# Mapper XML 파일

마이바티스의 가장 큰 장점은 매핑구문이다. 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. 마이바티스는 SQL을 작성하는데 집중하도록 만들어졌다.

SQL Map XML파일은 첫번째(first class)엘리먼트만을 가진다.

- `cache` - 해당 네임스페이스을 위한 캐시 설정
- `cache-ref` - 다른 네임스페이스의 캐시 설정에 대한 참조
- `resultMap` - 데이터베이스 결과데이터를 객체에 로드하는 방법을 정의하는 엘리먼트
- `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 DELETE 구문.
- `select` - 매핑된 SELECT 구문.

다음 섹션에서는 각각에 대해 세부적으로 살펴볼 것이다.

# select

select구문은 마이바티스에서 가장 흔히 사용할 엘리먼트이다. 데이터베이스에서 데이터를 가져온다. 아마도 대부분의 애플리케이션은 데이터를 수정하기보다는 조회하는 기능이 많다. 그래서 마이바티스는 데이터를 조회하고 그 결과를 매핑하는데 집중하고 있다. 조회는 다음 예제처럼 단순한 경우에는 단순하게 설정된다.

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

이 구문의 이름은 selectPerson이고 int타입의 파라미터를 가진다. 그리고 결과 데이터는 HashMap 에 저장된다.

파라미터 표기법을 보자.

```
#{id}
```

이 표기법은 마이바티스에게 PreparedStatement파라미터를 만들도록 지시한다. JDBC를 사용할 때 PreparedStatement에는 “?”형태로 파라미터가 전달된다. 즉 결과적으로 위 설정은 아래와 같이 작동하게 되는 셈이다.

```
// JDBC 코드와 유사함, 마이바티스 코드는 아님…
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 마이바티스는 그 코드들을 작성하지 않아도 되게 해준다.

select 엘리먼트는 각각의 구문이 처리하는 방식에 대해 세부적으로 설정하도록 많은 속성을 설정할 수 있다.

```
<select
  id="selectPerson"
  parameterType="int"
  parameterMap="deprecated"
  resultType="hashmap"
  resultMap="personResultMap"
  flushCache="false"
  useCache="true"
  timeout="10"
  fetchSize="256"
  statementType="PREPARED"
  resultSetType="FORWARD_ONLY">
```

Select 엘리먼트 속성

| 속성 | 설명 |
|---|---|
| `id` | 구문을 찾기 위해 사용될 수 있는 네임스페이스내 유일한 구분자 |
| `parameterType` | 구문에 전달될 파라미터의 패키지 경로를 포함한 전체 클래스명이나 별칭 |
| parameterMap | 외부 parameterMap을 찾기 위한 비권장된 접근방법. 인라인 파라미터 매핑과 parameterType을 대신 사용하라. |
| `resultType` | 이 구문에 의해 리턴되는 기대타입의 패키지 경로를 포함한 전체 클래스명이나 별칭. collection인 경우 collection 타입 자체가 아닌 collection 이 포함된 타입이 될 수 있다. resultType이나 resultMap을 사용하라. |
| `resultMap` | 외부 resultMap 의 참조명. 결과맵은 마이바티스의 가장 강력한 기능이다. resultType이나 resultMap을 사용하라. |
| `flushCache` | 이 값을 true 로 셋팅하면 구문이 호출될때마다 로컬, 2nd 레벨 캐시가 지워질것이다(flush). 디폴트는 false이다. |
| `useCache` | 이 값을 true 로 셋팅하면 구문의 결과가 2nd 레벨 캐시에 캐시 될 것이다. 디폴트는 true이다. |
| `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. 마이바티스에게 Statement, PreparedStatement 또는 CallableStatement를 사용하게 한다. 디폴트는 PREPARED이다. |
| `resultSetType` | FORWARD_ONLY\|SCROLL_SENSITIVE\|SCROLL_INSENSITIVE\|DEFAULT(same as unset)중 하나를 선택할 수 있다. 디폴트는 셋팅하지 않는 것이고 드라이버에 다라 다소 지원되지 않을 수 있다. |
| `databaseId` | 설정된 databaseIdProvider가 있는 경우 마이바티스는 `databaseId` 속성이 없는 모든 구문을 로드하거나 일치하는 `databaseId` 와 함께 로드될 것이다. 같은 구문에서 `databaseId`가 있거나 없는 경우 모두 있다면 뒤에 나온 것이 무시된다. |
| `resultOrdered` | 이 설정은 내포된 결과를 조회하는 구문에서만 적용이 가능하다. true로 설정하면 내포된 결과를 가져오거나 새로운 주요 결과 레코드를 리턴할때 함께 가져오도록 한다. 이전의 결과 레코드에 대한 참조는 더 이상 발생하지 않는다. 이 설정은 내포된 결과를 처리할때 조금 더 많은 메모리를 채운다. 디폴트값은 `false` 이다. |
| `resultSets` | This is only applicable for multiple result sets. It lists the result sets that will be returned by the statement and gives a name to each one. Names are separated by commas. |
| `affectData` | Set this to true when writing a INSERT, UPDATE or DELETE statement that returns data so that the transaction is controlled properly. Also see Transaction Control Method. Default: `false` (since 3.5.12) |

# insert, update and delete

데이터를 변경하는 구문인 insert, update, delete는 매우 간단하다.

```
<insert
  id="insertAuthor"
```

```
  parameterType="domain.blog.Author"
  flushCache="true"
  statementType="PREPARED"
  keyProperty=""
  keyColumn=""
  useGeneratedKeys=""
  timeout="20">

<update
  id="updateAuthor"
  parameterType="domain.blog.Author"
  flushCache="true"
  statementType="PREPARED"
  timeout="20">

<delete
  id="deleteAuthor"
  parameterType="domain.blog.Author"
  flushCache="true"
  statementType="PREPARED"
  timeout="20">
```

Insert, Update 와 Delete 엘리먼트 속성

| 속성 | 설명 |
|---|---|
| `id` | 구문을 찾기 위해 사용될 수 있는 네임스페이스내 유일한 구분자 |
| `parameterType` | 구문에 전달될 파라미터의 패키지 경로를 포함한 전체 클래스명이나 별칭 |
| `parameterMap` | 외부 parameterMap 을 찾기 위한 비권장된 접근방법. 인라인 파라미터 매핑과 parameterType을 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 CALLABLE중 하나를 선택할 수 있다. 마이바티스에게 Statement, PreparedStatement 또는 CallableStatement를 사용하게 한다. 디폴트는 PREPARED 이다. |
| `useGeneratedKeys` | (입력(insert, update)에만 적용) 데이터베이스에서 내부적으로 생성한 키 (예를들어 MySQL또는 SQL Server와 같은 RDBMS의 자동 증가 필드)를 받는 JDBC getGeneratedKeys메소드를 사용하도록 설정하다. 디폴트는 false 이다. |
| `keyProperty` | (입력(insert, update)에만 적용) getGeneratedKeys 메소드나 insert 구문의 selectKey 하위 엘리먼트에 의해 리턴된 키를 셋팅할 프로퍼티를 지정. 디폴트는 셋팅하지 않는 것이다. 여러개의 칼럼을 사용한다면 프로퍼티명에 콤마를 구분자로 나열할수 있다. |
| `keyColumn` | (입력(insert, update)에만 적용) 생성키를 가진 테이블의 칼럼명을 셋팅. 키 칼럼이 테이블이 첫번째 칼럼이 아닌 데이터베이스(PostgreSQL 처럼)에서만 필요하다. 여러개의 칼럼을 사용한다면 프로퍼티명에 콤마를 구분자로 나열할수 있다. |
| `databaseId` | 설정된 databaseIdProvider가 있는 경우 마이바티스는 `databaseId` 속성이 없는 모든 구문을 로드하거나 일치하는 `databaseId`와 함께 로드될 것이다. 같은 구문에서 `databaseId`가 있거나 없는 경우 모두 있다면 뒤에 나온 것이 무시된다. |

Insert, update, delete 구문의 예제이다.

```
<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>
```

앞서 설명했지만 insert는 key생성과 같은 기능을 위해 몇가지 추가 속성과 하위 엘리먼트를 가진다.

먼저 사용하는 데이터베이스가 자동생성키(예를들면 MySQL과 SQL서버)를 지원한다면 useGeneratedKeys="true" 로 설정하고 대상 프로퍼티에 keyProperty 를 셋팅할 수 있다. 예를들어 Author 테이블이 id 칼럼에 자동생성키를 적용했다고 하면 구문은 아래와 같은 형태일 것이다.

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

사용하는 데이터베이스가 다중레코드 입력을 지원한다면, `Author`의 목록이나 배열을 전달할수 있고 자동생성키를 가져올 수 있다.

```
<insert id="insertAuthor" useGeneratedKeys="true"
    keyProperty="id">
  insert into Author (username, password, email, bio) values
  <foreach item="item" collection="list" separator=",">
    (#{item.username}, #{item.password}, #{item.email}, #{item.bio})
  </foreach>
</insert>
```

마이바티스는 자동생성키 칼럼을 지원하지 않는 다른 데이터베이스를 위해 다른 방법 또한 제공한다.

이 예제는 랜덤 ID 를 생성하고 있다.

```
<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>
```

위 예제에서 selectKey구문이 먼저 실행되고 Author id프로퍼티에 셋팅된다. 그리고 나서 insert 구문이 실행된다. 이건 복잡한 자바코드 없이도 데이터베이스에 자동생성키의 행위와 비슷한 효과를 가지도록 해준다.

selectKey 엘리먼트는 다음처럼 설정가능하다.

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

selectKey 엘리먼트 속성

| 속성 | 설명 |
|---|---|
| keyProperty | selectKey구문의 결과가 셋팅될 대상 프로퍼티. |
| keyColumn | 리턴되는 결과셋의 칼럼명은 프로퍼티에 일치한다. 여러개의 칼럼을 사용한다면 칼럼명의 목록은 콤마를 사용해서 구분한다. |
| resultType | 결과의 타입. 마이바티스는 이 기능을 제거할 수 있지만 추가하는게 문제가 되지는 않을것이다. 마이바티스는 String을 포함하여 키로 사용될 수 있는 간단한 타입을 허용한다. |
| order | BEFORE 또는 AFTER를 셋팅할 수 있다. BEFORE로 설정하면 키를 먼저 조회하고 그 값을 keyProperty 에 셋팅한 뒤 insert 구문을 실행한다. AFTER로 설정하면 insert 구문을 실행한 뒤 selectKey 구문을 실행한다. 오라클과 같은 데이터베이스에서는 insert구문 내부에서 일관된 호출형태로 처리한다. |
| statementType | 위 내용과 같다. 마이바티스는 Statement, PreparedStatement 그리고 CallableStatement을 매핑하기 위해 STATEMENT, PREPARED 그리고 CALLABLE 구문타입을 지원한다. |

As an irregular case, some databases allow INSERT, UPDATE or DELETE statement to return result set (e.g. `RETURNING` clause of PostgreSQL and MariaDB or `OUTPUT` clause of MS SQL Server). This type of statement must be written as `<select>` to map the returned data.

```
<select id="insertAndGetAuthor" resultType="domain.blog.Author"
      affectData="true" flushCache="true">
  insert into Author (username, password, email, bio)
  values (#{username}, #{password}, #{email}, #{bio})
  returning id, username, password, email, bio
</select>
```

# sql

이 엘리먼트는 다른 구문에서 재사용가능한 SQL구문을 정의할 때 사용된다. 로딩시점에 정적으로 파라미터처럼 사용할 수 있다. 다른 프로퍼티값은 포함된 인스턴스에서 달라질 수 있다.

```
<sql id="userColumns"> ${alias}.id,${alias}.username,${alias}.password </sql>
```

SQL 조각은 다른 구문에 포함시킬수 있다.

```
<select id="selectUsers" resultType="map">
  select
    <include refid="userColumns"><property name="alias" value="t1"/></include>,
    <include refid="userColumns"><property name="alias" value="t2"/></include>
  from some_table t1
    cross join some_table t2
</select>
```

프로퍼티값은 다음처럼 refid속성이나 include절 내부에서 프로퍼티값으로 사용할 수 있다.

```
<sql id="sometable">
  ${prefix}Table
</sql>
```

```
<sql id="someinclude">
  from
    <include refid="${include_target}"/>
</sql>

<select id="select" resultType="map">
  select
    field1, field2, field3
  <include refid="someinclude">
    <property name="prefix" value="Some"/>
    <property name="include_target" value="sometable"/>
  </include>
</select>
```

# Parameters

앞서 본 구문들에서 간단한 파라미터들의 예를 보았을 것이다. Parameters는 마이바티스에서 매우 중요한 엘리먼트이다. 대략 90%정도 간단한 경우 이러한 형태로 설정할 것이다.

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

위 예제는 매우 간단한 명명된 파라미터 매핑을 보여준다. parameterType은 “int”로 설정되어 있다. Integer과 String과 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>
```

User타입의 파라미터 객체가 구문으로 전달되면 id, username, password 프로퍼티는 찾아서 PreparedStatement파라미터로 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}
```

javaType은 파라미터 객체의 타입을 판단하는 기준이 된다. javaType은 TypeHandler를 사용하여 정의할 수도 있다.

참고 만약 특정 칼럼에 null 이 전달되면 JDBC 타입은 null가능한 칼럼을 위해 필요하다. 처리 방법에 대해서는 PreparedStatement.setNull()메소드의 JavaDoc을 보라.

다양하게 타입 핸들링하기 위해서는 TypeHandler클래스를 명시할 수 있다.

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

다소 설정이 장황하게 보일수 있지만 실제로 이렇게 설정할일은 거의 없다.

숫자 타입을 위해서 크기를 판단하기 위한 numericScale속성도 있다.

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

마지막으로 mode속성은 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(예를들어 오라클 REFCURSOR)라면 파라미터의 타입에 ResultSet 를 매핑하기 위해 resultMap을 명시해야만 한다.

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

마이바티스는 structs와 같은 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 마이바티스로 하여금 PreparedStatement프로퍼티를 만들어서 PreparedStatement파라미터(예를들면 ?)에 값을 셋팅하도록 할 것이다. 이 방법이 안전하기는 하지만 빠른 방법이 선호되기도 한다. 가끔은 SQL 구문에 변하지 않는 값으로 삽입하길 원하기도 한다. 예를들면 ORDER BY와 같은 구문들이다.

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

여기서 마이바티스는 문자열을 변경하거나 이스케이프 처리하지 않는다.

문자열 대체는 SQL 구문의 메타데이터(예를 들어 테이블 이름, 칼럼 이름)가 동적일 때 매우 유용하게 사용할 수 있다. 예를 들면, 테이블의 칼럼 중 하나로 테이블의 데이터를 `select` 하고 싶을 때 아래와 같이 작성하는 대신:

```
@Select("select * from user where id = #{id}")
User findById(@Param("id") long id);

@Select("select * from user where name = #{name}")
User findByName(@Param("name") String name);

@Select("select * from user where email = #{email}")
User findByEmail(@Param("email") String email);

// and more "findByXxx" method
```

다음과 같이 작성할 수 있다:

```
@Select("select * from user where ${column} = #{value}")
User findByColumn(@Param("column") String column, @Param("value") String value);
```

여기서 `${column}` 은 지정한 문자열로 직접 대체되고, `#{value}` 는 "prepared" 될 것이므로 아래와 같이 사용할 수 있다:

```
User userOfId1 = userMapper.findByColumn("id", 1L);
User userOfNameKid = userMapper.findByColumn("name", "kid");
User userOfEmail = userMapper.findByColumn("email", "noone@nowhere.com");
```

이 아이디어는 테이블 이름을 대체하는 데에도 적용할 수 있다.

참고 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.

# Result Maps

resultMap엘리먼트는 마이바티스에서 가장 중요하고 강력한 엘리먼트이다. ResultSet에서 데이터를 가져올때 작성되는 JDBC코드를 대부분 줄여주는 역할을 담당한다. 사실 join매핑과 같은 복잡한 코드는 굉장히 많은 코드가 필요하다. ResultMap은 간단한 구문에서는 매핑이 필요하지 않고 복잡한 구문에서 관계를 서술하기 위해 필요하다.

이미 앞에서 명시적인 resultMap을 가지지 않는 간단한 매핑 구문은 봤을 것이다.

```
<select id="selectUsers" resultType="map">
  select id, username, hashedPassword
  from some_table
```

```
  where id = #{id}
</select>
```

모든 칼럼의 값이 결과가 되는 간단한 구문에서는 HashMap에서 키 형태로 자동으로 매핑된다. 하지만 대부분의 경우 HashMap은 매우 좋은 도메인 모델이 되지는 못한다. 그래서 대부분 도메인 모델로는 자바빈이나 POJO 를 사용할 것이다. 마이바티스는 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 3개의 프로퍼티(id, username, hashedPassword)를 가진다. 이 프로퍼티는 select구문에서 칼럼명과 정확히 일치한다.

그래서 자바빈은 HashMap과 마찬가지로 매우 쉽게 ResultSet에 매핑될 수 있다.

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

그리고 TypeAliases 가 편리한 기능임을 기억해두는게 좋다. TypeAliases를 사용하면 타이핑 수를 줄일 수 있다. 예를들면,

```
<!-- XML설정파일에서 -->
<typeAlias type="com.someapp.model.User" alias="User"/>

<!-- SQL매핑 XML파일에서 -->
<select id="selectUsers" resultType="User">
  select id, username, hashedPassword
  from some_table
  where id = #{id}
</select>
```

이 경우 마이바티스는 칼럼을 자바빈에 이름 기준으로 매핑하여 ResultMap을 자동으로 생성할 것이다. 만약 칼럼명이 프로퍼티명과 다르다면 SQL구문에 별칭을 지정할 수 있다. 예를들면.

```
<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>
```

ResultMap에 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="user_name"/>
  <result property="password" column="hashed_password"/>
</resultMap>
```

구문에서는 resultMap속성에 이를 지정하여 참조한다. 예를들면

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

대부분 이런 유형이라면 지극히 간단할 것이다.

## 복잡한 결과매핑

마이바티스는 한가지 기준으로 만들어졌다. 데이터베이스는 당신이 원하거나 필요로 하는 것이 아니다. 정규화 개념 중 3NF나 BCNF가 완벽히 되도록 하는게 좋지만 실제로는 그렇지도 않다. 그래서 하나의 데이터베이스를 모든 애플리케이션에 완벽히 매핑하는 것이 가능하다면 그것이 가장 좋겠지만 그렇지도 않다. 마이바티스가 이 문제를 해결하기 위해 제공하는 답은 결과매핑이다.

이런 복잡한 구문은 어떻게 매핑할까?

```
<!-- 매우 복잡한 구문 -->
<select id="selectBlogDetails" 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 Author에 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" 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" javaType="Author"/>
    <collection property="comments" ofType="Comment">
      <id property="id" column="comment_id"/>
    </collection>
    <collection property="tags" ofType="Tag" >
      <id property="id" column="tag_id"/>
    </collection>
    <discriminator javaType="int" column="draft">
      <case value="1" resultType="DraftPost"/>
    </discriminator>
  </collection>
</resultMap>
```

resultMap엘리먼트는 많은 하위 엘리먼트를 가진다. 다음은 resultMap 엘리먼트의 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.
  - 중첩된 결과 매핑 – resultMap 스스로의 연관관계
- `collection` – 복잡한 타입의 컬렉션
  - 중첩된 결과 매핑 – resultMap 스스로의 연관관계
- `discriminator` – 사용할 resultMap 을 판단하기 위한 결과값을 사용
  - `case` – 몇가지 값에 기초한 결과 매핑
    - 중첩된 결과 매핑 – 이 경우 또한 결과매핑 자체이고 이러한 동일한 엘리먼트를 많이 포함하거나 외부 resultMap을 참조할 수 있다.

resultMap 엘리먼트 속성

| 속성 | 설명 |
|---|---|
| id | 결과매핑을 참조하기 위해 사용할 수 있는 값으로 네임스페이스에서 유일한 식별자 |
| type | 패키지를 포함한 자바 클래스명이나 타입별칭(내장된 타입별칭이 목록은 위 표를 보자). |
| autoMapping | 이 설정을 사용하면 마이바티스는 결과매핑을 자동매핑으로 처리할지 말지를 처리한다. 이 속성은 autoMappingBehavior 라는 전역설정을 덮는다. 디폴트는 unset이다. |

**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 엘리먼트에 대해 상세하게 살펴볼 것이다.

### id, result

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

이건 결과 매핑의 가장 기본적인 형태이다. id와 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.

id 와 result 엘리먼트 속성

| 속성 | 설명 |
|---|---|
| property | 결과 칼럼에 매핑하기 위한 필드나 프로퍼티. 자바빈 프로퍼티가 해당 이름과 일치한다면 그 프로퍼티가 사용될 것이다. 반면에 마이바티스는 해당 이름이 필드를 찾을 것이다. 점 표기를 사용하여 복잡한 프로퍼티 검색을 사용할 수 있다. 예를들어 “username”과 같이 간단하게 매핑될 수 있거나 “address.street.number” 처럼 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 마이바티스는 타입을 찾아낼 수 있다. 반면에 HashMap으로 매핑한다면 기대하는 처리를 명확히 하기 위해 javaType 을 명시해야 한다. |
| jdbcType | 지원되는 타입 목록에서 설명하는 JDBC 타입. JDBC타입은 insert, update 또는 delete 하는 null 입력이 가능한 칼럼에서만 필요하다. JDBC의 요구사항이지 마이바티스의 요구사항이 아니다. JDBC로 직접 코딩을 하다보면 null이 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 TypeHandler구현체의 패키지를 포함한 전체 클래스명이나 타입별칭이다. |

### 지원되는 JDBC 타입

상세한 설명전에 마이바티스는 jdbcType열거를 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` | `ARRAY` |

## 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 메소드가 없어도 인스턴스화 할 때 값을 셋팅하도록 해준다. 마이바티스는 private 프로퍼티와 private 자바빈 프로퍼티를 지원하지만 많은 사람들은 생성자 주입을 선호한다. constructor엘리먼트는 이러한 처리를 가능하게 한다.

다음의 생성자를 보자.

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

결과를 생성자에 주입하려면 MyBatis 가 어떻게든 생성자를 식별해야 한다. 다음 예제에서 MyBatis 는 `java.lang.Integer`, `java.lang.String` and `int` 의 순서로 세 개의 매개 변수로 선언 된 생성자를 검색한다.

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

많은 매개 변수를 가진 생성자를 다루는 경우, arg 엘리먼트의 순서를 유지하는 것은 오류가 발생하기 쉽다.
3.4.3 부터, 각 매개 변수의 이름을 지정하여 임의의 순서로 arg 엘리먼트를 작성할 수 있다. 이름으로 생성자 매개변수를 참조하려면, `@Param` annotation 을 추가하거나 '-parameters' 컴파일러 옵션을 통해 프로젝트를 컴파일하고 `useActualParamName` 을 활성화 할 수 있다. (이 옵션은 기본적으로 활성화 되어있다). 다음 예제는 2번째 및 3번째 매개변수의 순서가 선언 된 순서와 일치하지 않더라도 동일한 생성자에 대해 유효하다.

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

같은 이름과 형태의 쓰기 가능한 property 가 있는 경우는 `javaType` 를 생략 할 수 있다.

나머지 속성과 규칙은 id와 result엘리먼트와 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 마이바티스는 타입을 찾아낼 수 있다. 반면에 HashMap 으로 매핑한다면 기대하는 처리를 명확히 하기 위해 javaType을 명시해야 한다. |
| `jdbcType` | 지원되는 타입 목록에서 설명하는 JDBC 타입. JDBC타입은 insert, update 또는 delete 하는 null 입력이 가능한 칼럼에서만 필요하다. JDBC의 요구사항이지 마이바티스의 요구사항이 아니다. JDBC로 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 TypeHandler구현체의 패키지를 포함한 전체 클래스명이 |

| 속성 | 설명 |
| --- | --- |
| | 나 타입별칭이다. |
| `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. 세부적인 설명은 association엘리먼트를 보라. |
| `resultMap` | 이 인자의 내포된 결과를 적절한 객체로 매핑할 수 있는 ResultMap 의 ID이다. 다른 select구문을 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. 자세히 알기 위해서는 association엘리먼트를 보라. |
| `name` | 생성자 매개변수의 이름. name 을 지정함으로써 순서에 상관없이 arg 엘리먼트를 작성할 수 있다. 위의 설명을 보아라. 3.4.3 부터 가능하다. |

### association

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

association 엘리먼트는 “has-one”타입의 관계를 다룬다. 예를들어 Blog는 하나의 Author를 가진다. association 매핑은 다른 결과와 작동한다. 값을 가져오기 위해 대상 프로퍼티를 명시한다.

마이바티스는 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 엘리먼트내 프로퍼티들을 보자. 보이는 것처럼 select와 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 마이바티스는 해당 이름이 필드를 찾을 것이다. 점 표기를 사용하여 복잡한 프로퍼티 검색을 사용할 수 있다. 예를들어 “username”과 같이 간단하게 매핑될 수 있거나 “address.street.number” 처럼 복잡하게 매핑될수도 있다. |
| `javaType` | 패키지 경로를 포함한 클래스 전체명이거나 타입 별칭. 자바빈을 사용한다면 마이바티스는 타입을 찾아낼 수 있다. 반면에 HashMap 으로 매핑한다면 기대하는 처리를 명확히 하기 위해 javaType을 명시해야 한다. |
| `jdbcType` | 지원되는 타입 목록에서 설명하는 JDBC 타입. JDBC타입은 insert, update 또는 delete 하는 null 입력이 가능한 칼럼에서만 필요하다. JDBC의 요구사항이지 마이바티스의 요구사항이 아니다. JDBC로 직접 코딩을 하다보면 null이 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 타입별칭이다. |

### 연관(Association)을 위한 중첩된 Select

| 속성 | 설명 |
| --- | --- |
| `column` | 데이터베이스의 칼럼명이나 별칭된 칼럼 라벨. resultSet.getString(columnName)에 전달되는 같은 문자열이다. 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. |

| 속성 | 설명 |
| --- | --- |
| `select` | 다른 매핑된 구문의 ID는 이 프로퍼티 매핑이 필요로 하는 복잡한 타입을 로드할 것이다. column 속성의 칼럼으로 부터 가져온 값은 대상 select 구문에 파라미터로 전달될 것이다. 노트: 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. |
| `fetchType` | 선택가능한 속성으로 사용가능한 값은 `lazy`와 `eager`이다. 이 속성을 사용하면 전역 설정파라미터인 `lazyLoadingEnabled`를 대체한다. |

예를들면.

```
<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>
```

여기엔 두개의 select 구문이 있다. 하나는 Blog를 로드하고 다른 하나는 Author를 로드한다. 그리고 Blog의 resultMap은 author프로퍼티를 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 조회 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 지연로딩으로 호출하고 게다가 성능은 많이 나빠질 것이다.

그래서 다른 방법이 있다.

### 관계를 위한 내포된 결과(Nested Results)

| 속성 | 설명 |
| --- | --- |
| `resultMap` | 이 인자의 내포된 결과를 적절한 객체로 매핑할 수 있는 ResultMap의 ID 이다. 다른 select구문을 호출하기 위한 대체방법이다. 여러개의 테이블을 조인하는 것을 하나의 ResultSet으로 매핑하도록 해준다. ResultSet은 사본을 포함할 수 있고 데이터를 반복할 수도 있다. 가능하게 하기 위해서 내포된 결과를 다루도록 결과맵을 "연결"하자. 자세히 알기 위해서는 association엘리먼트를 보라. |
| `columnPrefix` | 여러개의 테이블을 조인할때 ResultSet에서 칼럼명의 중복을 피하기 위해 칼럼별칭을 사용할 수 있다. 칼럼을 외부 결과매핑에 매핑하기 위해 columnPrefix를 명시하자. 이 절의 뒤에 나오는 에제를 보자. |
| `notNullColumn` | 기본적으로 자식객체는 칼럼중 적어도 하나를 null이 아닌 자식객체의 프로퍼티에 매핑할때 만들어진다. 이 속성을 사용해서 칼럼이 값을 가져야만 하는 것을 명시해서 행위를 변경할 수 있다. 그래서 마이바티스는 이러한 칼럼이 null이 아닐때만 자식 객체를 만들것이다. 여러개의 칼럼명은 구분자로 콤마를 사용해서 명시한다. 디폴트값은 unset이다. |

| 속성 | 설명 |
| --- | --- |
| autoMapping | 이 속성을 사용하면 마이바티스는 결과를 프로퍼티에 매핑할때 자동매핑을 사용할지 말지를 정한다. 이 속성은 전역설정인 autoMappingBehavior를 무시하게 한다. 외부 결과매핑에는 영향을 주지 않는다. 그래서 select 나 resultMap 속성을 함께 사용하는 것은 의미가 없다. 디폴트값은 unset이다. |

위에서 내포된 관계의 매우 복잡한 예제를 보았을 것이다. 다음은 작동하는 것을 보기 위한 간단한 예제이다. 개별구문을 실행하는 것 대신에 Blog와 Author테이블을 함께 조인했다.

```
<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>
```

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

```
<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>
```

위 예제에서 Author인스턴스를 로드하기 위한 “authorResult” 결과매핑으로 위임된 Blog의 “author”관계를 볼 수 있을 것이다.

매우 중요 : id 엘리먼트는 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 엘리먼트를 사용했다. 이 외부 resultMap은 Author resultMap을 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="id" column="blog_id" />
  <result property="title" column="blog_title"/>
  <association property="author" 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>
```

블로그에 공동저자가 있다면 어쩌지? select구문은 다음과 같을 것이다.

```
<select id="selectBlog" resultMap="blogResult">
  select
    B.id            as blog_id,
    B.title         as blog_title,
    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,
    CA.id           as co_author_id,
    CA.username     as co_author_username,
    CA.password     as co_author_password,
    CA.email        as co_author_email,
    CA.bio          as co_author_bio
  from Blog B
  left outer join Author A on B.author_id = A.id
  left outer join Author CA on B.co_author_id = CA.id
  where B.id = #{id}
</select>
```

저자(Author)를 위한 결과매핑은 다음처럼 정의했다.

```
<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>
```

결과의 칼럼명은 결과매핑에 정의한 칼럼과는 다르기 때문에 공동저자 결과를 위한 결과매핑을 재사용하기 위해 `columnPrefix`를 명시할 필요가 있다.

```
<resultMap id="blogResult" type="Blog">
  <id property="id" column="blog_id" />
  <result property="title" column="blog_title"/>
  <association property="author"
    resultMap="authorResult" />
  <association property="coAuthor"
    resultMap="authorResult"
    columnPrefix="co_" />
</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 엘리먼트는 관계를 파악하기 위해 작동한다. 사실 이 내용이 중복되는 내용으로 차이점에 대해서만 주로 살펴보자.

위 예제를 계속 진행하기 위해 Blog는 오직 하나의 Author를 가진다. 하지만 Blog는 많은 Post 를 가진다. Blog 클래스에 다음처럼 처리될 것이다.

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

List에 내포된 결과를 매핑하기 위해 collection엘리먼트를 사용한다. association 엘리먼트와는 달리 조인에서 내포된 select나 내포된 결과를 사용할 수 있다.

## Collection 을 위한 내포된(Nested) Select

먼저 Blog의 Post를 로드하기 위한 내포된 select 를 사용해보자.

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

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

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

바로 눈치챌 수 있는 몇가지가 있지만 대부분 앞서 배운 association 엘리먼트와 매우 유사하다. 먼저 collection 엘리먼트를 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="id" ofType="Post" select="selectPostsForBlog"/>
```

독자 왈: “Post의 ArrayList타입의 글 목록”

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

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

## Collection을 위한 내포된(Nested) Results

이 시점에 collection 을 위한 내포된 결과가 어떻게 작동하는지 짐작할 수 있을 것이다. 왜냐하면 association와 정확히 일치하기 때문이다. 하지만 “ofType” 속성이 추가로 적용되었다.

먼저 SQL을 보자.

```
<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>
```

다시보면 Blog와 Post테이블을 조인했고 간단한 매핑을 위해 칼럼명에 적절한 별칭을 주었다. 이제 Post의 Collection을 가진 Blog의 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 id엘리먼트의 중요성을 기억해두거나 기억이 나지 않으면 association섹션에서 다시 읽어둬라.

혹시 결과 매핑의 재사용성을 위해 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" columnPrefix="post_"/>
</resultMap>

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

참고 associations과 collections에서 내포의 단계 혹은 조합에는 제한이 없다. 매핑할때는 성능을 생각해야 한다. 단위테스트와 성능테스트는 애플리케이션에서 가장 좋은 방법을 찾도록 지속해야 한다. 마이바티스는 이에 수정비용을 최대한 줄이도록 해줄 것이다.

복잡한 association 과 collection 매핑은 어려운 주제다. 문서에서는 여기까지만 설명을 할수 있다. 연습을 더 하면 명확하게 이해할 수 있을것이다.

## discriminator

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

종종 하나의 데이터베이스 쿼리는 많고 다양한 데이터 타입의 결과를 리턴한다. discriminator엘리먼트는 클래스 상속 관계를 포함하여 이러한 상황을 위해 고안되었다. discriminator는 자바의 switch와 같이 작동하기 때문에 이해하기 쉽다.

discriminator정의는 colume과 javaType속성을 명시한다. colume은 마이바티스로 하여금 비교할 값을 찾을 것이다. javaType은 동일성 테스트와 같은 것을 실행하기 위해 필요하다. 예를들어

```
<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 마이바티스는 결과데이터에서 각각의 레코드를 가져와서 vehicle_type값과 비교한다. 만약 discriminator비교값과 같은 경우가 생기면 이 경우에 명시된 resultMap을 사용할 것이다. 해당되는 경우가 없다면 무시된다. 만약 일치하는 경우가 하나도 없다면 마이바티스는 discriminator블럭 밖에 정의된 resultMap을 사용한다. carResult가 다음처럼 정의된다면

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

doorCount프로퍼티만이 로드될 것이다. discriminator경우들의 독립적인 결과를 만들어준다. 이 경우 우리는 물론 car와 vehicle간의 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>
```

vehicleResult와 carResult의 모든 프로퍼티들이 로드 될 것이다.

한가지 더 도처에 설정된 외부 정의를 찾게 될지도 모른다. 그러므로 간결한 매핑 스타일의 문법이 있다. 예를들면

```
<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>
```

참고 모든 결과 매핑이 있고 모두 명시하고 싶지 않다면 마이바티스는 칼럼과 프로퍼티 명으로 자동으로 매핑할 것이다. 이 예제는 실제로 필요한 내용보다 많이 서술되어 있다.

# Auto-mapping

이전의 절에서 이미 본것처럼 간단한 경우 마이바티스는 결과를 자동으로 매핑할 수 있고 간단하지 않은 경우에는 직접 결과 매핑을 만들필요가 있다. 하지만 이 절에서 보는것처럼 두가지 방법을 적절히 혼용할수도 있다. 자동매핑을 처리하는 방법을 조금 더 보자.

결과를 자동매핑할때 마이바티스는 칼럼명을 가져와서 대소문자를 무시한 같은 이름의 프로퍼티를 찾을 것이다. 칼럼명이 *ID*라면 *id* 이름의 프로퍼티를 찾는다는 것을 뜻한다. 마이바티스는 *ID* 칼럼값을 사용해서 *id* 프로퍼티를 설정할것이다.

대개 데이터베이스 칼럼은 대문자를 사용해서 명명하고 단어 사이사이에는 밑줄을 넣는다. 자바 프로퍼티는 대개 낙타표기법을 사용해서 명명한다. 이런 둘사이의 자동매핑을 가능하게 하기 위해서는 `mapUnderscoreToCamelCase` 를 true로 설정하자.

자동매핑은 결과매핑을 선언한 경우에도 동작한다. 이런 경우 각각의 결과매핑에서 ResultSet의 모든 칼럼은 자동매핑이 아니라 수동매핑으로 처리한다. 다음 샘플에서 *id* 와 *userName* 칼럼은 자동매핑되고 *hashed_password* 칼럼은 수동으로 매핑한다.

```
<select id="selectUsers" resultMap="userResultMap">
  select
    user_id             as "id",
```

```
    user_name           as "userName",
    hashed_password
  from some_table
  where id = #{id}
</select>
```

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

자동매핑에는 3가지가 있다.

- `NONE` - 자동매핑을 사용하지 않는다. 오직 수동으로 매핑한 프로퍼티만을 설정할것이다.
- `PARTIAL` - 조인 내부에 정의한 내포된 결과매핑을 제외하고 자동매핑한다.
- `FULL` - 모든것을 자동매핑한다.

디폴트값은 `PARTIAL`이다. `FULL` 을 사용할때 자동매핑은 조인결과나 같은 레코드의 여러가지 다른 엔터티를 가져올때 예상치 못한 문제를 야기할 수 있다. 이런 위험을 이해하기 위해 다음의 샘플을 보자.

```
<select id="selectBlog" resultMap="blogResult">
  select
    B.id,
    B.title,
    A.username,
  from Blog B left outer join Author A on B.author_id = A.id
  where B.id = #{id}
</select>
```

```
<resultMap id="blogResult" type="Blog">
  <association property="author" resultMap="authorResult"/>
</resultMap>

<resultMap id="authorResult" type="Author">
  <result property="username" column="author_username"/>
</resultMap>
```

*Blog* 와 *Author* 는 자동매핑으로 처리하지만 *Author*는 *id* 프로퍼티를 가지고 ResultSet은 *id* 칼럼을 가진다. 그래서 기대한 것과는 달리 저자(Author)의 id는 블로그(Blog)의 id로 채워질것이다. `FULL` 는 이런 위험을 가진다.

자동매핑 설정에 상관없이 구문별로 `autoMapping`속성을 추가해서 자동매핑을 사용하거나 사용하지 않을수도 있다.

```
<resultMap id="userResultMap" type="User" autoMapping="false">
  <result property="password" column="hashed_password"/>
</resultMap>
```

# cache

마이바티스는 쉽게 설정가능하고 변경가능한 쿼리 캐싱 기능을 가지고 있다. 마이바티스 3 캐시 구현체는 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. (예를들면. no Flush Interval)
- 캐시는 리스트나 객체에 대해 1024 개의 참조를 저장할 것이다. (쿼리 메소드가 실행될때마다)

- 캐시는 읽기/쓰기 캐시처럼 처리될 것이다. 이것은 가져올 객체는 공유되지 않고 호출자에 의해 안전하게 변경된다는 것을 의미한다.

참고 캐시는 cache 태그를 사용한 매퍼에 선언된 구문에만 적용된다. XML 매퍼와 함께 Java API를 사용하는 경우 Java 매퍼에 작성된 구문을 캐시 적용 대상에 포함시키려면 `@CacheNamespaceRef`을 사용하여 XML 매퍼의 namespace를 지정해야 한다.

모든 프로퍼티는 cache 엘리먼트의 속성을 통해 변경가능하다. 예를들면

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

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

사용가능한 캐시 전략은 4가지이다.

- `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.

size는 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 리턴할 것이다. 게다가 그 객체는 변경할 수 없다. 이건 종종 성능에 잇점을 준다. 읽고 쓰는 캐시는 캐시된 객체의 복사본을 리턴할 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. type속성에 명시된 클래스는 org.apache.ibatis.cache.Cache인터페이스를 반드시 구현해야 한다. 이 인터페이스는 마이바티스 프레임워크의 가장 복잡한 구성엘리먼트 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();
}
```

캐시를 설정하기 위해 캐시 구현체에 public 자바빈 프로퍼티를 추가하고 cache 엘리먼트를 통해 프로퍼티를 전달한다. 예를들어 다음 예제는 캐시 구현체에서 “setCacheFile(String file)” 를 호출하여 메소드를 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. 마이바티스는 변환할 것이다. configuration properties 에 정의된 값을 대체할 placeholder(예: `${cache.file}` ) 를 지정할 수 있다.

3.4.2 부터, MyBatis 는 모든 properties 를 설정한 후 초기화 메서드를 호출하도록 지원한다. 이 기능을 사용하려면, custom 캐시 클래스에서 `org.apache.ibatis.builder.InitializingObject` interface 를 구현해야 한다.

```
public interface InitializingObject {
  void initialize() throws Exception;
}
```

캐시 설정과 캐시 인스턴스가 SQL Map 파일의 네임스페이스에 묶여지는(bound) 것을 기억하는게 중요하다. 게다가 같은 네임스페이스내 모든 구문은 묶여진다. 구문별로 캐시와 상호작동하는 방법을 변경할 수 있거나 두개의 간단한 속성을 통해 완전히 배제될 수도 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 flushCache와 useCache 속성을 변경하는 것이 좋다. 예를들어 캐시에서 특정 select 구문의 결과를 제외하고 싶거나 캐시를 지우기 위한 select 구문을 원할 것이다. 유사하게 실행할때마다 캐시를 지울 필요가 없는 update구문도 있을 것이다.

## cache-ref

이전 섹션 내용을 돌이켜보면서 특정 네임스페이스을 위한 캐시는 오직 하나만 사용되거나 같은 네임스페이스내에서는 구문마다 캐시를 지울수 있다. 네임스페이스간의 캐시 설정과 인스턴스를 공유하고자 할때가 있을 것이다. 이 경우 cache-ref 엘리먼트를 사용해서 다른 캐시를 참조할 수 있다.

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