매퍼 설정

마이바티스 XML 설정파일은 다양한 설정과 프로퍼티를 가진다. 문서의 구조는 다음과 같다.:

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

properties

이 설정은 외부에 옮길 수 있다. 자바 프로퍼티 파일 인스턴스에 설정할 수도 있고 properties 엘리먼트의 하위 엘리먼트에 둘 수도 있다. 예를들면:

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

속성들은 파일 도처에 둘 수도 있다. 예를들면:

```
<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 예제에서 username과 password는 properties엘리먼트의 설정된 값으로 대체될 수 있다. driver와 url속성은 config.properties파일에 포함된 값으로 대체될 수도 있다. 이것은 설정에 대한 다양한 옵션을 제공하는 셈이다.

속성은 SqlSessionFactoryBuilder.build() 메소드에 전달될 수 있다. 예를들면:

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

// ... or ...

SqlSessionFactory factory = new SqlSessionFactoryBuilder().build(reader, environment, props);
```

속성이 한개 이상 존재한다면 마이바티스는 일정한 순서로 로드한다.:

- properties 엘리먼트에 명시된 속성을 가장 먼저 읽는다.
- properties 엘리먼트의 클래스패스 자원이나 url 속성으로 부터 로드된 속성을 두번째로 읽는다. 그래서 이미 읽은 값이 있다면 덮어쓴다.
- 마지막으로 메소드 파라미터로 전달된 속성을 읽는다. 앞서 로드된 값을 덮어쓴다.

그래서 가장 우선순위가 높은 속성은 메소드의 파라미터로 전달된 값이고 그 다음은 자원 및 url 속성이고 마지막은 properties 엘리먼트에 명시된 값이다.

Mybatis 3.4.2 부터, placeholder 에 기본값을 아래처럼 지정할 수 있다.

```
<dataSource type="POOLED">
  <!-- ... -->
  <property name="username" value="${username:ut_user}"/> <!-- 만약 'username' 속성이 없다면, username 은 'ut_user' 가 된다. -->
</dataSource>
```

이 기능은 기본적으로 비활성화되어 있다. placeholder 에 기본값을 지정한다면, 이 기능을 활성화하려면 다음과 같이 특별한 속성을 추가해주어야 한다.

```
<properties resource="org/mybatis/example/config.properties">
  <!-- ... -->
  <property name="org.apache.ibatis.parsing.PropertyParser.enable-default-value" value="true"/> <!-- 이 기능을 활성화한다. -->
</properties>
```

NOTE 또한 이미 property key (예 : `db : username`)로 `":"` 를 사용하거나 SQL 정의에서 OGNL 표현식의 삼항 연산자 (예 : `${tableName != null ? tableName : 'global_constants'}`)를 사용하는 경우 다음과 같이 특별한 속성을 추가하여 키와 기본값을 구분하는 문자를 변경해야한다.

```
<properties resource="org/mybatis/example/config.properties">
  <!-- ... -->
  <property name="org.apache.ibatis.parsing.PropertyParser.default-value-separator" value="?:"/> <!-- separator 의 기본값을 변경한다. -->
</properties>
```

```
<dataSource type="POOLED">
  <!-- ... -->
  <property name="username" value="${db:username?:ut_user}"/>
</dataSource>
```

settings

런타임시 마이바티스의 행위를 조정하기 위한 중요한 값들이다. 다음 표는 설정과 그 의미 그리고 디폴트 값을 설명한다.

설정	설명	사용가능한 값들	디폴트
cacheEnabled	설정에서 각 매퍼에 설정된 캐시를 전역적으로 사용할지 말지에 대한 여부	true | false	true

설정	설명	사용가능한 값들	디폴트
lazyLoadingEnabled	지연로딩을 사용할지에 대한 여부. 사용하지 않는다면 모두 즉시 로딩할 것이다. 이 값은 `fetchType` 속성을 사용해서 대체할 수 있다.	true | false	false
aggressiveLazyLoading	활성화되면 모든 메서드 호출은 객체의 모든 lazy properties 을 로드한다. 그렇지 않으면 각 property 가 필요에 따라 로드된다. (`lazyLoadTriggerMethods` 참조).	true | false	false (3.4.1 부터 true)
multipleResultSetsEnabled	한 개의 구문에서 여러 개의 ResultSet을 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. true로 설정하면 생성키를 강제로 생성한다. 일부 드라이버(예를들면, Derby)에서는 이 설정을 무시한다.	true | false	False
autoMappingBehavior	마이바티스가 칼럼을 필드/프로퍼티에 자동으로 매핑할지와 방법에 대해 명시. PARTIAL은 간단한 자동매핑만 할뿐 내포된 결과에 대해서는 처리하지 않는다. FULL은 처리가능한 모든 자동매핑을 처리한다.	NONE, PARTIAL, FULL	PARTIAL
autoMappingUnknownColumnBehavior	자동매핑 대상 중 알 수 없는 칼럼(이나 알 수 없는 프로퍼티 타입)을 발견했을 때 행위를 명시 • `NONE` : 아무것도 하지 않음 • `WARNING` : 경고 로그를 출력 (`'org.apache.ibatis.session.AutoMappingUnknownColumnBehavior'` 의 로그레벨은 `WARN` 이어야 한다.) • `FAILING` : 매핑이 실패한다. (`SqlSessionException` 예외를 던진다.) Note that there could be false-positives when `autoMappingBehavior` is set to `FULL`.	NONE, WARNING, FAILING	NONE
defaultExecutorType	디폴트 실행자(executor) 설정. SIMPLE 실행자는 특별히 하는 것이 없다. REUSE 실행자는 PreparedStatement를 재사용한다. BATCH 실행자는 구문을 재사용하고 수정을 배치처리한다.	SIMPLE REUSE BATCH	SIMPLE
defaultStatementTimeout	데이터베이스로의 응답을 얼마나 오래 기다릴지를 판단하는 타임아웃을 설정	양수	설정되지 않음(null)
defaultFetchSize	조회결과를 가져올때 가져올 데이터 크기를 제어하는 용도로 드라이버에 힌트를 설정 이 파라미터값은 쿼리 설정으로 변경할 수 있다.	양수	설정하지 않음(null)
defaultFetchSize	결과를 가져오는 크기를 제어하는 힌트처럼 드라이버에 설정한다. 이 파라미터는 쿼리설정으로 변경할 수 있다.	양수	셋팅되지 않음(null)
defaultResultSetType	각 구문의 설정을 생략할 때 스크롤 하는 방법을 지정한다. (3.5.2 부터)	FORWARD_ONLY | SCROLL_SENSITIVE | SCROLL_INSENSITIVE | DEFAULT(설정하지 않을 때와 동일하게 동작)	Not Set (null)
safeRowBoundsEnabled	중첩구문내 RowBound사용을 허용 허용한다면 false로 설정	true | false	False
safeResultHandlerEnabled	중첩구문내 ResultHandler사용을 허용 허용한다면 false로 설정	true | false	True
mapUnderscoreToCamelCase	전통적인 데이터베이스 칼럼명 형태인 A_COLUMN을 CamelCase형태의 자바 프로퍼티명 형태인 aColumn으로 자동으로 매핑하도록 함	true | false	False
localCacheScope	마이바티스는 순환참조를 막거나 반복된 쿼리의 속도를 높히기 위해 로컬캐시를 사용한다. 디폴트 설정인 SESSION을 사용해서 동일 세션의 모든 쿼리를 캐시한다. localCacheScope=STATEMENT 로 설정하면 로컬 세션은 구문 실행할때만 사용하고 같은 SqlSession에서 두 개의 다른 호출 사이에는 데이터를 공유하지 않는다.	SESSION | STATEMENT	SESSION
jdbcTypeForNull	JDBC타입을 파라미터에 제공하지 않을때 null값을 처리한 JDBC타입을 명시한다. 일부 드라이버는 칼럼의 JDBC타입을 정의하도록 요구하지만 대부분은 NULL, VARCHAR 나 OTHER 처럼 일반적인 값을 사용해서 동작한다.	JdbcType 이늄. 대부분은 NULL, VARCHAR 나 OTHER 를 공통적으로 사용한다.	OTHER
lazyLoadTriggerMethods	지연로딩을 야기하는 객체의 메소드를 명시	메소드 이름을 나열하고 여러 개일 경우 콤마(,)로 구분	equals,clone,hashCode,toStrir
defaultScriptingLanguage	동적으로 SQL을 만들기 위해 기본적으로 사용하는 언어를 명시	타입별칭이나 패키지 경로를 포함한 클래스명	org.apache.ibatis.scripting.xml
defaultEnumTypeHandler	Enum에 기본적으로 사용되는 `TypeHandler` 를 지정합니다. (3.4.5 부터)	타입별칭이나 패키지 경로를 포함한 클래스명	org.apache.ibatis.type.EnumT
callSettersOnNulls	가져온 값이 null일 때 setter나 맵의 put 메소드를 호출할지를 명시 Map.keySet() 이나 null값을 초기화할때 유용하다. int,	true | false	false

설정	설명	사용가능한 값들	디폴트
	boolean 등과 같은 원시타입은 null을 설정할 수 없다는 점은 알아두면 좋다.		
returnInstanceForEmptyRow	MyBatis 는 기본적으로 모든 열들의 행이 NULL 이 반환되었을 때 `null` 을 반환한다. 이 설정을 사용하면 MyBatis가 대신 empty 인스턴스를 반환한다. nested results(collection 또는 association) 에도 적용된다. 3.4.2 부터	true | false	false
logPrefix	마이바티스가 로거(logger) 이름에 추가할 접두사 문자열을 명시	문자열	설정하지 않음
logImpl	마이바티스가 사용할 로깅 구현체를 명시 이 설정을 사용하지 않으면 마이바티스가 사용할 로깅 구현체를 자동으로 찾는다.	SLF4J | LOG4J(deprecated since 3.5.9) | LOG4J2 | JDK_LOGGING | COMMONS_LOGGING | STDOUT_LOGGING | NO_LOGGING	설정하지 않음
proxyFactory	마이바티스가 지연로딩을 처리할 객체를 생성할 때 사용할 프록시 툴을 명시	CGLIB (deprecated since 3.5.10) | JAVASSIST	JAVASSIST (마이바티스 3.3과
vfsImpl	VFS 구현체를 명시	콤마를 사용해서 VFS구현체의 패키지를 포함한 전체 클래스명	
useActualParamName	메소드 시그니처에 명시된 실제 이름으로 구문파라미터를 참조하는 것을 허용 이 기능을 사용하려면 프로젝트를 자바 8 의 `-parameters` 옵션을 사용해서 컴파일해야만 한다.(마이바티스 3.4.1이상의 버전)	true | false	true
configurationFactory	`Configuration` 인스턴스를 제공하는 클래스를 지정한다. 반환된 Configuration 인스턴스는 역직렬화 된 객체의 지연로딩 속성들을 불러오는 데 사용된다. 이 클래스는 `static Configuration getConfiguration()` 메서드를 가져야 한다. (3.2.3 부터)	타입별칭이나 패키지 경로를 포함한 클래스명	설정하지 않음
shrinkWhitespacesInSql	SQL에서 여분의 whitespace 문자들을 삭제한다. 이는 SQL의 리터럴 문자열에도 영향을 미친다. (Since 3.5.5)	true | false	false
defaultSqlProviderType	Provider method를 가지고 있는 sql provider class를 지정한다. (3.5.6 부터). 이 클래스는 sql provider annotation(예: `@SelectProvider`)의 `type` (혹은 `value`)속성이 누락되었을때 기본으로 적용된다.	타입별칭이나 패키지 경로를 포함한 클래스명	설정하지 않음
nullableOnForEach	'foreach' 태그에서 'nullable' 속성의 기본값을 지정한다. (3.5.9 부터)	true | false	false
argNameBasedConstructorAutoMapping	생성자 자동 매핑을 적용할 때 칼럼 순서가 아닌 칼럼 이름과 생성자 인수들의 이름을 기반으로 매핑한다. (3.5.10 부터)	true | false	false

위 설정을 모두 사용한 setting 엘리먼트의 예제이다:

```
<settings>
  <setting name="cacheEnabled" value="true"/>
  <setting name="lazyLoadingEnabled" value="true"/>
  <setting name="aggressiveLazyLoading" value="true"/>
  <setting name="multipleResultSetsEnabled" value="true"/>
  <setting name="useColumnLabel" value="true"/>
  <setting name="useGeneratedKeys" value="false"/>
  <setting name="autoMappingBehavior" value="PARTIAL"/>
  <setting name="autoMappingUnknownColumnBehavior" value="WARNING"/>
  <setting name="defaultExecutorType" value="SIMPLE"/>
  <setting name="defaultStatementTimeout" value="25"/>
  <setting name="defaultFetchSize" value="100"/>
  <setting name="safeRowBoundsEnabled" value="false"/>
  <setting name="safeResultHandlerEnabled" value="true"/>
  <setting name="mapUnderscoreToCamelCase" value="false"/>
  <setting name="localCacheScope" value="SESSION"/>
  <setting name="jdbcTypeForNull" value="OTHER"/>
  <setting name="lazyLoadTriggerMethods" value="equals,clone,hashCode,toString"/>
  <setting name="defaultScriptingLanguage" value="org.apache.ibatis.scripting.xmltags.XMLLanguageDriver"/>
  <setting name="defaultEnumTypeHandler" value="org.apache.ibatis.type.EnumTypeHandler"/>
  <setting name="callSettersOnNulls" value="false"/>
  <setting name="returnInstanceForEmptyRow" value="false"/>
  <setting name="logPrefix" value="exampleLogPreFix_"/>
  <setting name="logImpl" value="SLF4J | LOG4J | LOG4J2 | JDK_LOGGING | COMMONS_LOGGING | STDOUT_LOGGING | NO_LOGGING"/>
  <setting name="proxyFactory" value="CGLIB | JAVASSIST"/>
  <setting name="vfsImpl" value="org.mybatis.example.YourselfVfsImpl"/>
  <setting name="useActualParamName" value="true"/>
  <setting name="configurationFactory" value="org.mybatis.example.ConfigurationFactory"/>
</settings>
```

typeAliases

타입 별칭은 자바 타입에 대한 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 여러 군데에서 “domain.blog.Blog” 대신 사용할 수 있다.

마이바티스가 빈을 찾도록 패키지를 명시할 수 있다. 예를들면:

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

`domain.blog` 에서 빈을 검색하고 애노테이션이 없을 경우 빈의 이름이 소문자로 변환된 형태의 별칭으로 등록할 것이다. 이때 빈의 패키지정보도 제거하고 등록된다. 이를테면 `domain.blog.Author` 는 `author` 로 등록될 것이다. 만약에 `@Alias` 애노테이션을 사용한다면 이 애노테이션에서 지정한 값이 별칭으로 사용될 것이다. 아래의 예를 보라:

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

공통 자바 타입을 위한 여러 내장 타입 별칭이 존재한다. 이들은 대소문자를 구별하지 않으며, 오버로딩된 이름 때문에 원시형 타입은 특별 취급된다는 것을 주의해라.

별칭	매핑된 타입
_byte	byte
_char (since 3.5.10)	char
_character (since 3.5.10)	char
_long	long
_short	short
_int	int
_integer	int
_double	double
_float	float
_boolean	boolean
string	String
byte	Byte
char (since 3.5.10)	Character
character (since 3.5.10)	Character
long	Long
short	Short
int	Integer
integer	Integer
double	Double
float	Float
boolean	Boolean
date	Date
decimal	BigDecimal
bigdecimal	BigDecimal
biginteger	BigInteger
object	Object
date[]	Date[]
decimal[]	BigDecimal[]
bigdecimal[]	BigDecimal[]
biginteger[]	BigInteger[]
object[]	Object[]
map	Map

별칭	매핑된 타입
hashmap	HashMap
list	List
arraylist	ArrayList
collection	Collection
iterator	Iterator

typeHandlers

마이바티스가 PreparedStatement에 파라미터를 설정하고 ResultSet에서 값을 가져올때마다 TypeHandler는 적절한 자바 타입의 값을 가져오기 위해 사용된다. 다음의 표는 디폴트 TypeHandlers를 설명한다.

NOTE 3.4.5 버전부터, MyBatis는 JSR-310(Date 와 Time API) 를 기본적으로 지원한다.

타입 핸들러	자바 타입	JDBC 타입
`BooleanTypeHandler`	`java.lang.Boolean`, `boolean`	어떤 호환가능한 BOOLEAN
`ByteTypeHandler`	`java.lang.Byte`, `byte`	어떤 호환가능한 NUMERIC 또는 BYTE
`ShortTypeHandler`	`java.lang.Short`, `short`	어떤 호환가능한 NUMERIC 또는 SMALLINT
`IntegerTypeHandler`	`java.lang.Integer`, `int`	어떤 호환가능한 NUMERIC 또는 INTEGER
`LongTypeHandler`	`java.lang.Long`, `long`	어떤 호환가능한 NUMERIC 또는 BIGINT
`FloatTypeHandler`	`java.lang.Float`, `float`	어떤 호환가능한 NUMERIC 또는 FLOAT
`DoubleTypeHandler`	`java.lang.Double`, `double`	어떤 호환가능한 NUMERIC 또는 DOUBLE
`BigDecimalTypeHandler`	`java.math.BigDecimal`	어떤 호환가능한 NUMERIC 또는 DECIMAL
`StringTypeHandler`	`java.lang.String`	`CHAR`, `VARCHAR`
`ClobReaderTypeHandler`	`java.io.Reader`	-
`ClobTypeHandler`	`java.lang.String`	`CLOB`, `LONGVARCHAR`
`NStringTypeHandler`	`java.lang.String`	`NVARCHAR`, `NCHAR`
`NClobTypeHandler`	`java.lang.String`	NCLOB
`BlobInputStreamTypeHandler`	`java.io.InputStream`	-
`ByteArrayTypeHandler`	`byte[]`	어떤 호환가능한 byte 스트림 타입
`BlobTypeHandler`	`byte[]`	`BLOB`, `LONGVARBINARY`
`DateTypeHandler`	`java.util.Date`	`TIMESTAMP`
`DateOnlyTypeHandler`	`java.util.Date`	`DATE`
`TimeOnlyTypeHandler`	`java.util.Date`	`TIME`
`SqlTimestampTypeHandler`	`java.sql.Timestamp`	`TIMESTAMP`
`SqlDateTypeHandler`	`java.sql.Date`	`DATE`
`SqlTimeTypeHandler`	`java.sql.Time`	`TIME`
`ObjectTypeHandler`	Any	`OTHER`, 또는 명시하지 않는
`EnumTypeHandler`	Enumeration Type	`VARCHAR` – 문자열 호환타입.
`EnumOrdinalTypeHandler`	열거형(Enumeration) 타입	코드자체가 아니라 위치를 저장할수 있는 `NUMERIC` 또는 `DOUBLE`와 호환가능한 타입
`SqlxmlTypeHandler`	`java.lang.String`	`SQLXML`
`InstantTypeHandler`	`java.time.Instant`	`TIMESTAMP`
`LocalDateTimeTypeHandler`	`java.time.LocalDateTime`	`TIMESTAMP`
`LocalDateTypeHandler`	`java.time.LocalDate`	`DATE`
`LocalTimeTypeHandler`	`java.time.LocalTime`	`TIME`
`OffsetDateTimeTypeHandler`	`java.time.OffsetDateTime`	`TIMESTAMP`
`OffsetTimeTypeHandler`	`java.time.OffsetTime`	`TIME`
`ZonedDateTimeTypeHandler`	`java.time.ZonedDateTime`	`TIMESTAMP`
`YearTypeHandler`	`java.time.Year`	`INTEGER`
`MonthTypeHandler`	`java.time.Month`	`INTEGER`
`YearMonthTypeHandler`	`java.time.YearMonth`	`VARCHAR` or `LONGVARCHAR`

타입 핸들러	자바 타입	JDBC 타입
`JapaneseDateTypeHandler`	`java.time.chrono.JapaneseDate`	`DATE`

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

```
// ExampleTypeHandler.java
@MappedJdbcTypes(JdbcType.VARCHAR)
public class ExampleTypeHandler extends BaseTypeHandler<String> {

  @Override
  public void setNonNullParameter(PreparedStatement ps, int i, String parameter, JdbcType jdbcType) throws SQLException {
    ps.setString(i, parameter);
  }

  @Override
  public String getNullableResult(ResultSet rs, String columnName) throws SQLException {
    return rs.getString(columnName);
  }

  @Override
  public String getNullableResult(ResultSet rs, int columnIndex) throws SQLException {
    return rs.getString(columnIndex);
  }

  @Override
  public String getNullableResult(CallableStatement cs, int columnIndex) throws SQLException {
    return cs.getString(columnIndex);
  }
}
```

```
<!-- mybatis-config.xml -->
<typeHandlers>
  <typeHandler handler="org.mybatis.example.ExampleTypeHandler"/>
</typeHandlers>
```

이러한 TypeHandler를 사용하는 것은 자바 String프로퍼티와 VARCHAR파라미터 및 결과를 위해 이미 존재하는 핸들러를 오버라이드하게 될 것이다. 마이바티스는 타입을 판단하기 위해 데이터베이스의 메타데이터를 보지 않는다. 그래서 파라미터와 결과에 정확한 타입 핸들러를 매핑해야 한다. 마이바티스가 구문이 실행될때까지는 데이터 타입에 대해 모르기 때문이다.

마이바티스는 제네릭타입을 체크해서 TypeHandler로 다루고자 하는 자바타입을 알것이다. 하지만 두 가지 방법으로 이 행위를 재정의할 수 있다:

- typeHandler 엘리먼트의 `javaType` 속성 추가(예제: `javaType="String"`)
- TypeHandler클래스에 관련된 자바타입의 목록을 정의하는 `@MappedTypes` 애노테이션 추가. `javaType` 속성도 함께 정의되어 있다면 `@MappedTypes`는 무시된다.

관련된 JDBC타입은 두가지 방법으로 명시할 수 있다:

- typeHandler 엘리먼트에 `jdbcType` 속성 추가(예제: `jdbcType="VARCHAR"`).
- TypeHandler클래스에 관련된 JDBC타입의 목록을 정의하는 `@MappedJdbcTypes` 애노테이션 추가. `jdbcType` 속성도 함께 정의되어 있다면 `@MappedJdbcTypes`는 무시된다.

`ResultMap`에서 타입핸들러를 사용한다고 결정하면 마이바티스가 자바타입은 잘 처리하지만 JDBC타입은 잘 처리하지 못할 수 있다. 그래서 마이바티스는 타입핸들러를 선택하기 위해 `javaType=[TheJavaType], jdbcType=null` 조합을 사용한다. `@MappedJdbcTypes` 애노테이션의 사용은 타입핸들러의 범위를 *제한*하고 명시적으로 설정하지 않는한 `결과매핑`을 사용하지 못하도록 만든다. `결과매핑`에서 타입핸들러를 사용할수 있도록 하려면 `@MappedJdbcTypes` 애노테이션에 `includeNullJdbcType=true`를 설정하자. 자바타입을 다루기 위해 **한개의** 타입핸들러만 등록한다면 `결과매핑`에서 자바타입을 다루기 위해 이 타입핸들러를 기본적으로 사용할 것이다. 한 개의 타입핸들러만 등록할 경우 `includeNullJdbcType=true`를 설정하지 않아도 동일하게 동작한다.

마지막으로 마이바티스로 하여금 TypeHandler를 찾도록 할수 있다:

```
<!-- mybatis-config.xml -->
<typeHandlers>
  <package name="org.mybatis.example"/>
</typeHandlers>
```

JDBC타입에 대한 자동검색 기능은 애노테이션을 명시한 경우에만 가능하다는 것을 알아둘 필요가 있다.

한 개 이상의 클래스를 다루는 제네릭 TypeHandler를 만들수 있다. 파라미터로 클래스를 가져오는 생성자를 추가하고 마이바티스는 TypeHandler를 만들때 실제 클래스를 전달할 것이다.

```
//GenericTypeHandler.java
public class GenericTypeHandler<E extends MyObject> extends BaseTypeHandler<E> {

  private Class<E> type;

  public GenericTypeHandler(Class<E> type) {
    if (type == null) throw new IllegalArgumentException("Type argument cannot be null");
    this.type = type;
  }
  ...
```

`EnumTypeHandler`와 `EnumOrdinalTypeHandler`는 제네릭 TypeHandler이다. 이어서 각각을 다룬다.

Handling Enums

`Enum`을 매핑하고자 한다면 `EnumTypeHandler`나 `EnumOrdinalTypeHandler`를 사용할 필요가 있을 것이다.

예를들어 순환 방식으로 몇개의 숫자를 사용하는 순환모드를 저장할 필요가 있다고 해보자. 기본적으로 마이바티스는 `Enum` 값을 각각의 이름으로 변환하기 위해 `EnumTypeHandler`를 사용한다.

`EnumTypeHandler`는 특히 다른 핸들러와 차이가 있다. 어떤 하나의 특정 클래스를 다루지 않고 `Enum`을 확장하는 모든 클래스를 다룬다.
아무리 이름을 저장하려해도 DBA는 숫자코드를 고집할수 있다. 이름 대신 숫자코드를 저장하는 방법은 쉽다. 설정파일의 `typeHandlers`에 `EnumOrdinalTypeHandler`를 추가하자. 그러면 각각의 `RoundingMode`는 순서값을 사용해서 숫자를 매핑할 것이다.

```
<!-- mybatis-config.xml -->
<typeHandlers>
```

```
  <typeHandler handler="org.apache.ibatis.type.EnumOrdinalTypeHandler" javaType="java.math.RoundingMode"/>
</typeHandlers>
```

같은 `Enum` 을 사용해서 어떤 곳에는 문자열로 매핑하고 다른 곳에는 숫자로 매핑해야 한다면 무엇을 해야 하나?

자동매퍼는 `EnumOrdinalTypeHandler` 를 자동으로 사용할 것이다. 그래서 평범한 순서를 나타내는 `EnumTypeHandler` 를 사용하고자 한다면 SQL구문에 사용할 타입핸들러를 명시적으로 설정한다.

(매퍼 파일은 다음 절까지는 다루지 않는다. 그래서 문서를 보면서 처음 봤다면 일단 이 부분은 건너띄고 다음에 다시 볼수도 있다.)

```
<!DOCTYPE mapper
    PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "https://mybatis.org/dtd/mybatis-3-mapper.dtd">

<mapper namespace="org.apache.ibatis.submitted.rounding.Mapper">
  <resultMap type="org.apache.ibatis.submitted.rounding.User" id="usermap">
    <id column="id" property="id"/>
    <result column="name" property="name"/>
    <result column="funkyNumber" property="funkyNumber"/>
    <result column="roundingMode" property="roundingMode"/>
  </resultMap>

  <select id="getUser" resultMap="usermap">
    select * from users
  </select>
  <insert id="insert">
      insert into users (id, name, funkyNumber, roundingMode) values (
        #{id}, #{name}, #{funkyNumber}, #{roundingMode}
      )
  </insert>

  <resultMap type="org.apache.ibatis.submitted.rounding.User" id="usermap2">
    <id column="id" property="id"/>
    <result column="name" property="name"/>
    <result column="funkyNumber" property="funkyNumber"/>
    <result column="roundingMode" property="roundingMode" typeHandler="org.apache.ibatis.type.EnumTypeHandler"/>
  </resultMap>
  <select id="getUser2" resultMap="usermap2">
    select * from users2
  </select>
  <insert id="insert2">
      insert into users2 (id, name, funkyNumber, roundingMode) values (
        #{id}, #{name}, #{funkyNumber}, #{roundingMode, typeHandler=org.apache.ibatis.type.EnumTypeHandler}
      )
  </insert>

</mapper>
```

여기서 사용한 select구문에서는 `resultType` 대신에 `resultMap`을 사용해야 한다는 점을 알아두자.

objectFactory

매번 마이바티스는 결과 객체의 인스턴스를 만들기 위해 ObjectFactory를 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. 예를들면:

```
// ExampleObjectFactory.java
public class ExampleObjectFactory extends DefaultObjectFactory {
  @Override
  public <T> T create(Class<T> type) {
    return super.create(type);
  }

  @Override
  public <T> T create(Class<T> type, List<Class<?>> constructorArgTypes, List<Object> constructorArgs) {
    return super.create(type, constructorArgTypes, constructorArgs);
  }

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

  @Override
  public <T> boolean isCollection(Class<T> type) {
    return Collection.class.isAssignableFrom(type);
  }}
```

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

ObjectFactory인터페이스는 매우 간단한다. 두 개의 create메소드를 가지고 있으며 하나는 디폴트 생성자를 처리하고 다른 하나는 파라미터를 가진 생성자를 처리한다. 마지막으로 setProperties 메소드는 ObjectFactory를 설정하기 위해 사용될 수 있다. objectFactory엘리먼트에 정의된 프로퍼티는 ObjectFactory인스턴스가 초기화된 후 setProperties에 전달될 것이다.

plugins

마이바티스는 매핑 구문을 실행하는 어떤 시점에 호출을 가로챈다. 기본적으로 마이바티스는 메소드 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 메소드는 각각 메소드 시그니처를 통해 찾을 수 있고 소스코드는 마이바티스 릴리즈 파일에서 찾을 수 있다. 오버라이드할 메소드의 행위를 이해해야만 한다. 주어진 메소드의 행위를 변경하거나 오버라이드하고자 한다면 마이바티스의 핵심기능에 악영향을 줄 수도 있다. 이러한 로우레벨 클래스와 메소드들은 주의를 해서 사용해야 한다.

플러그인을 사용하도록 처리하는 방법은 간단하다. Interceptor인터페이스를 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 {
  private Properties properties = new Properties();

  @Override
  public Object intercept(Invocation invocation) throws Throwable {
    // implement pre processing if need
    Object returnObject = invocation.proceed();
    // implement post processing if need
    return returnObject;
  }

  @Override
  public void setProperties(Properties properties) {
    this.properties = properties;
  }
}
```

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

위 플러그인은 매핑된 구문의 로우레벨 실행을 책임지는 내부 객체인 Executor인스턴스의 “update” 메소드 호출을 모두 가로챌것이다.

참고 **설정파일 오버라이드하기**

플러그인을 사용해서 마이바티스 핵심 행위를 변경하기 위해 Configuration클래스 전체를 오버라이드 할 수 있다. 이 클래스를 확장하고 내부 메소드를 오버라이드하고 SqlSessionFactoryBuilder.build(myConfig)메소드에 그 객체를 넣어주면 된다. 다시 얘기하지만 이 작업은 마이바티스에 큰 영향을 줄수 있으니 주의해서 해야 한다.

environments

마이바티스는 여러 개의 환경으로 설정할 수 있다. 여러가지 이유로 여러 개의 데이터베이스에 SQL Map을 적용하는데 도움이 된다. 예를들어, 개발, 테스트, 리얼 환경을 위해 별도의 설정을 가지거나 같은 스키마를 여러 개의 DBMS 제품을 사용할 경우들이다. 그 외에도 많은 경우가 있을 수 있다.

중요하게 기억해야 할 것은 다중 환경을 설정할 수는 있지만 SqlSessionFactory 인스턴스마다 한 개만 사용할 수 있다는 것이다.

두 개의 데이터베이스에 연결하고 싶다면 SqlSessionFactory 인스턴스를 두 개 만들 필요가 있다. 세 개의 데이터베이스를 사용한다면 역시 세 개의 인스턴스를 필요로 한다. 기억하기 쉽게

- **데이터베이스별로 하나의 SqlSessionFactory**

환경을 명시하기 위해 SqlSessionFactoryBuilder에 옵션으로 추가 파라미터를 주면 된다. 환경을 선택하는 두가지 시그니처는

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

environment 파라미터가 없으면 디폴트 환경이 로드된다.

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

environments 엘리먼트는 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

마이바티스는 두 가지 타입의 TransactionManager를 제공한다.

- JDBC - 이 설정은 간단하게 JDBC 커밋과 롤백을 처리하기 위해 사용된다. 트랜잭션의 스코프를 관리하기 위해 dataSource 로 부터 커넥션을 가져온다. By default, it enables auto-commit when closing the connection for compatibility with some drivers. However, for some drivers, enabling auto-commit is not only unnecessary, but also is an expensive operation. So, since version 3.5.10, you can skip this step by setting the "skipSetAutoCommitOnClose" property to true. For example:

```
<transactionManager type="JDBC">
  <property name="skipSetAutoCommitOnClose" value="true"/>
</transactionManager>
```

- 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 false로 설정하라. 예를 들면:

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

참고 당신은 마이바티스를 스프링과 함께 사용하는 경우에는 구성할 필요가 없습니다 스프링 모듈 자체의 설정 때문에 어떤 TransactionManager 이전에 설정된 구성을 무시합니다.

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

```
public interface TransactionFactory {
  default void setProperties(Properties props) { // Since 3.5.2, change to default method
    // NOP
  }
  Transaction newTransaction(Connection conn);
  Transaction newTransaction(DataSource dataSource, TransactionIsolationLevel level, boolean autoCommit);
}
```

XML에 설정된 프로퍼티는 인스턴스를 만든 뒤 setProperties()메소드에 전달할 것이다. 당신의 구현체가 Transaction구현체를 만들 필요가 있을 것이다.:

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

이 두 개의 인터페이스를 사용하여 마이바티스가 트랜잭션을 처리하는 방법을 완벽하게 정의할 수 있다.

dataSource

dataSource엘리먼트는 표준 JDBC DataSource인터페이스를 사용하여 JDBC Connection객체의 소스를 설정한다.

- 대부분의 마이바티스 애플리케이션은 예제처럼 dataSource를 설정할 것이다.

여기엔 3 가지의 내장된 dataSource타입이 있다.

UNPOOLED - 이 구현체는 매번 요청에 대해 커넥션을 열고 닫는 간단한 DataSource이다. 조금 늦긴 하지만 성능을 크게 필요로 하지 않는 간단한 애플리케이션을 위해서는 괜찮은 선택이다. UNPOOLED DataSource에는 다음과 같은 속성이 있습니다.

- `driver` - JDBC드라이버의 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
- `defaultNetworkTimeout` – The default network timeout value in milliseconds to wait for the database operation to complete. See the API documentation of `java.sql.Connection#setNetworkTimeout()` for details.

필수는 아니지만 선택적으로 데이터베이스 드라이버에 프로퍼티를 전달할 수도 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)메소드를 통해서도 프로퍼티를 설정할 수 있다.

POOLED - DataSource에 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 DataSource에 비해 많은 프로퍼티를 설정할 수 있다.

- `poolMaximumActiveConnections` - 주어진 시간에 존재할 수 있는 활성화된(사용중인) 커넥션의 수. 디폴트는 10이다.
- `poolMaximumIdleConnections` - 주어진 시간에 존재할 수 있는 유휴 커넥션의 수
- 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 설정. 디폴트는 20000ms(20 초)
- `poolMaximumLocalBadConnectionTolerance` – 이것은 모든 쓰레드에 대해 bad Connection이 허용되는 정도에 대한 낮은 수준의 설정입니다. 만약 쓰레드가 bad connection 을 얻게 되어도 유효한 또 다른 connection 을 다시 받을 수 있습니다. 하지만 재시도 횟수는 `poolMaximumIdleConnections` 과 `poolMaximumLocalBadConnectionTolerance` 의 합보다 많아야 합니다. 디폴트는 3 이다. (3.4.5 부터)
- `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 설정해야 한다. 디폴트는 false 이다.
- `poolPingConnectionsNotUsedFor` - poolPingQuery가 얼마나 자주 사용될지 설정한다. 필요이상의 핑을 피하기 위해 데이터베이스의 타임아웃 값과 같을 수 있다. 디폴트는 0이다. 디폴트 값은 poolPingEnabled가 true일 경우에만 모든 커넥션이 매번 핑을 던지는 값이다.

JNDI - 이 DataSource 구현체는 컨테이너에 따라 설정이 변경되며 JNDI 컨텍스트를 참조한다. 이 DataSource 는 오직 두 개의 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_source프로퍼티가 InitialContext에서 직접 찾을 것이다.
- `data_source` - DataSource인스턴스의 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 DataSource설정과 유사하게 다음처럼 “env.”를 접두어로 프로퍼티를 전달할 수 있다.

- `env.encoding=UTF8`

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

`org.apache.ibatis.datasource.DataSourceFactory` 인터페이스를 구현해서 또다른 DataSource구현체를 만들수 있다.

```
public interface DataSourceFactory {
  void setProperties(Properties props);
```

```
  DataSource getDataSource();
}
```

`org.apache.ibatis.datasource.unpooled.UnpooledDataSourceFactory`는 새로운 데이터소스를 만들기 위한 상위 클래스처럼 사용할 수 있다. 예를 들면 다음의 코드를 사용해서 C3P0를 사용할 수 있다.

```
import org.apache.ibatis.datasource.unpooled.UnpooledDataSourceFactory;
import com.mchange.v2.c3p0.ComboPooledDataSource;

public class C3P0DataSourceFactory extends UnpooledDataSourceFactory {

  public C3P0DataSourceFactory() {
    this.dataSource = new ComboPooledDataSource();
  }
}
```

마이바티스가 호출할 setter메소드가 사용하는 프로퍼티를 추가해서 설정한다. 다음의 설정은 PostgreSQL데이터베이스에 연결할때 사용한 샘플 설정이다.

```
<dataSource type="org.myproject.C3P0DataSourceFactory">
  <property name="driver" value="org.postgresql.Driver"/>
  <property name="url" value="jdbc:postgresql:mydb"/>
  <property name="username" value="postgres"/>
  <property name="password" value="root"/>
</dataSource>
```

databaseIdProvider

마이바티스는 데이터베이스 제품마다 다른 구문을 실행할 수 있다. 여러 개의 데이터베이스 제품을 가진 업체 제품은 `databaseId` 속성을 사용한 매핑된 구문을 기반으로 지원한다. 마이바티스는 `databaseId` 속성이 없거나 `databaseId` 속성을 가진 모든 구문을 로드한다. 같은 구문인데 하나는 `databaseId` 속성이 있고 하나는 `databaseId` 속성이 없을때 뒤에 나온 것이 무시된다. 다중 지원을 사용하기 위해서는 mybatis-config.xml파일에 다음처럼 `databaseIdProvider`를 추가하라:

```
<databaseIdProvider type="DB_VENDOR" />
```

DB_VENDOR구현체 databaseIdProvider는 `DatabaseMetaData#getDatabaseProductName()`에 의해 리턴된 문자열로 databaseId를 설정한다. 이때 리턴되는 문자열이 너무 길거나 같은 제품의 서로 다른 버전으로 다른 값을 리턴하는 경우 다음처럼 프로퍼티를 추가해서 짧게 처리할 수도 있다:

```
<databaseIdProvider type="DB_VENDOR">
  <property name="SQL Server" value="sqlserver"/>
  <property name="DB2" value="db2"/>
  <property name="Oracle" value="oracle" />
</databaseIdProvider>
```

프로퍼터가 제공되면 DB_VENDOR databaseIdProvider는 리턴된 데이터베이스 제품명에서 찾은 첫번째 프로퍼티값이나 일치하는 프로퍼티가 없다면 "null" 을 찾을 것이다. 이 경우 `getDatabaseProductName()`가 "Oracle (DataDirect)"를 리턴한다면 databaseId는 "oracle"로 설정될 것이다.

`org.apache.ibatis.mapping.DatabaseIdProvider` 인터페이스를 구현해서 자신만의 DatabaseIdProvider를 빌드하고 mybatis-config.xml파일에 등록할 수 있다:

```
public interface DatabaseIdProvider {
  default void setProperties(Properties p) { // Since 3.5.2, change to default method
    // NOP
  }
  String getDatabaseId(DataSource dataSource) throws SQLException;
}
```

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

```
<!-- 클래스패스의 상대경로의 리소스 사용 -->
<mappers>
  <mapper resource="org/mybatis/builder/AuthorMapper.xml"/>
  <mapper resource="org/mybatis/builder/BlogMapper.xml"/>
  <mapper resource="org/mybatis/builder/PostMapper.xml"/>
</mappers>
```

```
<!-- 절대경로의 url을 사용 -->
<mappers>
  <mapper url="file:///var/mappers/AuthorMapper.xml"/>
  <mapper url="file:///var/mappers/BlogMapper.xml"/>
  <mapper url="file:///var/mappers/PostMapper.xml"/>
</mappers>
```

```
<!-- 매퍼 인터페이스를 사용 -->
<mappers>
  <mapper class="org.mybatis.builder.AuthorMapper"/>
  <mapper class="org.mybatis.builder.BlogMapper"/>
  <mapper class="org.mybatis.builder.PostMapper"/>
</mappers>
```

```
<!-- 매퍼로 패키지내 모든 인터페이스를 등록 -->
<mappers>
  <package name="org.mybatis.builder"/>
</mappers>
```