용해야 할 요소가 반 이하로 줄었다. MyBatis 다른 요소의 사용을 최대한 제거하기 위해 OGNL 기반의 표현식을 가져왔다.

- if
- choose (when, otherwise)
- trim (where, set)
- foreach

### if

동적 SQL 에서 가장 공통적으로 사용되는 것으로 where 의 일부로 포함될 수 있다. 예를 들면:

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

이 구문은 선택적으로 문자열 검색 기능을 제공할 것이다. 만약에 title 값이 없다면, 모든 active 상태의 Blog 가 리턴될 것이다. 하지만 title 값이 있다면, 그 값과 비슷한 데이터를 찾게 될 것이다.
title 과 author 을 사용하여 검색하고 싶다면? 먼저, 의미가 좀더 잘 전달되도록 구문의 이름을 변경할 것이다. 그리고 다른 조건을 추가한다.

```
<select id="findActiveBlogLike"
        parameterType="Blog" resultType="Blog">
```

```
  SELECT * FROM BLOG WHERE state = 'ACTIVE'
  <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>
```

## choose, when, otherwise

우리는 종종 적용 할 모든 조건을 원하는 대신에 한가지 경우만을 원할 수 있다. 자바에서는 switch 구문과 유사하며, MyBatis 에서는 choose 요소를 제공한다.

위 예제를 다시 사용해보자. 지금은 title 만으로 검색하고 author 가 있다면 그 값으로 검색된다. 둘다 제공하지 않는다면, featured 상태의 blog 가 리턴된다.

```
<select id="findActiveBlogLike"
        parameterType="Blog" 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

앞서 예제는 악명높게 다양한 요소가 사용된 동적 SQL 이다. "if" 예제를 사용해보자.

```
<select id="findActiveBlogLike"
        parameterType="Blog" 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>
```

어떤 조건에도 해당되지 않는다면 어떤 일이 벌어질까? 아마도 다음과 같은 SQL 이 만들어질 것이다.

```
SELECT * FROM BLOG
WHERE
```

아마도 이건 실패할 것이다. 두번째 조건에만 해당된다면 무슨 일이 벌어질까? 아마도 다음과 같은 SQL 이 만들어질 것이다.

```
SELECT * FROM BLOG
WHERE
AND title like 'someTitle'
```

이것도 아마 실패할 것이다. 이 문제는 조건만 가지고는 해결되지 않았다. 이렇게 작성했다면, 다시는 이렇게 작성하지 않게 될 것이다.
실패하지 않기 위해서 조금 수정해야 한다. 조금 수정하면 아마도 다음과 같을 것이다:

```
<select id="findActiveBlogLike"
        parameterType="Blog" 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>
```

where 요소는 태그에 의해 컨텐츠가 리턴되면 단순히 “WHERE” 만을 추가한다. 게다가, 컨텐츠가 “AND” 나 “OR” 로 시작한다면, 그 “AND” 나 “OR”를 지워버린다.

만약에 where 요소가 기대한 것처럼 작동하지 않는다면, trim 요소를 사용자 정의할 수도 있다. 예를 들어, 다음은 where 요소에 대한 trim 기능과 동일하다.:

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

override 속성은 오버라이드하는 텍스트의 목록을 제한한다. 결과는 override 속성에 명시된 것들을 지우고 with 속성에 명시된 것을 추가한다.

다음 예제는 동적인 update 구문의 유사한 경우이다. set 요소는 update 하고자 하는 칼럼을 동적으로 포함시키기 위해 사용될 수 있다. 예를 들어:

```
<update id="updateAuthorIfNecessary"
        parameterType="domain.blog.Author">
  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>
```

여기서 set 요소는 동적으로 SET 키워드를 붙히고, 필요없는 콤마를 제거한다.

아마도 trim 요소로 처리한다면 아래와 같을 것이다.

```
<trim prefix="SET" suffixOverrides=",">
...
</trim>
```

이 경우, 접두사는 추가하고, 접미사를 오버라이딩 한다.

## foreach

동적 SQL 에서 공통적으로 필요한 것은 collection 에 대해 반복처리를 하는 것이다. 종종 IN 조건을 사용하게 된다. 예를 들면,

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

foreach 요소는 매우 강력하고 collection 을 명시하는 것을 허용한다. 요소 내부에서 사용할 수 있는 item, index 두가지 변수를 선언한다. 이 요소는 또한 열고 닫는 문자열로 명시할 수 있고 반복간에 둘 수 있는 구분자도 추가할 수 있다.

⇨ Note: 파라미터 객체로 MyBatis 에 List 인스턴스나 배열을 전달 할 수 있다. 그렇게 하면 MyBatis 는 Map 으로 자동으로 감싸고 이름을 키로 사용한다. List 인스턴스는 “list” 를 키로 사용하고, 배열 인스턴스는 “array” 를 키로 사용한다.

XML 설정 파일과 XML 매핑 파일에 대해서는 이 정도에서 정리하고, 다음 섹션에서는 Java API 에 대해 좀더 상세하게 살펴볼 것이다.

# 자바 API

이제 MyBatis 를 설정하는 방법과 매핑을 만드는 방법을 알게 되었다. 이미 충분히 잘 사용할 준비가 된 셈이다. MyBatis 자바 API 는 당신의 노력에 대한 보상을 얻게 할 것이다. JDBC 와 비교해보면, MyBatis 는 코드를 굉장히 단순하게 만들고 깔끔하게 만든다. 이해하기 쉬워서 유지보수도 편하게 해준다. MyBatis 3 은 SQL Map 을 사용하는 많은 수의 개선 내용을 소개했다.

## 디렉터리 구조

자바 API 를 살펴보기 전에, 디렉터리 구조에 대해 전반적으로 이해하는 것이 중요하다. MyBatis 는 매우 유연하고 파일을 사용해서 어떤 것도 할 수 있다. 하지만 프레임워크이기 때문에 선호하는 방법이 있다.

전형적인 애플리케이션 디렉터리 구조를 살펴보자.

```
/my_application
        /bin
        /devlib
        /lib                                    ← MyBatis *.jar 파일이 여기 있다.
        /src
                /org/myapp/
                        /action                 ← MyBatis 산출물이 여기 있다. Mapper
                        /data                     클래스, XML 설정, XML 매핑 파일들
                                /SqlMapConfig.xml
                                /BlogMapper.java
                                /BlogMapper.xml ← XML 설정파일에 포함된 프로퍼티들이 여기
                                                  있다.
                        /model
                        /service
                        /view
                /properties
        /test                                   기억해달라. 이건 선호하는 것이지 필수요건이
                /org/myapp/                     아니다. 하지만 다른 이들은 공통적인
                        /action                 디렉터리를 구조를 사용하면 좋아할 것이다.
                        /data
                        /model
                        /service
                        /view
                /properties
        /web
                /WEB-INF
```

/web.xml

이 섹션의 나머지 예제는 디렉터리 구조가 이렇게 되어 있다고 가정하고 설명한다.

## SqlSessions

MyBatis 를 사용하기 위한 기본적인 자바 인터페이스는 SqlSession 이다. 이 인터페이스를 통해 명령어를 실행하고, 매퍼를 얻으며 트랜잭션을 관리 할 수 있다. 우리는 SqlSession 에 대해서 좀더 얘기해볼 것이지만 먼저 SqlSession 의 인스턴스를 만드는 방법을 배워보자. SqlSession 은 SqlSessionFactory 인스턴스를 사용해서 만든다. SqlSessionFactory 는 몇가지 방법으로 SqlSession 인스턴스를 생성하기 위한 메서드를 포함하고 있다. SqlSessionFactory 자체는 XML, 애노테이션 또는 자바 설정에서 SqlSessonFactory 를 생성할 수 있는 SqlSessionFactoryBuilder 를 통해 만들어진다.

## SqlSessionFactoryBuilder

SqlSessionFactoryBuilder 는 5 개의 build() 메서드를 가진다. 각각은 서로 다른 소스에서 SqlSession 을 빌드한다.

```
SqlSessionFactory build(Reader reader)
SqlSessionFactory build(Reader reader, String environment)
SqlSessionFactory build(Reader reader, Properties properties)
SqlSessionFactory build(Reader reader, String env, Properties props)
SqlSessionFactory build(Configuration config)
```

처음 4 개의 메서드가 가장 공통적이다. XML 문서를 나타내는 Reader 인스턴스를 가진다. SqlMapConfig.xml 파일은 위에서 다루었다. 선택적으로 사용가능한 프로퍼티는 environment 와 properties 이다. environment 는 데이터소스와 트랜잭션 관리자를 포함하여 로드할 환경을 판단한다. 예를 들면:

```
<environments default="development">
  <environment id="development">
   <transactionManager type="JDBC">
     …
        <dataSource type="POOLED">
     …
  </environment>
  <environment id="production">
   <transactionManager type="EXTERNAL">
     …
        <dataSource type="JNDI">
     …
  </environment>
</environments>
```

environment  파라미터를 가진 메서드를 호출한다면, MyBatis 는 사용할 환경을 위한 설정을 사용할 것이다. 물론 잘못된 환경설정을 사용하면, 에러를 보게 될 것이다. environment 파라미터를 가지지 않는 메서드 중 하나를 호출한다면, 디폴트 환경(위 예제에서 default=“development”)이 사용될 것이다.

properties 인스턴스를 가진 메서드를 호출하면, MyBatis 는 프로퍼티들을 로드해서 설정에서 사용가능한 부분을 사용할 것이다. 프로퍼티들은 ${propName} 와 같은 문법을 사용해서 설정의 값으로 대체될 수 있다.

프로퍼티들은 SqlMapConfig.xml 파일에서 사용되거나 직접 명시할 수 있다. 그러므로 프로퍼티들의 우선순위를 이해하는 것이 중요하다. 우리는 앞서 언급하긴 했지만, 쉽게 이해할 수 있도록 다시 보여주도록 하겠다.

---

프로퍼티가 한개 이상 존재한다면, MyBatis 는 일정한 순서로 로드한다.:

- properties 요소에 명시된 속성을 가장 먼저 읽는다.
- properties 요소의 클래스패스 자원이나 url 속성으로 부터 로드된 속성을 두번재로 읽는다. 그래서 이미 읽은 값이 있다면 덮어쓴다.,
- 마지막으로 메서드 파라미터로 전달된 속성을 읽는다. 앞서 로드된 값을 덮어쓴다

그래서 가장 우선순위가 높은 속성은 메서드의 파라미터로 전달된 값이고 그 다음은 자원및 url 속성이고 마지막은 properties 요소에 명시된 값이다.

---

요약해보면, 처음 4 개의 메서드는 사실 같지만, environment 그리고/또는 properties 에 명시한 값을 오버라이드한다. SqlMapConfig.xml 파일에서 SqlSessionFactory 를 빌드하는 예제이다.

```
String resource = "org/mybatis/builder/MapperConfig.xml";
Reader reader = Resources.getResourceAsReader(resource);
SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
SqlSessionFactory factory = builder.build(reader);
```

Resources 유틸리티 클래스를 사용하고 있는 것을 주의깊게 보면 된다. 이 클래스는 org.mybatis.io 패키지에 있다. Resources 클래스는 그 이름이 나타내는 것처럼, 클래스패스나 파일 시스템 또는 웹 URL 에서 자원으로 로드하도록 해준다. IDE 를 통해 클래스의 소스 코드를 보는 것으로 유용한 메서드를 보게 될 것이다. 그 유용한 메서드 목록들이다.

```
URL getResourceURL(String resource)
URL getResourceURL(ClassLoader loader, String resource)
InputStream getResourceAsStream(String resource)
InputStream getResourceAsStream(ClassLoader loader, String resource)
Properties getResourceAsProperties(String resource)
Properties getResourceAsProperties(ClassLoader loader, String resource)
Reader getResourceAsReader(String resource)
Reader getResourceAsReader(ClassLoader loader, String resource)
File getResourceAsFile(String resource)
File getResourceAsFile(ClassLoader loader, String resource)
InputStream getUrlAsStream(String urlString)
Reader getUrlAsReader(String urlString)
Properties getUrlAsProperties(String urlString)
Class classForName(String className)
```

마지막 build 메서드는 Configuration 의 인스턴스를 가진다. Configuration 클래스는 SqlSessionFactory 인스턴스에 대해 알 필요가 있는 모든 것으로 가지고 있다. Configuration 클래스는 SQL Map 을 찾거나 관리하는 것을 포함하여 설정을 살펴보기 위해 유용하다. Configuration 클래스는 앞서 봤던 모든 설정을 처리할 수 있으며, 자바 API 로 나타낼 수 있다. SqlSessionFactory 를 생성하기 위해 Configuration 인스턴스를 build() 메서드에 전달하는 예제이다.

```
DataSource dataSource = BaseDataTest.createBlogDataSource();
TransactionFactory transactionFactory = new JdbcTransactionFactory();

Environment environment =
        new Environment("development", transactionFactory, dataSource);

Configuration configuration = new Configuration(environment);
configuration.setLazyLoadingEnabled(true);
configuration.setEnhancementEnabled(true);
configuration.getTypeAliasRegistry().registerAlias(Blog.class);
configuration.getTypeAliasRegistry().registerAlias(Post.class);
configuration.getTypeAliasRegistry().registerAlias(Author.class);
configuration.addMapper(BoundBlogMapper.class);
configuration.addMapper(BoundAuthorMapper.class);

SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
SqlSessionFactory factory = builder.build(configuration);
```

이제 SqlSessionFactory 를 만들었다. SqlSession 인스턴스를 만들기 위해 사용해보자.

## SqlSessionFactory

SqlSessionFactory 는 SqlSession 인스턴스를 생성하기 위해 사용할 수 있는 6 개의 메서드를 가지고 있다. 6 개의 메서드가 선택해서 사용하는 것들을 보자.

- **Transaction:** 세션에서 트랜잭션 스코프 또는 자동 커밋을 사용하고 싶은가?
- **Connection:** 설정된 DataSource 에서 Connection 을 회득하고 싶은가?
- **Execution:** PreparedStatements 그리고/또는 배치(insert, delete 를 포함해서) 업데이트를 재사용하고 싶은가?

오버로드된 메서드인 penSession() 이 3 가지를 적절히 혼합해서 사용할 수 있다.

```
SqlSession openSession()
SqlSession openSession(boolean autoCommit)
SqlSession openSession(Connection connection)
SqlSession openSession(TransactionIsolationLevel level)
SqlSession openSession(ExecutorType execType,TransactionIsolationLevel level)
SqlSession openSession(ExecutorType execType)
SqlSession openSession(ExecutorType execType, boolean autoCommit)
SqlSession openSession(ExecutorType execType, Connection connection)
Configuration getConfiguration();
```

파라미터를 가지지 않는 디폴트 openSession() 메서드는 다음과 같은 성격을 가진 SqlSession 을 만들것이다.

- 트랜잭션 스코프는 시작될 것이다.
- Connection 객체는 활성화된 환경에 의해 설정된 DataSource 인스턴스를 획득할 것이다.
- 트랜잭션 격리 레벨은 드라이버나 데이터소스가 디폴트로 제공하는 옵션을 사용할 것이다.
- PreparedStatements 는 재사용되지 않을 것이다. 그리고 update 또한 배치처리되지 않을 것이다.

메서드 대부분은 그 이름과 파라미터가 그 역할을 충분히 설명한다. 자동커밋을 활성화하기 위해서, autoCommit 파라미터에 “true” 값을 셋팅하라. 자체적인 커넥션을 제공하기 위해서는, connection 파라미터에 Connection 인스턴스를 셋팅하라.

Connection 과 autoCommit 둘다 셋팅하는 것을 오버라이드하지 않는다. 왜냐하면 MyBatis 는 제공된 connection 객체를 셋팅할때마다 현재 사용중인 것을 사용한다. MyBatis 는 TransactionIsolationLevel 라고 불리는 트랜잭션 격리 레벨을 위한 자바 enum 래퍼를 사용한다. JDBC 를 5 가지를 지원한다(NONE, READ_UNCOMMITTED, READ_COMMITTED, REPEATABLE_READ, SERIALIZABLE).

새롭게 보일수 있는 하나의 파라미터는 ExecutorType 이다. enum 으로는 3 개의 값을 정의한다.

**ExecutorType.SIMPLE**
이 타입의 실행자는 아무것도 하지 않는다. 구문 실행마다 새로운 PreparedStatement 를 생성한다.

**ExecutorType.REUSE**

이 타입의 실행자는 PreparedStatements 를 재사용할 것이다.

**ExecutorType.BATCH**
이 실행자는 모든 update 구문을 배치처리하고 중간에 select 가 실행될 경우 필요하다면 경계를 표시한다. 이러한 과정은 행위를 좀더 이해하기 쉽게 하기 위함이다.

➔ Note: SqlSessionFactory 에서 언급하지 않는 한개 이상의 메서드가 있다. getConfiguration() 메서드인데, 이 메서드는 런타임시 MyBatis 설정을 조사하는 Configuration 인스턴스를 리턴할것이다.

## SqlSession

앞서 언급한 것처럼, SqlSession 인스턴스는 MyBatis 에서 굉장히 강력한 클래스이다. 구문을 실행하고, 트랜잭션을 커밋하거나 롤백하는 그리고 mapper 인스턴스를 습득하기 위해 필요한 모든 메서드를 찾을 수 있을 것이다.

SqlSession 에는 20 개 이상의 메서드가 있다. 좀더 적절히 모아서 보도록 하자.

### *구문을 실행하는 메서드*

이 메서드들은 SQL 매핑 XML 파일에 정의된 SELECT, INSERT, UPDATE 그리고 DELETE 구문을 실행하기 위해 사용된다. 메서드 이름 자체가 그 역할을 설명하도록 명명되었다. 메서드 각각은 구문의 ID 와 파라미터 객체(원시타입, 자바빈, POJO 또는 Map)을 가진다.

```
Object selectOne(String statement, Object parameter)
List selectList(String statement, Object parameter)
Map selectMap(String statement, Object parameter, String mapKey)
int insert(String statement, Object parameter)
int update(String statement, Object parameter)
int delete(String statement, Object parameter)
```

selectOne 과 selectList 의 차이점은 selectOne 메서드는 오직 하나의 객체만을 리턴해야 한다는 것이다. 한개 이상을 리턴하거나 null 이 리턴된다면, exception 이 발생할 것이다. 얼마나 많은 객체가 리턴될지 모른다면, selectList 를 사용하라. 객체의 존재여부를 체크하고 싶다면, 개수를 리턴하는 방법이 더 좋다. selectMap 은 결과 목록을 Map 을 변환하기 위해 디자인된 특별한 경우이다. 이 경우 결과 객체의 프로퍼티 중 하나를 키로 사용하게 된다. 모든 구문이 파라미터를 필요로 하지는 않기 때문에, 파라미터 객체를 요구하지 않는 형태로 오버로드되었다.

```
Object selectOne(String statement)
List selectList(String statement)
Map selectMap(String statement, String mapKey)
int insert(String statement)
int update(String statement)
int delete(String statement)
```

마지막으로, 리턴되는 데이터의 범위를 제한하거나 결과를 핸들링 하는 로직을 부여할 수 있는 3 개의 select 메서드가 있다.

```
List selectList
        (String statement, Object parameter, RowBounds rowBounds)
Map selectMap(String statement, Object parameter, String mapKey,
                RowBounds rowbounds)
void select
        (String statement, Object parameter, ResultHandler handler)
void select
        (String statement, Object parameter, RowBounds rowBounds,
                ResultHandler handler)
```

RowBounds 파라미터는 MyBatis 로 하여금 특정 개수 만큼의 레코드를 건너띄게 한다. RowBounds 클래스는 offset 과 limit 둘다 가지는 생성자가 있다.

```
int offset = 100;
int limit = 25;
RowBounds rowBounds = new RowBounds(offset, limit);
```

가장 좋은 성능을 위해, 결과셋의 타입을 SCROLL_SENSITIVE 나 SCROLL_INSENSITIVE 로 사용하라.

ResultHandler 파라미터는 레코드별로 다룰수 있도록 해준다. List 에 추가할수도 있고, Map, Set 을 만들수도 있으며, 각각의 결과를 그냥 던질수도 있다. ResultHandler 로 많은 것을 할 수 있고 MyBatis 는 결과셋을 다루기 위해 내부적으로 사용한다.

인터페이스는 매우 간단하다.

```
package org.mybatis.executor.result;
public interface ResultHandler {
  void handleResult(ResultContext context);
}
```

ResultContext 파라미터는 결과 객체에 접근할 수 있도록 해준다.

## *트랙잭션 제어 메서드*

트랜잭션을 제어하기 위해 4 개의 메서드가 있다. 물론 자동커밋을 선택하였거나 외부 트랜잭션 관리자를 사용하면 영향이 없다. 어쨌든, Connection 인스턴스에 의해 관리되고 JDBC 트랜잭션 관리자를 사용하면, 이 4 개의 메서드를 사용할 수 있다.

```
void commit()
void commit(boolean force)
void rollback()
void rollback(boolean force)
```

기본적으로 MyBatis 는 insert, update 또는 delete 를 호출하여 데이터베이스가 변경된 것으로 감지하지 않는 한 실제로 커밋하지 않는다. 이러한 메서드 호출없이 변경되면, 커밋된 것으로 보장하기 위해 commit 와 rollback 메서드에 true 값을 전달한다.

## *세션 레벨의 캐시를 지우기*

```
void clearCache()
```

SqlSession 인스턴스는 update, commit, rollback 또는 close 할때마다 지워지는 로컬 캐시이다. 명시적으로 닫기 위해서는, clearCache()메서드를 호출할 수 있다.

## *SqlSession 을 반드시 닫도록 한다.*

```
void close()
```

반드시 기억해야 하는 중요한 것은 당신이 열었던 세션을 닫아주는 것이다. 확실히 하기 위해 가장 좋은 방법은 다음과 같은 형태로 개발하는 것이다.

```
SqlSession session = sqlSessionFactory.openSession();
try {
   // following 3 lines pseudocod for “doing some work”
   session.insert(…);
   session.update(…);
   session.delete(…);
   session.commit();
```

```
} finally {
   session.close();
}
```

➔ Note: SqlSessionFactory 처럼, SqlSession 이 getConfiguration() 메서드를 호출하여 Configuration 인스턴스를 얻을 수 있다.

```
Configuration getConfiguration()
```

## *Mappers 사용하기*

```
<T> T getMapper(Class<T> type)
```

다양한 insert, update, delete 그리고 select 메서드는 강력하지만, 다소 장황하고 타입에 안전하지 않다. 더군다나 IDE 나 단위 테스트에 그다지 도움이 되지 않는 형태이다. 우리는 Mapper 를 사용하는 예제는 이미 시작하기 섹션에서 봤다.

그러므로, 매핑된 구문을 실행하기 위해 좀더 공통적인 방법은 Mapper 클래스를 사용하는 것이다. Mapper 클래스는 SqlSession 메서드에 일치하는 메서드와 간단히 연동된다. 다음의 예제 클래스는 몇가지 메서드 시그니처와 SqlSession 에 매핑하는 방법을 보여준다.

```
public interface AuthorMapper {
  // (Author) selectOne("selectAuthor",5);
  Author selectAuthor(int id);
  // (List<Author>) selectList("selectAuthors")
  List<Author> selectAuthors();


  // (Map<Integer,Author>) selectMap("selectAuthors", "id")
  @MapKey("id")
  List<Author> selectAuthorsAsMap();
  // insert("insertAuthor", author)
  int insertAuthor(Author author);
  // updateAuthor("updateAuthor", author)
  int updateAuthor(Author author);
  // delete("deleteAuthor",5)
  int deleteAuthor(int id);
}
```

아주 간결하게, 각각의 Mapper 메서드 시그니처는 SqlSession 의 메서드 시그니처와 일치해야만 한다. String 파라미터 ID 가 없지만, 대신 메서드명은 매핑된 구문의 ID 와 같아야 한다.

추가로, 리턴 타입은 기대하는 결과 타입과 일치해야만 한다. 원시타입과 Map, POJO 그리고 자바빈등 대부분의 타입이 지원된다.

➔ Mapper 인터페이스는 어떠한 인터페이스를 구현할 필요가 없고 어떠한 클래스를 확장할 필요도 없다. 메서드 시그니처는 관련된 매핑된 구문을 유일하게 확인하기 위해 사용될 수 있다..

➔ Mapper 인터페이스는 다른 인터페이스를 확장할 수 있다. 적절한 명명공간의 구문은 Mapper 인터페이스에 XML 바인딩을 사용한다. 오직 하나의 제한점은 두개의 인터페이스에 같은 메서드 시그니처를 사용할 수 없다는 것이다.

mapper 메서드에 여러개의 파라미터를 전달 할 수 있다. 여러개의 파라미터를 전달하면, 파라미터 목록의 위치에 따라 명명될 것이다. 예를 들면, #{1}, #{2} 기타 등등 이런 식이다. 만약 파라미터의 이름을 변경하고자 한다면, 파라미터의 @Param("paramName") 애노테이션을 사용할 수 있다.

쿼리 결과를 제한하기 위해 메서드에 RowBounds 인스턴스를 전달 할 수 있다.

## *Mapper 애노테이션*

이 프레임워크가 만들어진 이후로 MyBatis 는 XML 기반의 프레임워크이다. 설정은 XML 기반이고 매핑된 구문또한 XML 에 정의한다. MyBatis 3 에서는 새로운 추가 옵션이 생겼다. MyBatis 3 은 편리하고 강력한 자바 기반의 설정 API 를 제공한다. 설정 API 는 XML 기반의 MyBatis 설정의 기초가 된다. 이는 새로운 애노테이션 기반의 설정에도 그대로 적용된다. 애노테이션은 소개하는데 많은 시간이 할애하지 않아도 될 정도로 매핑된 구문을 구현하는 간단한 방법을 제공한다.

➔ Note: 자바 애노테이션은 복잡하고 유연해야 하는 경우에 대해서는 다소 제한적이다. 조사하는데 많은 시간이 소요됨에도 불구하고, 가장 강력한 MyBatis 매핑은 애노테이션으로 처리되지는 못한다. C# 속성은 이러한 제한사항이 없어서 MyBatis.NET 버전은 XML 대안으로 자바보다 다소 더 풍부한 기능을 제공한다. 하지만 자바 애노테이션 기반의 설정이 장점이 없지는 않다.

**사용가능한 애노테이션은 아래에서 설명한다.**

| 애노테이션 | 대상 | XML 요소 | 설명 |
|---|---|---|---|
| @CacheNamespace | Class | 〈cache〉 | 명명공간을 위한 캐시 설정<br>사용가능한 속성들 : implementation, eviction, flushInterval, size 그리고 readWrite. |
| @CacheNamespaceRef | Class | 〈cacheRef〉 | 다른 명명공간의 캐시에 대한 참조<br>사용가능한 속성들 : value(명명공간의 이름). |
| @ConstructorArgs | Method | 〈constructor〉 | 결과 객체 생성자에 전달되는 결과들<br>사용가능한 속성들 : value(인자의 배열) |
| @Arg | Method | 〈arg〉<br>〈idArg〉 | ConstructorArgs 의 일부로 한개의 생성자 인자<br>사용가능한 속성들 : id, column, javaType, jdbcType, typeHandler, select 그리고 resultMap.<br>id 속성은 비교하기 위해 사용되는 값이다.<br>XML 에서는 〈idArg〉 요소와 유사하다. |
| @TypeDiscriminator | Method | 〈discriminator〉 | 결과매핑을 할때 사용될 수 있는 경우에 대한 값들<br>사용가능한 속성들 : column, javaType, jdbcType, typeHandler, cases.<br>cases 속성은 경우(case)의 배열이다. |
| @Case | Method | 〈case〉 | case 의 값과 매핑<br>사용가능한 속성들 : value, type, results.<br>results 속성은 Result 의 배열이다.<br>게다가 이 Case 애노테이션은 Results 애노테이션에서 명시된 ResultMap 와 유사하다. |

| 애노테이션 | 대상 | XML 요소 | 설명 |
|---|---|---|---|
| @Results | Method | <resultMap> | 결과 칼럼이 프로퍼티나 필드에 매핑되는 방법에 대한 상세 설정을 포함하는 결과 매핑의 목록<br>사용가능한 속성들 : value(Result 애노테이션의 배열) |
| @Result | Method | <result><br><id> | 칼럼이 프로퍼티나 필드에 매핑되는 한개의 결과 매핑<br>사용가능한 속성들 : id, column, property, javaType, jdbcType, typeHandler, one, many.<br>id 속성은 프로퍼티를 비교할 때 사용할지를 표시하는 boolean 값이다. (XML 에서 <id> 와 유사하다.)<br>one 속성은 한개의 관계(associations)이고, <association> 과 유사하다. many 속성은 collection 이고 <collection>과 유사하다. 클래스의 명명규칙 충돌을 피하기 위해 명명되었다. |
| @One | Method | <association> | 복잡한 타입의 한개의 프로퍼티를 위한 매핑이다.<br>사용가능한 속성들 : select(매핑 구문의 이름, 예를 들어 매퍼 메서드)<br>Note: 조인 매핑은 애노테이션 API 를 통해서는 지원되지 않는다는 것을 알아야 한다. 순환(circular) 참조를 허용하지 않는 자바 애노테이션의 제약사항때문이다. |
| @Many | Method | <collection> | 복잡한 타입의 collection 프로퍼티를 위한 매핑이다.<br>사용가능한 속성들 : select(매핑 구문의 이름, 예를 들어 매퍼 메서드)<br>Note: 조인 매핑은 애노테이션 API 를 통해서는 지원되지 않는다는 것을 알아야 한다. 순환(circular) 참조를 허용하지 않는 자바 애노테이션의 제약사항때문이다. |
| @MapKey | Method | | 리턴 타입이 Map 인 메서드에서 사용된다. 결과 객체의 List 를 객체의 프로퍼티에 기초한 Map 으로 변환하기 위해 사용된다. |

| 애노테이션 | 대상 | XML 요소 | 설명 |
|---|---|---|---|
| @Options | Method | 매핑 구문의 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.<br>사용가능한 속성들 : useCache=true, flushCache=false, resultSetType=FORWARD_ONLY, statementType=PREPARED, fetchSize=-1, timeout=-1, useGeneratedKeys=false, keyProperty="id", keyColumn="".<br>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.<br>keyColumn 은 키 칼럼이 테이블의 첫번째 칼럼이 아닌 특정 데이터베이스에서만(PostgreSQL 같은) 필요하다.. |
| @Insert<br>@Update<br>@Delete<br>@Select | Method | <insert><br><update><br><delete><br><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.<br>사용가능한 속성들 : value(한개의 SQL 구문을 만들기 위한 문자열의 배열) |
| @InsertProvider<br>@UpdateProvider<br>@DeleteProvider<br>@SelectProvider | Method | <insert><br><update><br><delete><br><select><br>동적 SQL 생성을 허용 | 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.<br>사용가능한 속성들 : type, method.<br>type 속성은 클래스의 패키지 경로를 포함한 전체 이름이다. 메서드는 클래스의 메서드명이다.<br>Note: 이 섹션은 SelectBuilder 클래스에 대한 설명으로, 동적 SQL 을 좀더 깔끔하고 읽기 쉽게 만드는데 도움이 될 수 있다. |
| @Param | Parameter | N/A | 매퍼 메서드가 여러개의 파라미터를 가진다면, 이 애노테이션은 이름에 일치하는 매퍼 메서드 파라미터에 적용된다. 반면에 여러개의 파라미터는 순서대로 명명된다.<br>예를 들어, #{1}, #{2} 등이 디폴트다.<br>@Param("person") 를 사용하면, 파라미터는 #{person} 로 명명된다. |

| 애노테이션 | 대상 | XML 요소 | 설명 |
|---|---|---|---|
| @SelectKey | Method | <selectKey> | 이 애노테이션은 @Insert 또는 @InsertProvider 애노테이션을 사용하는 메서드에서 <selectKey>와 똑같다. 다른 메서드에서는 무시된다. @SelectKey 애노테이션을 명시하면, MyBatis 는 @Options 애노테이션이나 설정 프로퍼티를 통해 셋팅된 key 프로퍼티를 무시할 것이다.<br>사용가능한 속성들 : statement 는 실행할 SQL 구문을 만드는 문자열의 배열이다. keyProperty 는 새로운 값으로 수정될 파라미터 객체의 프로퍼티이다. SQL 이 insert 전후에 실행되는 것을 나타내기 위해 true 나 false 가 되어야 한다. resultType 은 keyProperty 의 자바 타입이다.<br>statementType=PREPARED. |
| @ResultMap | Method | N/A | 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. |

### *Mapper 애노테이션 예제*

이 예제는 insert 하기전에 일련번호를 가져오기 위해 @SelectKey 애노테이션을 사용하는 것을 보여준다.

```
@Insert("insert into table3 (id, name) values(#{nameId}, #{name})")
@SelectKey(statement="call next value for TestSequence",
      keyProperty="nameId", before=true, resultType=int.class)
int insertTable3(Name name);
```

이 예제는 insert 한 후에 값을 가져오기 위해 @SelectKey 애노테이션을 사용하는 것으로 보여준다.

```
@Insert("insert into table2 (name) values(#{name})")
@SelectKey(statement="call identity()", keyProperty="nameId",
      before=false, resultType=int.class)
int insertTable2(Name name);
```

## SelectBuilder

자바 개발자에게 가장 끔찍한 일중 하나는 자바 코드에서 내장 SQL 을 처리하는 것이다. SQL 은 항상 동적으로 생성되기 때문에 그렇다. 반면에 파일이나 저장 프로시저를 외부에 둘수도 있다. 이미 그렇게 하고 있다면, MyBatis 는 XML 매핑에서 동적으로 SQL 을 생성하기 위한 강력한 대답이 될 것이다. 어쨌든 때때로 자바코드에서 SQL 구문을 문자열을 만들어야 할 필요가 종종 있다. 이 경우, MyBatis 는 좀더 쉽게 만들도록 해준다. 이 기능은 + 기호, 따옴표, 개행문자 그리고 콤마와 AND 등의 포맷팅에 관련된 작업을 줄여준다. 자바 코드에서 SQL 코드를 동적으로 생성하는 것은 정말 끔찍하다.

MyBatis 3 은 이 문제는 다루기 위해 다른 컨셉을 소개한다. 단계별로 SQL 구문을 만들도록 메서드를 호출하는 클래스의 인스턴스를 생성 할 수 있다. 하지만 그 후 SQL 은 SQL 이라기 보다는 자바에 가까운 형태로 끝나게 된다. 대신 조금 다른 것을 시도할 것이다.

**SelectBuilder 의 비밀**

SelectBuilder 클래스는 마법스럽지는 않다. 어떻게 작동하는지 모른다면 좋은 선택이라고 보기에도 어렵다. 그래서 바로 살펴보자. SelectBuilder 는 깔끔한 문법이 가능하도록 Static Import 와 ThreadLocal 변수의 조합을 사용한다. 생성하는 메서드는 다음과 같다.

```
public String selectBlogsSql() {
  BEGIN(); // Clears ThreadLocal variable
  SELECT("*");
  FROM("BLOG");
  return SQL();
}
```

정적으로 빌드하는 매우 간단한 예제이다. 그래서 좀더 복잡한 예제를 보자.

```
private String selectPersonSql() {
  BEGIN(); // Clears ThreadLocal variable
  SELECT("P.ID, P.USERNAME, P.PASSWORD, P.FULL_NAME");
  SELECT("P.LAST_NAME, P.CREATED_ON, P.UPDATED_ON");
  FROM("PERSON P");
  FROM("ACCOUNT A");
  INNER_JOIN("DEPARTMENT D on D.ID = P.DEPARTMENT_ID");
  INNER_JOIN("COMPANY C on D.COMPANY_ID = C.ID");
  WHERE("P.ID = A.ID");
  WHERE("P.FIRST_NAME like ?");
  OR();
  WHERE("P.LAST_NAME like ?");
  GROUP_BY("P.ID");
  HAVING("P.LAST_NAME like ?");
  OR();
  HAVING("P.FIRST_NAME like ?");
  ORDER_BY("P.ID");
  ORDER_BY("P.FULL_NAME");
  return SQL();
 }
```

위 SQL 이 빌드되면 다음의 문자열과 거의 같을 것이다. 예를 들면

```
"SELECT P.ID, P.USERNAME, P.PASSWORD, P.FULL_NAME, "
"P.LAST_NAME,P.CREATED_ON, P.UPDATED_ON " +
"FROM PERSON P, ACCOUNT A " +
"INNER JOIN DEPARTMENT D on D.ID = P.DEPARTMENT_ID " +
"INNER JOIN COMPANY C on D.COMPANY_ID = C.ID " +
```

```
"WHERE (P.ID = A.ID AND P.FIRST_NAME like ?) " +
"OR (P.LAST_NAME like ?) " +
"GROUP BY P.ID " +
"HAVING (P.LAST_NAME like ?) " +
"OR (P.FIRST_NAME like ?) " +
"ORDER BY P.ID, P.FULL_NAME";
```

이러한 문법을 선호한다면, 이 기능을 사용해도 좋다. 아마도 다소 에러를 야기할 수도 있다. 각 라인마다 끝에 공백을 추가하는 것을 잊지 말아야 한다. 지금부터 이 문법을 선호한다면, 다음 예제는 자바 문자열 처리보다 좀더 간단할 것이다.

```
private String selectPersonLike(Person p){
  BEGIN(); // Clears ThreadLocal variable
  SELECT("P.ID, P.USERNAME, P.PASSWORD, P.FIRST_NAME, P.LAST_NAME");
  FROM("PERSON P");
  if (p.id != null) {
    WHERE("P.ID like #{id}");
  }
  if (p.firstName != null) {
    WHERE("P.FIRST_NAME like #{firstName}");
  }
  if (p.lastName != null) {
    WHERE("P.LAST_NAME like #{lastName}");
  }
  ORDER_BY("P.LAST_NAME");
  return SQL();
}
```

이 예제에서 다소 특별한건 무엇인가? 자 좀더 자세히 보자. “AND” 키워드가 중복되는 것에 대해서는 그다지 걱정하지 않아도 된다. “WHERE” 와 “AND” 간의 선택하는 것도 그렇다. 예를 들어 위 구문은 PERSON 의 모든 레코드를 리턴하는 쿼리를 생성할 것이다. 파라미터로 ID 또는 firstName, lastName 등 3 개의 조합을 가진다. SelectBuilder 는 “WHEN” 가 어디서 필요한지 “AND”가 어디서 사용되는지 이해하는데 주의해야 한다. 무엇보다도 이러한 메서드의 호출 순서에 영향을 받지 않는다.(단 OR() 메서드는 예외)

아마도 두개의 메서드가 먼저 보일 것이다. 두개의 메서드는 BEGIN() 과 SQL()이다. 먼저 SelectBuilder 의 모든 메서드는 BEGIN()을 호출해서 시작하고 SQL()을 호출해서 끝난다. 물론 로직 중간에 생성될 SQL 을 추출하는 메서드를 호출할 수도 있으나 SQL 을 생성하는 대부분의 상황에서는 BEGIN()으로 시작하고 SQL()로 끝난다. BEGIN() 메서드는 ThreadLocal 변수를 초기화하고 SQL()메서드는 BEGIN() 이후 호출에 따라 SQL 구문을 만든다. BEGIN()은 RESET()과 동의어이다.

위 예제처럼 SelectBuilder 를 사용하기 위해, static import 할 필요가 있다.

```
import static org.mybatis.jdbc.SelectBuilder.*;
```

한번 import 하고나면, SelectBuilder 메서드 모두 사용할 수 있다. 실제 사용가능한 모든 메서드들이다.

| Method | Description |
|---|---|
| BEGIN() / RESET() | 이 메서드들은 SelectBuilder 클래스의 ThreadLocal 상태를 초기화하고 새로운 구문을 추가하기 위해 준비한다. BEGIN() 메서드는 새로운 구문을 추가할때 호출하는 것이 가장 |

| Method | Description |
|---|---|
| | 좋다. RESET() 메서드는 어떠한 이유로 인해 실행중간에 구문을 초기화할때 호출하면 된다. |
| SELECT(String) | SELECT 절을 시작하거나 추가한다. 한번 이상 호출될 수 있고,파라미터는 SELECT 절에 추가될 것이다. 파라미터들은 칼럼과 별칭의 목록이고 각각의 값은 콤마로 구분된다. |
| SELECT_DISTINCT(String) | SELECT 절을 시작하거나 추가한다. 생성된 쿼리에 “DISTINCT”키워드 만을 추가한다. 한번 이상 호출될 수 있고, 파라미터들은 SELECT 절에 추가될 것이다. 파라미터들은 칼럼과 별칭의 목록이고 각각의 값은 콤마로 구분된다. |
| FROM(String) | FROM 절을 시작하거나 추가한다. 한번 이상 호출될 수 있고, 파라미터는 FROM 절에 추가될 것이다. 파라미터는 테이블 명과 별칭이다. |
| JOIN(String)<br>INNER_JOIN(String)<br>LEFT_OUTER_JOIN(String)<br>RIGHT_OUTER_JOIN(String) | 메서드를 호출할때 적절한 타입으로 JOIN 절을 추가한다. 파라미터는 칼럼과 조인하고자 하는 조건으로 구성된 표준 조인을 포함할 수 있다. |
| WHERE(String) | WHERE 절의 조건을 추가한다. AND 를 자동으로 추가하고 여러번 호출될 수 있다. 여러차례 AND 로 새로운 조건을 만든다. OR 로 분기하고자 한다면 OR()메서드를 사용하면 된다. |
| OR() | WHERE 절의 조건을 OR 로 분리한다. 한번 이상 호출될 수 있으나 잘못 호출하면 SQL 에 에러가 발생할 수 있다. |
| AND() | WHEER 절의 조건을 AND 로 분리한다. 한번 이상 호출될 수 있으나 잘못 호출하면 SQL 에 에러가 발생할 수 있다. WHERE 과 HAVING 모두 조건에 AND 를 자동으로 붙여준다. 흔히 사용되지는 않을 것이며,반드시 필요한 경우에 추가로 사용하면 된다.. |
| GROUP_BY(String) | GROUP BY 절을 추가한다. 콤마를 자동으로 추가하고 여러차례 호출될수 있다. 매번 콤마를 추가해서 새로운 조건을 만든다. |
| HAVING(String) | HAVING 절의 조건을 추가한다. AND 를 자동으로 추가하고 여러차례 호출될 수 있다. 매번 AND 를 추가해서 새로운 조건을 추가한다. OR 로 분기하고자 한다면 OR()메서드를 사용하면 된다. |
| ORDER_BY(String) | ORDER BY 절을 추가한다. 콤마를 자동으로 추가하고 여러차례 호출될 수 있다. 매번 콤마를 추가해서 새로운 조건을 만든다. |
| SQL() | 생성되는 SQL 을 리턴하고 SelectBuilder 상태를 리셋(BEGIN()이나 RESET()가 호출된 것처럼)한다. 게다가 이 메서드는 한번만 호출될 수 있다. |

## SqlBuilder

SelectBuilder 와 유사하게, MyBatis 는 일반적인 SqlBuilder 를 가지고 있다. SelectBuilder 가 가진 모든 메서드를 포함할 뿐 아니라, insert, update 그리고 delete 를 만드는 메서드도 가지고 있다. 이 클래스는 SelectProvider , DeleteProvider, InsertProvider, 또는 UpdateProvider 에서 SQL 문자열을 만들 때 유용하다.

위 예제에서 SqlBuilder 를 사용하기 위해, 다음처럼 static 으로 import 할 필요가 있다.

```
import static org.mybatis.jdbc.SqlBuilder.*;
```

SqlBuilder 는 SelectBuilder 의 모든 메서드를 가지고 있으며 몇가지 추가로 메서드를 더 가지고 있다.

| 메서드 | 설명 |
|---|---|
| DELETE_FROM(String) | delete 구문을 시작하고 실제 삭제하고자 하는 테이블을 명시한다. 대개는 WHERE 구문이 뒤에 붙는다. |
| INSERT_INTO(String) | insert 구문을 시작하고 실제 입력하고자 하는 테이블을 명시한다. 대개는 하나 이상의 VALUES() 호출을 뒤따른다. |
| SET(String) | update 구문에서 “set” 에 관련된 목록을 추가한다. |
| UPDATE(String) | update 구문을 시작하고 실제 수정하고자 하는 테이블을 명시한다. 하나 이상의 SET() 호출이 뛰따르고 대개는 WHERE() 호출도 사용한다. |
| VALUES(String, String) | insert 구문에 추가된다. 첫번째 파라미터는 칼럼(column(s))이고 두번째 파라미터는 값(value(s))이다. |

이건 몇가지 예제이다.

```
public String deletePersonSql() {
  BEGIN(); // Clears ThreadLocal variable
  DELETE_FROM("PERSON");
  WHERE("ID = ${id}");
  return SQL();
}

public String insertPersonSql() {
  BEGIN(); // Clears ThreadLocal variable
  INSERT_INTO("PERSON");
  VALUES("ID, FIRST_NAME", "${id}, ${firstName}");
  VALUES("LAST_NAME", "${lastName}");
  return SQL();
}

public String updatePersonSql() {
  BEGIN(); // Clears ThreadLocal variable
  UPDATE("PERSON");
  SET("FIRST_NAME = ${firstName}");
  WHERE("ID = ${id}");
  return SQL();
}
```

## Logging

MyBatis 는 내부 로그 팩토리를 사용하여 로깅 정보를 제공한다. 내부 로그 팩토리는 로깅 정보를 다른 로그 구현체 중 하나에 전달한다.

1. SLF4J

2. Jakarta Commons Logging (JCL - NOT Job Control Language!)

3. Log4J

4. JDK logging

로깅 솔루션은 내부 MyBatis 로그 팩토리의 런타임 체크를 통해 선택된다. MyBatis 로그 팩토리는 가능하면 첫번째 구현체를 사용할 것이다(위 로깅 구현체의 나열 순서는 내부적으로 선택하는 우선순위이다). 만약 MyBatis 가 위 구현체중 하나도 찾지 못한다면, 로깅을 하지 않을 것이다.

많은 환경은 애플리케이션 서버(좋은 예는 Tomcat 과 WebSphere)의 클래스패스의 일부로 JCL 을 사용한다. 이러한 환경을 아는 것이 중요하다. MyBatis 는 로깅 구현체로 JCL 을 사용할 것이다. WebSphere 와 같은 환경에서 Log4J 설정은 무시될 것이다. 왜냐하면 WebSphere 는 자체 JCL 구현체를 제공하기 때문이다. 이러한 사항은 불만스러울수 있다. 왜냐하면 MyBatis 는 당신의 Log4J 설정을 무시하는 것처럼 보일수도 있기 때문이다. (사실 MyBatis 는 당신의 Log4J 설정을 무시한다. 왜냐하면 MyBatis 는 이러한 환경에서 JCL 을 사용할 것이기 때문이다.) 만약 당신의 애플리케이션이 클래스패스에 JCL 을 포함한 환경에서 돌아가지만 다른 로깅 구현체 중 하나를 더 선호한다면, 다음의 메서드 중 하나를 호출하여 다른 로깅 구현체를 선택 할 수 있다.

org.apache.ibatis.logging.LogFactory.useSlf4jLogging();

org.apache.ibatis.logging.LogFactory.useLog4JLogging();

org.apache.ibatis.logging.LogFactory.useJdkLogging();

org.apache.ibatis.logging.LogFactory.useCommonsLogging();

org.apache.ibatis.logging.LogFactory.useStdOutLogging();

MyBatis 메서드를 호출하기 전에 위 메서드 중 하나를 호출해야 한다. 이 메서드들은 런타임 클래스패스에 구현체가 존재하면 그 로그 구현체를 사용하게 한다. 예를 들어, Log4J 로깅을 선택했지만 런타임에 Log4J 구현체가 클래스패스에 없다면, MyBatis 는 Log4J 구현체의 사용을 무시하고 로깅 구현체를 찾아 다시 사용할 것이다.

Jakarta Commons 로깅, Log4J 그리고 JDK 로깅 API 에 대한 설명은 이 문서의 범위를 벗어난다. 이러한 로깅 관련 프레임워크에 대해 좀더 알고 싶다면, 개별 위치에서 좀더 많은 정보를 얻을 수 있을 것이다.

Jakarta Commons 로깅

· http://jakarta.apache.org/commons/logging/index.html

Log4J

· http://jakarta.apache.org/log4j/docs/index.html

JDK 로깅 API

· http://java.sun.com/j2se/1.4.1/docs/guide/util/logging/

로그 설정

MyBatis 가 실제로 사용하는 로그 클래스는 MyBatis 패키지에 포함되어 있지 않다. MyBatis 로깅 구문을 보기 위해서는, java.sql 패키지의 클래스에 대해 로깅을 활성화해야 할 것이다. 해당되는 클래스 목록들이다.

● java.sql.Connection

● java.sql.PreparedStatement

● java.sql.Resultset

● java.sql.Statement

Log4J 를 사용하는 방법을 보여줄 것이다. 로깅 서비스는 하나 이상이 설정파일(예를 들면, log4j.properties)과 새로운 JAR 파일(예를 들면, log4j.jar)을 사용한다. 다음의 예제는 Log4J 를 사용하여 로깅 서비스를 설정할 것이다. 두가지 단계를 거친다.

첫번째 단계 : Log4J JAR 파일 추가하기

Log4J 를 사용하기 때문에, 애플리케이션에 JAR 파일이 있어야 한다. Log4J 를 사용하기 위해, 애플리케이션의 클래스패스에 JAR 파일을 추가할 필요가 있다. 위 URL 에서 Log4J 를 다운로드 할 수 있다.

웹이나 기업용 애플리케이션에서는 WEB-INF/lib 디렉터리에 log4j.jar 파일을 추가할 수 있다. 단독으로 실행되는 애플리케이션에서는 JVM 의 -classpath 시작 파라미터에서 간단히 추가할 수 있다.

두번째 단계 : Log4J 설정하기

Log4J 를 설정하는 것은 간단하다. 먼저 log4j.properties 파일을 만들어서 다음처럼 설정하면 된다.

log4j.properties

```
# 전역 로깅 설정
log4j.rootLogger=ERROR, stdout
# MyBatis 로깅 설정...
#log4j.logger.org.apache.ibatis=DEBUG
#log4j.logger.java.sql.Connection=DEBUG
#log4j.logger.java.sql.Statement=DEBUG
#log4j.logger.java.sql.PreparedStatement=DEBUG
#log4j.logger.java.sql.ResultSet=DEBUG
# Console 출력...
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
```

위 파일은 에러만을 리포트하는 최소한의 설정이다. 파일의 두번째 라인은 stdout appender 에 에러만을 출력한다. appender 는 출력(예를 들면, 콘솔, 파일, 데이터베이스 등)을 모으는 컴포넌트이다. 로그를 최대한 출력하기 위해서는 다음처럼 설정을 변경해야 한다.

log4j.rootLogger=DEBUG, stdout

두번째 라인을 위처럼 변경하면, Log4J 는 'stdout' appender 에 모든 로깅 이벤트를 출력한다. finer 레벨로 레벨을 조정하고자 한다면, 위 파일에서 'SqlMap 로깅 설정'(4 번째라인에서 8 번째라인까지) 에서 각 클래스별로 설정할 수 있다. PreparedStatement 에 대해 콘솔에 DEBUG 레벨로 로깅(SQL 구문)을 하고 싶다면, 다음처럼 7 번째 라인을 수정하면 된다.

```
log4j.logger.java.sql.PreparedStatement=DEBUG
```

log4j.properties 파일에서 남은 설정은 appender 를 설정하기 위해 사용된다. 하지만 이 내용은 이 문서의 범위를 벗어난다. 어쨌든 Log4J 웹사이트에서 좀더 많은 정보를 찾을 수 있다.