iBATIS

데이터 매퍼
(a.k.a SQL Maps)

Version 2.0

개발자 가이드

2006년 **11**월 **30**일

번역 **:** 이동국**(fromm0@gmail.com)**
오타 및 오역은 위 메일주소로 보내주시기 바랍니다**.**

목차

- 소개
- 데이터 매퍼
- 설치
  - 1.x에서 업그레이드하기
- SQL Map XML 설정파일
  - <properties> 요소
  - <settings> 요소
  - <resultObjectFactory> 요소
  - <typeAlias> 요소
  - <transactionManager> 요소
  - <dataSource> 요소
  - <sqlMap> 요소
- SQL Map XML 파일
- 매핑 구문
  - 구문 타입
  - SQL
  - SQL 재사용하기
  - 자동 생성 키
  - 저장 프로시저
- 파라미터 맵과 인라인 파라미터
  - 인라인 파라미터 맵
  - 원시타입 파라미터
  - Map 타입 파라미터
- 대체 문자열
- 결과맵
  - 내포하는 결과맵
  - 원시타입의 결과
  - 복합 프라퍼티
  - N+1 조회(1:1) 피하기
  - 복합 Collection 프라퍼티
  - N+1 조회(1:M and M:N) 피하기
  - 복합키 또는 다중 복합 파라미터 프라퍼티
- 파라미터 맵과 결과 맵을 지원하는 타입
  - 사용자 정의 타입 핸들러 생성하기
- 캐싱된 매핑 구문 결과
  - 읽기전용 대 읽기/쓰기
  - 직렬화가능한 읽기/쓰기 캐시
  - 캐시 타입
- 동적인 매핑 구문
  - Dynamic 요소
  - 이항연산 요소
  - 단항연산 요소
  - 다른 요소
- 간단한 동적 SQL요소
- 데이터 매퍼로 프로그래밍하기: The API
  - 설정
  - 트랜잭션
  - 다중 쓰레드 프로그래밍
  - iBATIS 클래스로딩
  - 일괄처리
  - SqlMapClient API를 통해 구문 실행하기
- SqlMap 로깅하기
- 한 페이지의 자바빈즈 과정
- Resources (com.ibatis.common.resources.*)
  - Resources 국제화
- SimpleDataSource (com.ibatis.common.jdbc.*)

소개

iBATIS 데이터 매퍼 프레임워크는 당신이 관계형 데이터베이스에 접근할 때 필요한 자바코드를 현저하게 줄일수 있도록 도와줄것이다. iBATIS는 간단한 XML서술자를 사용해서 간단하게 자바빈즈를 SQL 구문에 매핑시킨다. 간단함 ***(Simplicity)***이란 다른 프레임워크와 객체관계매핑툴에 비해 iBATIS의 가장 큰 장점이다. iBATIS 데이터 매퍼를 사용하기 위해서 당신은 자바빈즈와 XML 그리고 SQL에 친숙할 필요가 있다. 여기에서는 추가적으로 배워야 할것이 거의 없고 테이블을 조인하거나 복잡한 쿼리문을 수행하기 위해 필요한 복잡한 스키마도 없다. 데이터 매퍼를 사용하면 당신은 실제 SQL문의 모든 기능을 그대로 사용할수 있다.

데이터 매퍼 **(com.ibatis.sqlmap.*)**

개념

iBATIS 데이터 매퍼 API는 프로그래머에게 자바빈즈 객체를 PreparedStatement파라미터와 ResultSets으로 쉽게 매핑할수 있도록 한다. 데이터 매퍼의 기본적인 생각은 간단함(simple)이다. 이는 자바코드의 20%를 사용하여 JDBC기능의 80%를 제공하는 간단한 프레임워크라는 뜻이다.

이것은 어떻게 작동하는가**.?**

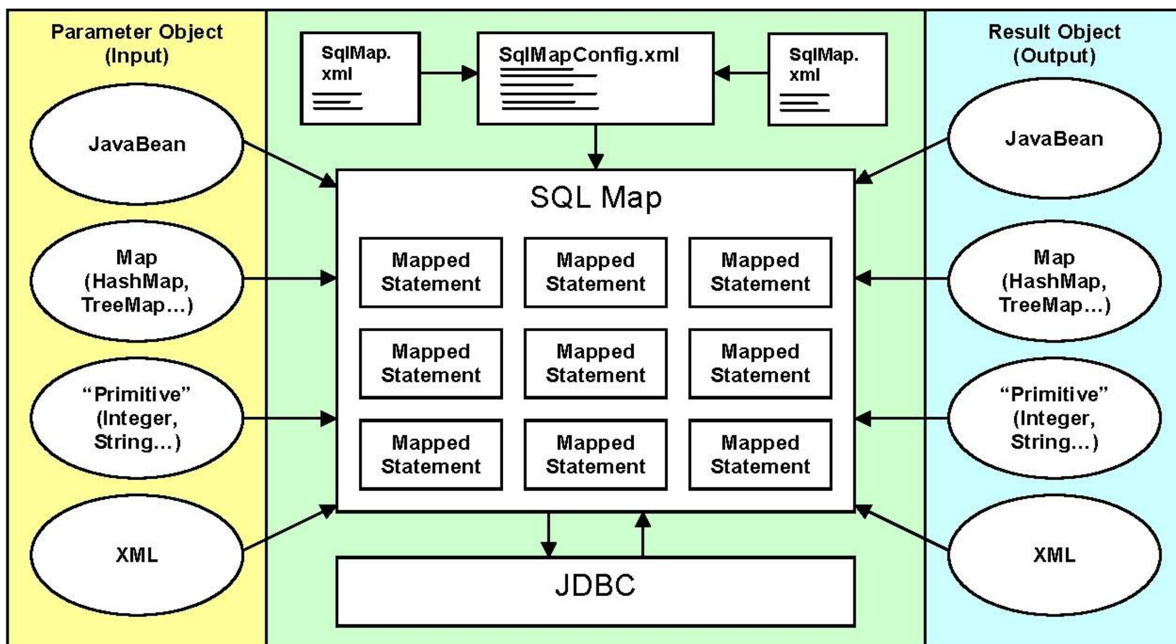
데이터 매퍼는 자바빈즈, Map구현체, 원시래퍼타입(String, Integer…) 그리고 SQL문을 위한 XML문서를 매핑하기 위한 XML서술자를 사용하는 매우 간단한 프레임워크를 제공한다. 다음은 생명주기에 대한 높은 레벨의 서술이다.

1) 파라미터(자바빈즈, Map 또는 원시래퍼)로써 객체를 제공한다. 파라미터 객체는 update문에서 입력값을 셋팅하기 위해 사용되거나 쿼리문의 where절을 셋팅하기 위해서 사용된다.

2) 매핑된 구문을 실행한다. 이 단계는 마법이 일어나는 곳이다. 데이터 매퍼프레임워크는 PreparedStatement 인스턴스를 생성할것이고 제공된 파라미터객체를 사용해서 파라미터를 셋팅한다. 그리고 구문를 실행하고 ResultSet으로부터 결과 객체를 생성한다.

3) update의 경우에 영향을 미친 rows의 숫자를 반환한다. 조회작업인 경우에 한 개(single)의 객체 또는 컬렉션 객체를 반환한다. 파라미터처럼 결과 객체는 자바빈즈, Map 원시타입래퍼또는 XML이 될수 있다.

밑의 다이어그램은 그림으로 다시 표현한것이다.



설치

iBATIS 데이터 매퍼 프레임워크를 설치는 클래스패스에 JAR파일을 넣어주는 간단한 작업이다. 이 클래스패스는 JVM시작시 지정된 클래스패스(*java* -cp 인자로 지정된)나 웹 애플리케이션의 /WEB-INF/lib 디렉토리가 될수도 있다. 자바 클래스패스에 대한 충분한 설명은 이 문서의 범위를 넘어선다. 자바와/또는 클래스패스가 처음이라면, 다음의 자료를 참고하라.

> http://java.sun.com/j2se/1.4/docs/tooldocs/win32/classpath.html
> http://java.sun.com/j2se/1.4.2/docs/api/java/lang/ClassLoader.html
> http://java.sun.com/j2se/1.4.2/docs/

iBATIS는 하나의 JAR파일만을 가진다. 파일명은 다음과 같은 형태를 가진다.

> ibatis-*version*.*build*.jar

예를 들면 ibatis-2.3.0.677.jar 과 같은 형태이다.
대개 애플리케이션의 클래스패스에 이 하나의 JAR파일을 두는 것으로 충분하다.

JAR 파일과 의존성

프레임워크가 너무 많은 의존성을 가진다면 이것은 애플리케이션이나 다른 프레임워크에 통합되기 힘들게 만든다. 2.0의 중요한 핵심사항은 의존성관리와 제거의 중점을 두었다. 그러므로 만약 당신이 jdk1.4를 사용한다면 실제 의존적인 것은 Jakarta Commons Logging 프레임워크뿐이다. 이 추가적인 JAR파일은 아래의 웹사이트에서 다운로드 받을수 있다. 그들은 기능에 의해 분류된다. 다음은 추가적인 패키지를 사용할 때 필요한 것들의 목록이다.

설명	사용할 시점	다운로드 위치
레거시 JDK 지원	만약에 당신이 JDK1.4보다 하위 버전을 사용하고 당신의 애플리케이션 서버가 이런 JAR파일을 제공하지 않는다면 당신은 이런 옵션 패키지가 필요할것이다.	**JDBC 2.0 Extensions** http://java.sun.com/products/jdbc/download.html **JTA 1.0.1a** http://java.sun.com/products/jta/ **Xerces 2.4.0** http://xml.apache.org/xerces2-j/
iBATIS 의 이전 버전 호환	당신이 iBATIS의 예전버전 DAO(1.x)프레임워크를 사용하고 있거나 SQL Maps(1.x)의 예전버전을 사용하고 있다면 이 디렉토리의 JAR파일을 간단하 포함시킴으로써 계속 작업을 할수 있다.	**iBATIS DAO 1.3.1** http://sourceforge.net/projects/ibatisdb/
런타임 바이트코드 향상	만약 당신이 늦은(lazy) 로딩과 성능에 대해 고려하기 위한 CGLIB2.0 bytecode 개선을 사용하길 원한다면	**CGLIB 2.0** http://cglib.sf.net
DataSource 구현체	당신이 Jakarta DBCP Connection pool 을 사용하길 원한다면	**DBCP 1.1** http://jakarta.apache.org/commons/dbcp/
분산 캐싱	중앙집중적이거나 분산 캐싱 지원을 위한 OSCache를 사용하길 원한다면	**OSCache 2.0.1** http://www.opensymphony.com/oscache/
로깅 솔루션	Log4J 로깅을 사용하길 원한다면	**Log4J 1.2.8** http://logging.apache.org/log4j/docs/
로깅 솔루션	Jakarta Commons Logging를 사용하고자 한다면	**Jakarta Commons Logging** **http://jakarta.apache.org/commons/logging**

1.x 에서 업그레이드하기

당신은 업그레이드 할것인가**.?**

만약 당신이 업그레이드를 시도한다면 결정할수 있는 가장 좋은 방법이다. 여기에 몇가지 업그레이드 절차가 있다.

1. 버전 2.0은 1.x릴리즈와 거의 완벽한 호완성을 가지도록 유지되었다. 그래서 몇몇 사람들에게는 단순히 JAR파일만 교체하는것으로 충분할 것이다. 이것은 최소한의 이득을 발생시키지만 가장 간단하다. 당신은 당신의 XML파일이나 자바코드를 변경할 필요가 없다. 몇몇 모순되는것들이 발견될지도 모른다.

2. 두번째는 당신의 XML파일을 2.0스펙에 적합하도록 변경하는것이다. 하지만 이는 1.x 자바 API를 그대로 사용한다. 적은 호환성이슈내 안전한 해결법은 매핑파일 사이에 발생한다. Ant작업은 당신을 위해 XML파일을 변환하기 위해서 프레임워크에 포함된다.

3. 세번째 옵션은 당신의 XML파일과 자바코드를 변환하는 것이다. 자바코드를 변환하기 위한 툴은 없다. 그래서 이것은 손으로 직접해야 한다.

4. 마지막 옵션은 전체를 업그레이드 하지 않는것이다. 만약에 당신이 어렵다고 느낀다면 1.x릴리즈에서 시스템이 작동하는 것을 두려워하지 마라. 당신의 오래된 애플리케이션을 그대로 놔두는 것은 나쁜 생각이 아니다. 만약에 오래된 애플리케이션이 인식적인 면에서 제대로 리팩토링되어 있다면 당신은 SQL Maps를 업그레이드 잘 할수 있을 것이다.

1.x에서 **2.x**으로 **XML**설정파일 변환하기

2.0프레임워크는 Ant빌드시스템을 통해 수행되는 XML문서 변환기를 포함한다. 당신의 XML문서를 변환하는 것은 1.x코드가 작동중에 자동으로 오래된 XML파일을 변환하는 것처럼 옵션적이다. 여전히 당신이 업그레이드를 함으로써 편안하게 당신의 파일을 변환하는 것이 좋은 생각이다. 당신은 다소 적은 호환적인 이슈를 경험할것이고 새로운 기능중 몇 개의 장점을 얻을수 있을것이다(비록 당신이 1.x자바 API을 사용하더라도.).

Ant작업은 당신의 build.xml파일내에 다음과 비슷하게 보일것이다.

```
<taskdef name="convertSqlMaps"
     classname="com.ibatis.db.sqlmap.upgrade.ConvertTask"
     classpathref="classpath"/>


<target name="convert">
  <convertSqlMaps todir="D:/targetDirectory/" overwrite="true">
    <fileset dir="D/sourceDirectory/">
      <include name="**/maps/*.xml"/>
    </fileset>
  </convertSqlMaps>

</target>
```

당신이 보는것처럼 이것은 Ant 복사 작업과 거의 같고 사실 이것은 Ant복사 작업을 확장한것이다. 그래서 당신은 복사하는 작업을 하는 어떤것도 할 수 있다.

JAR 파일들**:** 예전것을 빼내고 새것을 넣자**.**
업그레이드를 할 때 존재하는(예전의) iBATIS파일과 의존적인 것들을 모두 지우고 새 파일을 대체하는것이 좋은 생각이다. 여전히 필요한 당신의 다른 컴포넌트또는 프레임워크를 모두 지우지 않도록 주의해라. JAR파일의 대부분은 당신 환경에 의존적이다. JAR파일과 의존적인것에 대해서는 위에서 서술된 것을 보아라.

다음의 테이블은 예전 파일과 새 파일을 목록화 한다.

Old Files	**New Files**
ibatis-db.jar *1.2.9b* 버전에서부터 이 파일은 다음의 *3*개의 파일로 분리되었다*.* ibatis-common.jar ibatis-dao.jar ibatis-sqlmap.jar	Ibatis-*version*.*build*.jar ***(***필수***)***

commons-logging.jar commons-logging-api.jar commons-collections.jar commons-dbcp.jar commons-pool.jar oscache.jar jta.jar jdbc2_0-stdext.jar xercesImpl.jar xmlParserAPIs.jar jdom.jar	commons-logging-1-0-3.jar ***(필수)*** commons-collections-2-1.jar ***(옵션)*** commons-dbcp-1-1.jar ***(옵션)*** commons-pool-1-1.jar ***(옵션)*** oscache-2-0-1.jar ***(옵션)*** jta-1-0-1a.jar ***(옵션)*** jdbc2_0-stdext.jar ***(옵션)*** xercesImpl-2-4-0.jar ***(옵션)*** xmlParserAPIs-2-4-0.jar ***(옵션)*** xalan-2-5-2.jar ***(옵션)*** log4j-1.2.8.jar ***(옵션)*** cglib-full-2-0-rc2.jar ***(옵션)***

이 가이드의 나머지는 당신이 *SQL Maps*를 사용하는것에 대해 소개할것이다.

SQL Map XML설정파일
(http://ibatis.apache.org/dtd/sql-map-config-2.dtd)

SQL Maps는 데이터소스, 데이터 맵퍼에 대한 설정, 쓰레드 관리와 같은 SQL Maps와 다른 옵션에 대한 설정을 제공하는 중앙집중적인 XML설정 파일을 사용해서 설정된다. 다음은 SQL Maps설정파일의 예이다.

SqlMapConfig.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMapConfig
   PUBLIC "-//ibatis.apache.org//DTD SQL Map Config 2.0//EN"
   "http://ibatis.apache.org/dtd/sql-map-config-2.dtd">

<!--다음은 정확한 XML헤더를 위한 필수값이다. -->
<sqlMapConfig>

 <!--여기서 명시된 파일내 프라퍼티(name=value)는 이 설정파일내 고정자(placeholder)에 의해 사용될수 있다.
 (이를테면. "${driver}". 이 파일은 클래스패스에 상대적이고 선택적인 사항이다. -->
 <properties resource=" examples/sqlmap/maps/SqlMapConfigExample.properties " />

 <!--이 셋팅은 가장 기본적으로는 트랜잭션 관리를 하며 SqlMapClient설정 상세를 제어한다.
 이것들은 모두 선택적이다.  -->
 <settings
  cacheModelsEnabled="true"
  enhancementEnabled="true"
  lazyLoadingEnabled="true"
  maxRequests="128"
  maxSessions="10"
  maxTransactions="5"
  useStatementNamespaces="false"
  defaultStatementTimeout="5"
  statementCachingEnabled="true"
  classInfoCacheEnabled="true"
  />

 <!-- 이 요소는 iBATIS가 결과 객체를 생성하기 위해 사용할 factory클래스를 선언한다.
         이 요소는 선택사항이며 뒷 부분에서 상세하게 설명한다. -->
 <resultObjectFactory type="com.mydomain.MyResultObjectFactory" >
    <property name="someProperty" value="someValue"/>
 </resultObjectFactory>

 <!--긴 전체 경로를 포함한 클래스명을 위한 좀더 짧은 이름을 사용하기 위한 별칭을 타이핑한다. -->
 <typeAlias alias="order" type="testdomain.Order"/>

 <!--SimpleDataSource 를 이용한  SQL Map를 사용하기 위한 데이터소스 설정.
```

```
위 자원으로 부터 프라퍼티 사용에 주의. -->
<transactionManager type="JDBC" >
 <dataSource type="SIMPLE">
  <property name="JDBC.Driver" value="${driver}"/>
  <property name="JDBC.ConnectionURL" value="${url}"/>
  <property name="JDBC.Username" value="${username}"/>
  <property name="JDBC.Password" value="${password}"/>
  <property name="JDBC.DefaultAutoCommit" value="true" />
  <property name="Pool.MaximumActiveConnections" value="10"/>
  <property name="Pool.MaximumIdleConnections" value="5"/>
  <property name="Pool.MaximumCheckoutTime" value="120000"/>
  <property name="Pool.TimeToWait" value="500"/>
  <property name="Pool.PingQuery" value="select 1 from ACCOUNT"/>
  <property name="Pool.PingEnabled" value="false"/>
  <property name="Pool.PingConnectionsOlderThan" value="1"/>
  <property name="Pool.PingConnectionsNotUsedFor" value="1"/>

 </dataSource>
</transactionManager>


<!--이 SQL map에 의해 로드되는 모든  SQL Map파일을 인식한다. 경로는 클래스패스에 상대적이다.  -->
<sqlMap resource="examples/sqlmap/maps/Person.xml" />

</sqlMapConfig>
```

이 문서의 다음 부분은 SQL Maps설정파일의 다양한 부분을 논의한다.

<properties> 요소

SQL Maps은 SQL Maps XML설정파일과 함께 속하는 표준적인 자바 속성파일(name=value)을 지정하는 하나의 <properties>요소를 가질 수 있다. 그렇게 함으로써 속성파일내에 각각의 이름지어진 값들은 SQL Maps설정파일내에 참조될수 있는 변수가 될수 있고 모든 SQL Maps는 내부에서 참조된다. 예를 들면 속성파일이 다음을 포함한다면

driver=org.hsqldb.jdbcDriver

그러면 SQL Maps설정파일또는 설정문서에 의해 참조되는 각각의 SQL Maps는 ${driver} 형태로 사용가능하고 org.hsqldb.jdbcDriver라는 값이 참조된다. 예를 들면

<property name="JDBC.Driver" value="**${driver}**"/>

이것은 빌드되거나 테스트 그리고 배치되는 동안 편리하게 된다. 이것은 다중 환경이나 설정파일을 위한 자동화툴을 사용하는 당신의 애플리케이션을 쉽게 인식하도록 한다. 프라퍼티는 클래스패스나 어떤 유효한 URL로부터 로드될수 있다. 예를 들면 고정된 파일경로를 위해 다음처럼 사용한다.

<properties url=”file:///c:/config/my.properties” />

<settings> 요소

<settings> 요소는 XML파일을 빌드하는 SqlMapClient 인스턴스를 위해 다양한 옵션과 최적화를 설정하도록 한다. setting요소와 그것의 모든 속성값은 모두 옵션적이다. 제공되는 속성값과 그것들의 다양한 행위는 다음의 테이블에서 서술된다.

maxRequests	이것은 한꺼번에 SQL문을 수행할수 있는 쓰레드의 수이다. 셋팅값보다 많은 쓰레드는 다른 쓰레드가 수행을 완료할때까지 블록된다. 다른 DBMS는 다른 제한을 가진다. 이것은 최소한 10개의 maxTransactions이고 언제나 maxSessions과 maxTransactions보다 크다. 종종 동시요청값의 최대치를 줄이면 성능향상을 보여준다. 예*:* *maxRequests="256"* *Default: 512*
maxSessions	이것은 주어진 시간동안 활성될수 있는 세션의 수이다. 세션은 명시적으로 주어질수도 있고 프로그램적으로 요청될수도 있고 쓰레드가 SqlMapClient인스턴스를 사용할때마다 자동적으로 생성될수도 있다. 이것은 언제나 maxTransaction보다 같거나 커야 하고 maxRequests보다 작아야 한다. 동시세션값의 최대치를 줄이면 전체적인 메모리사용량을 줄일수 있다. 예*:* *maxSessions="64"* *Default: 128*
maxTransactions	이것은 한꺼번에 SqlMapClient.startTransaction()에 들어갈수 있는 쓰레드의 최대갯수이다. 셋팅값보다 많은 쓰레드는 다른 쓰레드가 나올때까지 블록된다. 다른 DBMS는 다른 제한을 가진다. 이 값은 언제나 maxSessions보다 작거나 같아야 하고 maxRequests보다 작아야 한다. 종종 동시트랜잭션의 최대치를 줄이면 성능향상을 보여준다. 예*:* *maxTransactions="16"* *Default: 32*
cacheModelsEnabled	이 셋팅은 SqlMapClient 를 위한 모든 캐쉬모델을 가능하게 하거나 가능하지 않게 한다. 이것은 디버깅시 도움이 된다. 예*:* *cacheModelsEnabled="true"* *Default: true (enabled)*
lazyLoadingEnabled	이 셋팅은 SqlMapClient 를 위한 모든 늦은(lazy)로딩을 가능하게 하거나 가능하지 않게 한다. 이것은 디버깅시 도움이 된다. 예*:* *lazyLoadingEnabled="true"* *Default: true (enabled)*
enhancementEnabled	이 셋팅은 향상된 늦은(lazy)로딩처럼 최적화된 자바빈즈 속성 접근을 위해 런타임시 바이트코드 향상을 가능하게 한다. 예*:* *enhancementEnabled="true"* *Default: false (disabled)*
useStatementNamespaces	이 셋팅을 가능하게 하면 당신은 sqlmap이름과 statement이름으로 구성된 전체적인 이름(fully qualified name)으로 매핑 구문를 참조해야 한다. 예를 들면: queryForObject("sqlMapName.statementName"); 예제*:* *useStatementNamespaces="false"* *Default: false (disabled)*

defaultStatementTimeout	(iBATIS 버전 2.2.0 이나 그 이후 버전) 이 셋팅은 모든 구문에 대한 JDBC 쿼리 타임아웃처럼 적용될 정수값이다. 이 값은 매핑 구문의 "statement" 속성으로 무시 될 수 있다. 이 값이 명시되지 않으면, 매핑 구문의 "statement" 속성이 설정되지 않는다면 쿼리 타임아웃이 셋팅되지 않을것이다. 드라이버가 구문이 종료되도록 기다리는 초단위의 값으로 설정한다. 하지만 모든 드라이버가 이 셋팅을 지원하지는 않으니 반드시 확인하고 사용해야 한다.
classInfoCacheEnabled	이 셋팅을 사용하도록 설정하면, iBATIS는 클래스의 캐시를 관리할것이다. 많은 클래스가 재사용된다면 재빨리 시작하도록 해줄것이다. 예제*: classInfoCacheEnabled="true"* *Default: true (enabled)*
statementCachingEnabled	(iBATIS 버전 2.3.0 이나 그 이후 버전) 이 셋팅을 사용하도록 설정하면, iBATIS는 prepared 구문의 지역적인 캐시를 관리할것이다. 이 설정은 명확한 성능향상을 보여줄 것이다. 예제*: statementCachingEnabled="true"* *Default: true (enabled)*

<resultObjectFactory> 요소

중요**:** 이 기능은 iBATIS 2.2.0 이나 그 이후 버전에서 사용가능하다.

resultObjectFactory 요소는 SQL구문의 실행으로 결과 객체를 생성하기 위한 factory클래스를 명시한다. 이 요소의 사용하지 않을수도 있다. 이 요소를 명시하지 않는다면, iBATIS는 결과 객체를 생성(class.newInstance())하기 위한 내부 기법을 사용할것이다.

iBATIS는 다음의 경우 결과 객체를 생성한다.

1. ResultSet에서 반환되는 레코드를 매핑할때 (가장 공통적인 경우)
2. resultMap으로 결과 요소의 내포된 select구문을 사용할때, 내포된 select구문이 parameterClass를 선언한다면, iBATIS는 내포된 select를 실행하기 전에 클래스의 인스턴스를 생성하고 값을 채울것이다.
3. 저장 프로시저를 실행할때, iBATIS는 OUTPUT파라미터를 위한 객체를 생성할것이다.
4. 내포된 결과맵을 처리할때, 내포된 결과맵이 N+1 쿼리를 피하도록 groupBy지원으로 조합되어 사용된다면, 객체는 대개 Collection, List 또는 Set 타입의 구현체이 될것이다. 결과 객체 factory를 통해 사용자 정의 구현체를 제공할수도 있다. 내포된 결과맵으로 1:1조인을 사용하는 것으로, iBATIS는 이 factory를 통해 명시한 도메인 객체의 인스턴스를 생성할것이다.

factory를 구현하도록 결정했다면, factory클래스는 **com.ibatis.sqlmap.engine.mapping.result.ResultObjectFactory** 인터페이스를 반드시 구현해야만 하고 public 디폴트 생성자를 반드시 가져야 한다. ResultObjectFactory 인터페이스는 두개의 메소드를 가진다. 하나는 객체를 생성하고 다른 하나는 설정에서 명시된 프라퍼티 값을 받는다.

예를 들어, 다음처럼 *resultObjectFactory* 설정 요소를 명시하도록 해보자.

```
<resultObjectFactory type="com.mydomain.MyResultObjectFactory" >
  <property name="someProperty" value="someValue"/>
</resultObjectFactory>
```

그리고 나서 다음처럼 결과 객체 factory클래스를 코딩할것이다.

```
package com.mydomain;

import com.ibatis.sqlmap.engine.mapping.result.ResultObjectFactory;

public class MyResultObjectFactory implements ResultObjectFactory {
```

```
  public MyResultObjectFactory() {
    super();
  }

  public Object createInstance(String statementId, Class clazz)
      throws InstantiationException, IllegalAccessException {

    // create and return instances of clazz here...

  }

  public void setProperty(String name, String value) {
    // save property values here...
  }
}
```

iBATIS는 설정에서 명시된 각각의 프라퍼티를 위해 매번 setProperty메소드를 호출할것이다. 모든 프라퍼티들은 *createInstance* 메소드가 처리되도록 호출하기 전에 셋팅될것이다.
iBATIS는 앞서 언급한 경우처럼 객체가 생성될 필요가 있을때마다 *createInstance* 메소드를 호출할것이다. *CreateInstance* 메소드에서 null이 반환된다면, *iBATIS*는 일반적인 방법(class.newInstance())을 통해 객체를 생성하도록 시도할것이다. java.util.Collection 이나 java.util.List를 생성하기 위한 요청에서 null을 반환한다면, iBATIS는 java.util.ArrayList를 생성할것이다. java.util.Set을 생성하기 위한 요청에서 null을 반환한다면, iBATIS는 java.util.HashSet을 생성할것이다. iBATIS는 객체 생성을 요청하는 컨텍스트를 알도록 현재의 구문을 전달한다.

<typeAlias> 요소

typeAlias 요소는 긴 전체 경로를 포함한 클래스명을 참조하기 위한 짧은 이름을 명시하도록 한다. 예를 들면

```
<typeAlias alias="shortname" type="com.long.class.path.Class"/>
```

SQL Maps설정 파일에서 사용되는 미리 정의된 몇몇 alias가 있다. 그것들은

Transaction Manager Aliases	
JDBC JTA EXTERNAL	com.ibatis.sqlmap.engine.transaction.jdbc.JdbcTransactionConfig com.ibatis.sqlmap.engine.transaction.jta.JtaTransactionConfig com.ibatis.sqlmap.engine.transaction.external.ExternalTransactionConfig
Data Source Factory Aliases	
SIMPLE DBCP JNDI	com.ibatis.sqlmap.engine.datasource.SimpleDataSourceFactory com.ibatis.sqlmap.engine.datasource.DbcpDataSourceFactory com.ibatis.sqlmap.engine.datasource.JndiDataSourceFactory

<transactionManager> 요소

1.0 변환노트**:** SQL Maps 1.0은 다중의 데이터소스 설정을 허락했다. 이것은 다루기 어렵고 몇가지 나쁜 예제를 소개했다. 그러므로 2.0에서는 오직 하나의 데이터소스만을 허락한다. 다중의 배치/설정을 위해서는 시스템에 의해 다르게 설정되거나 SQL Maps를 빌드할 때 파라미터처럼 전달되는 다중속성파일이 추천된다.

<transactionManager> 요소는 당신이 SQL Maps를 위한 트랜잭션 관리를 설정하도록 한다. *type* 속성값은 사용하기 위한 트랜잭션 관리자를 표시한다. 그 값은 클래스명이거나 타입 alias일수 있다. 3개의 트랜잭션 관리자는 JDBC, JTA 그리고 EXTERNAL 중에 하나로 표시할수 있다.

JDBC – 커넥션 commit()과 rollback()메소드를 통해 트랜잭션를 제어하기 위한 JDBC를 사용하게 된다.

JTA – 이 트랜잭션관리자는 SQL Maps가 다른 데이터베이스나 트랜잭션 자원을 포함하는 더욱더 넓은 범위의 트랜잭션을 포함하도록 하는 JTA전역트랜잭션를 사용한다. 이 설정은 JNDI자원으로부터 사용자 트랜잭션을 위치시키기 위한 UserTransaction 속성값을 요구한다. JNDI데이터소스예제는 다음의 설정예제에서 보라.

EXTERNAL – 이것은 당신 자신이 트랜잭션을 관리하도록 한다. 당신은 여전히 데이터소스를 설정할수 있지만 프레임워크 생명주기의 부분처럼 트랜잭션이 커밋되거나 롤백되지 않는다. 이것은 당신 애플리케이션의 부분이

외부적으로 SQL Maps 트랜잭션을 관리해야 한다는것이다. 이 셋팅은 비-트랜잭션(예를 들면 읽기전용) 데이터베이스에 유용하다.

<transactionManager> 요소는 true나 false가 될수 있는 *commitRequired*요소를 선택할수 있다. 대개 iBATIS는 insert, update 또는 delete작업이 실행되지 않고서는 트랜잭션을 커밋하지 않을것이다. 하지만 명시적으로 commitTransaction() 메소드를 호출하면 커밋이 된다. 이런 명시적인 호출은 종종 문제를 야기한다. insert, update, delete작업이 실행되지 않아도 iBATIS가 모든 작업에 대해 커밋하길 바란다면, *commitRequired* 속성을 true값으로 셋팅하라. 다음과 같은 경우에 이 속성값이 유용하다.

1. 반환되는 레코드 수 만큼 데이터를 수정하는 저장 프로시저를 호출할 경우. 이런 경우 queryForList() 메소드를 사용하여 프로시저를 호출할것이다. 그래서 iBATIS는 대개 커밋을 하지 않을것이다. 그렇기 때문에 실제 데이터를 수정하는 작업이 롤백이 될것이다.
2. 웹스피어 환경에서 커넥션 풀링과 JNDI <dataSource> 그리고 JDBC와 JTA트랜잭션 관리자를 사용할 경우. 웹스피어는 풀링된 커넥션의 모든 트랜잭션이 커밋되거나 커넥션이 풀에 반환되지 않을것이다.

commitRequired 속성이 EXTERNAL 트랜잭션 관리자를 사용할때에는 영향을 끼치지 않는다는 것을 주의하라.
몇몇 트랜잭션 관리자는 추가적인 설정 프라퍼티를 요구한다. 다음의 표는 다양한 트랜잭션 관리자를 위해 사용가능한 추가적인 프라퍼티들을 보여준다.

트랜잭션 관리자	프라퍼티	설명
EXTERNAL	DefaultAutoCommit	"true"로 셋팅하면, 각각의 데이터소스가 제공하는 값이 아니라면 setAutoCommit(true)메소드가 각각의 트랜잭션에서 호출될것이다. "false"로 셋팅하거나 명시하지 않는다면, 각각의 데이터소스가 제공하는 값이 아니라면 setAutoCommit(false) 메소드가 각각의 트랜잭션에서 호출될것이다. 이 값은 "SetAutoCommitAllowed" 프라퍼티에 의해 무시될수 있다.
	SetAutoCommitAllowed	"true"로 셋팅하거나 명시하지 않는다면, "DefaultAutoCommit" 프라퍼티가 발생하는 것으로 명시된다. "false"로 셋팅한다면, iBATIS는 어떤 경우에 setAutoCommit를 호출하지 않을것이다. 이 값은 setAutoCommit 메소드가 특정 환경에서 호출이 되지 않는 웹스피어와 같은 환경에서 유용하다.
JTA	UserTransaction	이 프라퍼티는 필수값이다. 사용자 트랜잭션의 값이다. 대개의 경우 "java:comp/UserTransaction"로 셋팅된다.

<dataSource> 요소

트랜잭션관리자 설정의 포함된 부분은 dataSource 요소이고 SQL Maps를 사용하기 위한 데이터소스를 설정하기 위한 속성값의 집합이다. 여기엔 프레임워크에서 제공되는 3가지 데이터소스타입이 있지만 당신은 당신만의 데이터소스를 사용할수도 있다. 포함된 DataSourceFactory구현은 다음에 상세하게 논의가 될것이고 각각을 위해 제공되는 설정은 아래 예제를 보라.

SimpleDataSourceFactory

SimpleDataSource 는 데이터소스를 제공하는 컨테이너가 없는 경우에 커넥션을 제공하기 위해 기본적으로 풀링 데이터소스 구현을 제공한다. 이것은 iBATIS SimpleDataSource 커넥션풀링을 기초로 한다.

```
<transactionManager type="JDBC">
  <dataSource  type="SIMPLE">
      <property name="JDBC.Driver" value="org.postgresql.Driver"/>
      <property name="JDBC.ConnectionURL"
```

```
              value="jdbc:postgresql://server:5432/dbname"/>
      <property name="JDBC.Username" value="user"/>
      <property name="JDBC.Password" value="password"/>
      <!-- OPTIONAL PROPERTIES BELOW -->
      <property name="JDBC.DefaultAutoCommit" value="false"/>
      <property name="Pool.MaximumActiveConnections" value="10"/>
      <property name="Pool.MaximumIdleConnections" value="5"/>
      <property name="Pool.MaximumCheckoutTime" value="120000"/>
      <property name="Pool.TimeToWait" value="10000"/>
      <property name="Pool.PingQuery" value="select * from dual"/>
      <property name="Pool.PingEnabled" value="false"/>
      <property name="Pool.PingConnectionsOlderThan" value="0"/>
      <property name="Pool.PingConnectionsNotUsedFor" value="0"/>
      <property name="Driver.DriverSpecificProperty" value="SomeValue"/>
  </dataSource>
</transactionManager>
```

"Driver." 이라는 접두사를 가지는 프라퍼티는 JDBC 드라이버를 참조하는 프라퍼티처럼 추가될것이다.

DbcpDataSourceFactory

이 구현체은 DataSource API를 통해 커넥션 풀링서비스를 제공하기 위해 Jakarta DBCP (Database Connection Pool)을 사용한다. 이 DataSource는 애플리케이션/웹 컨테이너가 DataSource구현체을 제공하지 못하거나 당신이 standalone애플리케이션을 구동할 때 이상적이다. DbcpDataSourceFactory 를 위해 명시해야 하는 설정 파라미터의 예제는 다음과 같다.

```
<transactionManager type="JDBC">
  <dataSource type="DBCP">
    <property name="driverClassName" value="${driver}"/>
    <property name="url" value="${url}"/>
    <property name="username" value="${username}"/>
    <property name="password" value="${password}"/>
    <!-- OPTIONAL PROPERTIES BELOW -->
    <property name="maxActive" value="10"/>
    <property name="maxIdle" value="5"/>
    <property name="maxWait" value="60000"/>
    <!-- Use of the validation query can be problematic.
         If you have difficulty, try without it.  -->
    <property name="validationQuery" value="select * from ACCOUNT"/>
    <property name="logAbandoned" value="false"/>
    <property name="removeAbandoned" value="false"/>
    <property name="removeAbandonedTimeout" value="50000"/>
    <property name="Driver.DriverSpecificProperty" value="SomeValue"/>
  </datasource>
 </transactionManager>
```

다음의 URL에서 사용가능한 모든 프라퍼티들을 볼수 있다.

http://jakarta.apache.org/commons/dbcp/configuration.html

"Driver." 이라는 접두사를 가지는 프라퍼티는 위에서 보여주는 것처럼 JDBC 드라이버를 참조하는 프라퍼티처럼 추가될것이다. iBATIS는 아래에서 보여주는 것처럼 다소 덜 유연한 레거시 설정 옵션또한 지원한다. 어쨌든, 우리는 위에서 보여주는 설정 옵션을 사용하도록 추천한다.

```
<transactionManager type="JDBC">  <!-- Legacy DBCP Configuration -->
  <dataSource type="DBCP">
    <property name="JDBC.Driver" value="${driver}"/>
    <property name="JDBC.ConnectionURL" value="${url}"/>
    <property name="JDBC.Username" value="${username}"/>
    <property name="JDBC.Password" value="${password}"/>
    <!-- OPTIONAL PROPERTIES BELOW -->
```

```
    <property name="Pool.MaximumActiveConnections" value="10"/>
    <property name="Pool.MaximumIdleConnections" value="5"/>
    <property name="Pool.MaximumWait" value="60000"/>
    <!-- Use of the validation query can be problematic.
         If you have difficulty, try without it.  -->
    <property name="Pool.ValidationQuery" value="select * from ACCOUNT"/>
    <property name="Driver.DriverSpecificProperty" value="SomeValue"/>
  </datasource>
</transactionManager>
```

앞에서 보여준 프라퍼티는 레거시 설정 옵션을 사용할때 iBATIS가 인식하는 프라퍼티일뿐이다. "Driver." 이라는 접두사를 가지는 프라퍼티는 위에서 보여주는 것처럼 JDBC 드라이버를 참조하는 프라퍼티처럼 추가될것이다.

JndiDataSourceFactory

이 구현체은 애플리케이션 컨테이너내 JNDI컨텍스트로부터 DataSource구현체을 가져와야 할것이다. 이것은 전형적으로 애플리케이션서버를 사용중이고 컨테이너관리 커넥션 풀 그리고 제공되는 DataSource구현체이 있을 때 사용한다. JDBC DataSource구현체에 접근하기 위한 표준적인 방법은 JNDI컨텍스트를 통하는것이다. JndiDataSourceFactory 는 JNDI를 통해 DataSource에 접근하는 기능을 제공한다. 데이터소스내에 명시되어야 하는 설정 파라미터는 다음과 같다.

```
<transactionManager type="JDBC" >
    <dataSource type="JNDI">
      <property name="DataSource" value="java:comp/env/jdbc/jpetstore"/>
    </dataSource>
 </transactionManager>
```

위 설정은 일반적인 JDBC트랜잭션 관리지만 컨테이너가 자원을 관리한다. 당신은 다음처럼 전역(global)트랜잭션을 설정하길 원할수도 있다.

```
<transactionManager type="JTA" >
    <property name="UserTransaction" value="java:/comp/UserTransaction"/>
    <dataSource type="JNDI">
      <property name="DataSource" value="java:comp/env/jdbc/jpetstore"/>
    </dataSource>
</transactionManager>
```

UserTransaction 인스턴스가 발견될수 있는 JNDI위치를 가지키는 *UserTransaction* 값에 주의하라. 좀더 넓은 범위의 트랜잭션을 가지는 SQL Maps가 다른 데이터베이스와 트랜잭션 자원을 포함하기 위해서는 JTA트랜잭션 관리가 요구된다.

JNDI 컨텍스트 프라퍼티는 "*context.*" 라는 접두사를 가진 추가적인 프라퍼티를 명시하기 전에 추가될수 있다. 예를 들어 다음과 같다.

<property name="context.java.naming.provider.url" value= "ldap://somehost:389"/>

<sqlMap> 요소

sqlMap 요소는 명시적으로 SQL Map이나 다른 SQL Map설정파일을 포함할 때 사용한다. SqlMapClient인스턴스에 의해 사용되는 각각의 SQL Map XML파일은 반드시 선언되어야 한다. SQL Map XML파일은 클래스패스나 URL로부터 스트림(stream)자원처럼 로드 될것이다. 당신은 SQL Maps를 명시해야 한다. 다음은 그에 대한 예이다.

```
<!--CLASSPATH RESOURCES -->
<sqlMap resource="com/ibatis/examples/sql/Customer.xml" />
<sqlMap resource="com/ibatis/examples/sql/Account.xml" />
<sqlMap resource="com/ibatis/examples/sql/Product.xml" />


<!--URL RESOURCES -->
<sqlMap url="file:///c:/config/Customer.xml " />
```

```
<sqlMap url="file:///c:/config/Account.xml " />
<sqlMap url="file:///c:/config/Product.xml" />
```

다음의 다양한 섹견은 *SQL Map XML*파일들의 구조에 대해서 서술한다.

SQL Map XML 파일

(http://ibatis.apache.org/dtd/sql-map-config-2.dtd)

위 예제에서 우리는 SQL Maps의 가장 간단한 형태를 보았다. SQL Map문서 구조내에 사용가능한 다른 옵션이 있다. 좀더 많은 기능을 가지는 매핑 구문의 예제이다.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMap
   PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
   "http://ibatis.apache.org/dtd/sql-map-2.dtd">
<sqlMap namespace="Product">

          <cacheModel  id="productCache"  type="LRU">
                 <flushInterval hours="24"/>
                 <property name="size" value="1000" />
          </cacheModel>

        <typeAlias alias="product" type="com.ibatis.example.Product" />

        <parameterMap  id="productParam" class="product">
                 <parameter property="id"/>
        </parameterMap>

        <resultMap   id="productResult" class="product">
                 <result property="id" column="PRD_ID"/>
                 <result property="description" column="PRD_DESCRIPTION"/>
          </resultMap>

        <select  id="getProduct"  parameterMap="productParam"
                   resultMap="productResult"  cacheModel="product-cache">
            select * from PRODUCT where PRD_ID = ?
          </select>

</sqlMap>
```

너무 많은가.**?** 비록 프레임워크가 당신을 위해 많은 것을 하더라도 간단한 select 구문을 위해 너무 많은 추가적인 작업을 하는것처럼 보인다. 걱정하지 마라 다음은 위의 것의 축소버전이다.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE sqlMap
   PUBLIC "-//ibatis.apache.org//DTD SQL Map 2.0//EN"
   "http://ibatis.apache.org/dtd/sql-map-2.dtd">
<sqlMap  namespace="Product">

     <select  id="getProduct"  parameterClass=" com.ibatis.example.Product"
                                  resultClass="com.ibatis.example.Product">
        select
           PRD_ID as id,
           PRD_DESCRIPTION as description
        from PRODUCT
        where PRD_ID = #id#
     </select>
```

```
</sqlMap>
```

지금 SQL Map을 행위적인 측면에서 보면 이 구문은 정확하게 같지는 않다. 즉 몇가지 다른점을 가진다. 먼저 후자의 구문은 캐쉬를 명시하지 않아서 매번의 요청시 데이터베이스에 직접 요청한다. 두번째 후자의 구문은 약간의 부하를 야기할수 있는 프레임워크의 자동매핑기능을 사용한다. 어쨌든 두가지 구문 모두 자바코드로부터 정확하게 같은 방법으로 작동하지 않을것이다 그리고 당신은 첫번째 좀더 간단한 솔루션으로 시작할것이고 나중에는 필요하면 좀더 향상된 매핑으로 옮겨갈 것이다. 가장 간단한 솔루션이 많은 경우에 가장 좋은 연습이다.

하나의 SQL Map XML 파일은 많은 캐쉬 모델, 파라미터 매핑, 결과 매핑 그리고 구문을 포함할수 없다. 당신의 애플리케이션을 위해 구문과 맵을 신중하게 구성하라.

매핑 구문

SQL Maps 개념은 매핑 구문에 집중한다. 매핑 구문는 어떠한 SQL문을 사용할수도 있고 파라미터 maps(input)과 결과 맵s(output)를 가질수 있다. 만약 간단한 경우라면 매핑 구문는 파라미터와 결과를 위한 클래스로 직접 설정할수 있다. 매핑 구문는 메모리내에 생산된 결과를 캐싱하기 위해 캐쉬 모델을 사용하도록 설정할수도 있다.

```
<statement      id="statementName"
                [parameterClass="some.class.Name"]
                [resultClass="some.class.Name"]
                [parameterMap="nameOfParameterMap"]
                [resultMap="nameOfResultMap"]
                [cacheModel="nameOfCache"]
                [timeout="5"]>
        select * from PRODUCT where PRD_ID = [?|#propertyName#]
        order by [$simpleDynamic$]
</statement>
```

위 구문에서 **[**괄호**]** 부분은 옵션이고 몇몇의 경우에만 혼합할 필요가 있다.

```
<insert id="insertTestProduct" >
        insert into PRODUCT (PRD_ID, PRD_DESCRIPTION) values (1, "Shih Tzu")
</insert>
```

위 예제는 명백하게 발생할꺼 같지는 않다. 어쨌든 이것은 당신이 임의의 SQL 문을 실행하기 위해 SQL Map프레임워크를 사용한다면 유용할수 있다. 어쨌든 이것은 파라미터 맵과 결과 맵을 사용하는 자바빈즈 매핑기능을 공통적으로 사용할것이다. 다음의 다양한 섹션은 구조와 속성, 그들이 어떻게 매핑 구문에 영향을 끼치는지 서술한다.

구문 타입

<statement> 요소는 어떤 타입의 SQL 문을 사용할수 있는 일반적인 "catch all" 구문이다. 일반적으로 이것은 좀더 다양한 특성의 statement요소 중 하나를 사용하기 위한 좋은 생각이다. 좀더 다양한 특성의 요소는 좀더 직관적인 XML DTD를 제공하고 때때로 일반적인 <statement>요소가 제공하지 않는 추가적인 기능을 제공한다. 다음의 테이블은 statement요소와 그들이 지원하는 속성과 기능을 목록화 한다.

Statement 요소	속성	하위 요소	메소드
<statement>	id parameterClass resultClass parameterMap resultMap cacheModel resultSetType fetchSize	모든 동적 요소	insert update delete 모든 쿼리 메소드

	xmlResultName remapResults timeout		
<insert>	id parameterClass parameterMap timeout	모든 동적 요소 <selectKey>	insert update delete
<update>	id parameterClass parameterMap timeout	모든 동적 요소	insert update delete
<delete>	id parameterClass parameterMap timeout	모든 동적 요소	insert update delete
<select>	id parameterClass resultClass parameterMap resultMap cacheModel resultSetType fetchSize xmlResultName remapResults timeout	모든 동적 요소	모든 쿼리 메소드
<procedure>	id parameterClass resultClass	모든 동적 요소	insert update delete

SQL

SQL 은 맵의 가장 중요한 부분을 차지한다. 이것은 당신의 데이터베이스와 JDBC드라이버에 적합한 어떤 SQL이 될수 있다. 당신은 가능한 어떤 기능을 사용할수 있고 당신의 드라이버가 지원하는 한 다중 구문에 전달할수도 있다. 당신이 하나의 문서에서 SQL과 XML을 혼합하기 때문에 특수문자의 충돌이 잠재적으로 존재한다. 대부분의 공통적인 것은 greater-than 과 less-than 문자들이다.(<>). 이것들은 SQL문에서 공통적으로 요구되고 XML에서는 예약어이다. 당신의 SQL문에 들어갈 필요가 있는 특수 XML문자를 처리하기 위한 간단한 해결법이 있다. 표준적인 XML CDATA 섹션을 사용함으로써 특수문자의 어떤것도 파싱되지 않고 문제는 해결된다. 예를 들면

```
<select id="getPersonsByAge" parameterClass=”int” resultClass="examples.domain.Person">
        SELECT *
        FROM PERSON
        WHERE AGE <![CDATA[ > ]]> #value#
</select>
```

SQL 재사용하기

SqlMaps 를 사용할때, 종종 SQL문의 일부가 중복되는 것을 보곤한다. 예를들어, FROM절이나 제약조건들이 그런 경우이다. iBATIS는 중복된 SQL문의 일부를 재사용하도록 하는 강력한 요소를 제공한다. 간단하게 몇가지 항목을 가져와서 그 항목의 개수를 센다고 가정해보자. 대개 다음처럼 작성할것이다.

```
<select id="selectItemCount" resultClass="int">
        SELECT COUNT(*) AS total
        FROM items
        WHERE parentid = 6
```

```
</select>

<select id="selectItems" resultClass="Item">
        SELECT id, name
        FROM items
        WHERE parentid = 6
</select>
```

중복을 제거하기 위해, <sql> 과 <include> 요소를 사용할것이다. <sql> 요소는 재사용하기 위한 일부를 저장한다. 구문내 그 일부를 포함하도록 <include> 요소를 사용한다. 예를 들어:

```
<sql id="selectItem_fragment">
        FROM items
        WHERE parentid = 6
</sql>

<select id="selectItemCount" resultClass="int">
        SELECT COUNT(*) AS total
        <include refid="selectItem_fragment"/>
</select>

<select id="selectItems" resultClass="Item">
        SELECT id, name
        <include refid="selectItem_fragment"/>
</select>
```

<include> 요소는 다른 맵에 있는 SQL을 참조할수 있도록 명명공간을 사용한다. (어쨌든 iBATIS가 SqlMaps을 로드하는 방식때문에 맵의 정보는 구문의 일부를 가져오기 전에 로그될것이다.)
SQL문의 일부분은 쿼리 실행시 포함되고 처리된다.

```
<sql id="selectItem_fragment">
        FROM items
        WHERE parentid = #value#
</sql>

<select id="selectItemCount" parameterClass="int" resultClass="int">
        SELECT COUNT(*) AS total
        <include refid="selectItem_fragment"/>
</select>

<select id="selectItems" parameterClass="int" resultClass="Item">
        SELECT id, name
        <include refid="selectItem_fragment"/>
</select>
```

자동 생성 키

많은 관계형 데이터베이스 시스템은 기본키(primay key)필드의 자동생성을 지원한다. 이 RDBMS의 기능은 종종 특정업체에 종속된다. SQL Map은 <insert>요소의 <selectKey>를 통해 자동생성키를 지원한다. 선생성키(pre-generated - 이를 테면 오라클)과 후생성키(post-generated - 이를 테면 MS-SQL 서버) 모두 지원한다. 여기에 그 예제가 있다.

```
<!—Oracle SEQUENCE Example -->
<insert id="insertProduct-ORACLE" parameterClass="com.domain.Product">
    <selectKey resultClass="int" >
        SELECT STOCKIDSEQUENCE.NEXTVAL AS ID FROM DUAL
    </selectKey>
    insert into PRODUCT (PRD_ID,PRD_DESCRIPTION)
    values  (#id#,#description#)
</insert>
```

```
<!— Microsoft SQL Server IDENTITY Column Example -->
<insert id="insertProduct-MS-SQL" parameterClass="com.domain.Product">
    insert into PRODUCT (PRD_DESCRIPTION)
    values  (#description#)
   <selectKey resultClass="int" >
        SELECT @@IDENTITY AS ID
   </selectKey>
</insert>
```

selectKey구문이 insert SQL앞에 있다면 insert구문보다 먼저 실행된다. 하지만 insert구문 뒤에 있다면 뒤에 실행된다. 앞의 예제에서 오라클 예제는 selectKey가 insert구문앞에서 실행되는 것을 보여준다(시퀀스에 적절한). SQL서버의 예제는 selectKey구문이 insert구문 뒤에서 실행된다는 것을 보여준다(identity 칼럼에 적절한).
IBATIS 버전 2.2.0이나 그 이후 버전에서, 원한다면 구문의 실행 순서를 명시적으로 정할수 있다. *SelectKey* 요소는 실행 순서를 명시적으로 셋팅하기 위해 사용될수 있는 type속성을 지원한다. *Type* 속성의 값은 “pre”나 “post” 둘중 하나가 될 수 있다. 이 값은 구문이 insert구문의 앞이나 뒤에서 실행된다는 것을 의미한다. *Type* 속성을 명시한다면, 명시한 값은 selectKey요소의 위치에 따를것이다. 예를 들어, 다음 구문에서 selectKey구문은 비록 요소가 insert구문뒤에 위치하더라도 insert구문 이전에 실행될것이다.

```
<insert id="insertProduct-ORACLE-type-specified" parameterClass="com.domain.Product">
    insert into PRODUCT (PRD_ID,PRD_DESCRIPTION)
    values  (#id#,#description#)
   <selectKey resultClass="int" type="pre" >
        SELECT STOCKIDSEQUENCE.NEXTVAL AS ID FROM DUAL
   </selectKey>
</insert>
```

<selectKey> 속성:

<selectKey> 속성	설명
resultClass	<selectKey> 구문의 실행 결과로 생성되는 자바 클래스(대개는 Integer나 Long)
keyProperty	<selectKey> 구문의 실행결과처럼 파라미터 객체에 셋팅될 프라퍼티. 명시하지 않으면, iBATIS는 데이터베이스에서 반환되는 칼럼명에 기초하여 프라퍼티를 찾도록 시도할것이다. 프라퍼티를 찾을수 없다면, 프라퍼티가 셋팅되지는 않을것이지만 iBATIS는 여전히 <insert>구문의 결과로 생성된 키를 반환할것이다.
type	“pre” 나 “post” 값을 가질수 있다. 명시한다면, 관련된 insert구문이 앞(pre)이나 뒤(post)에 select key구문이 실행된다는 것을 나타낸다. 명시하지 않는다면 순서는 insert구문내 요소의 위치에 따라 좌우될것이다. 즉 SQL앞에 위치한다면 selectKey는 구문이 실행되기 전에 먼저 실행될것이다. 이 속성은 iBATIS 2.2.0 이나 그 이후의 버전에서만 사용가능하다.

저장 프로시저

저장 프로시저는 <procedure> statement요소를 통해 지원된다. 저장 프로시저를 출력물 파라미터와 함께 어떻게 사용하는지 다음 예제에서 보여준다.

```
<parameterMap id="swapParameters" class="map" >
  <parameter property="email1" jdbcType="VARCHAR" javaType="java.lang.String"
  mode="INOUT"/>
  <parameter property="email2" jdbcType="VARCHAR" javaType="java.lang.String"
mode="INOUT"/>

</parameterMap>


<procedure id="swapEmailAddresses" parameterMap="swapParameters" >
  {call swap_email_address (?, ?)}

</procedure>
```

위처럼 프로시저를 호출하는 것은 파라미터 객체(map)내에서 두개의 칼럼사이에 두개의 이메일주소를 교체하는것이다. 파라미터 객체는 파라미터 매핑의 mode속성값이 "INOUT"또는 "OUT"일 경우에만 변경된다. 다른 경우라면 변경되지 않고 남는다. 명백한 불변의 파라미터 객체(이를 테면 String)는 변경할수 없다.

주의*!* 언제나 표준적인 JDBC저장 프로시저를 사용하도록 하라. 좀더 다양한 정보를 보기 위해서는 JDBC CallableStatement문서를 보라.

parameterClass

parameterClass 속성값은 자바클래스의 전체경로를 포함(예를 들면 패키지를 포함한)한 이름이다. parameterClass 속성은 옵션이지만 사용이 굉장히 추천되는 것이다. 이것은 프레임워크 성능을 향상시키는 만큼 statement에 전달하는 파라미터를 제한하는데 사용된다. 만약 당신이 parameterMap을 사용한다면 parameterClass속성을 사용할 필요가 없다. 예를 들면 당신이 파라미터로 전달하기 위한 "examples.domain.Product" 타입의 객체를 허락하길 원한다면 당신은 다음처럼 할수 있을것이다.

```
<insert id="statementName"   parameterClass=" examples.domain.Product">
        insert into PRODUCT values (#id#, #description#, #price#)
</insert>
```

중요**:** 비록 이전버전과의 호환성을 위한 옵션이지만 이것은 언제나 파라미터 클래스를 제공하는 것은 매우 추천되는 사항이다(물론 요구되는 파라미터가 없더라도). 프레임워크가 먼저 타입을 안다면 스스로 최적화능력을 가지기 때문에 당신은 클래스를 제공함으로써 좀더 나은 성능을 달성할수 있다.

명시된 parameterClass 없이 선호하는 속성(get/set메소드)을 가지는 자바빈즈는 파라미터를 받을것이고 어느 위치에서 매우 유용하다.

parameterMap

parameterMap 속성값은 명시된(밑의 경우처럼) parameterMap요소의 이름이다. parameterMap속성은 parameterClass 속성과 인라인 파라미터의 이익이 되도록 사용된다. XML의 깔끔함과 일관성이 당신의 걱정이거나 당신이 좀더 상세한 parameterMap(이를 테면 저장프로시저)이 필요하다면 이것은 좋은 접근법이다.

주의*!* 동적으로 매핑 구문는 단지 인라인 파라미터만 지원하고 파라미터 map과는 작동하지 않는다.

parameterMap의 생각은 JDBC PreparedStatement의 값 토큰과 매치되는 정렬된 파라미터 목록을 명시한다.

예를들면:

```
<parameterMap id="insert-product-param" class="com.domain.Product">
        <parameter property="id"/>
        <parameter property="description"/>
</parameterMap>

<insert id="insertProduct"  parameterMap="insert-product-param">
        insert into PRODUCT (PRD_ID, PRD_DESCRIPTION) values (?,?)
</insert>
```

위의 예제에서, 파라미터 map은 SQL문에서 값토큰("?")에 매치되고 정렬되는 두개의 파라미터를 서술한다. 그래서 첫번째 "?"는 "id" 속성값에 대체되고 두번째는 "description" 속성값에 대체된다. 파라미터 map과 그들의 옵션은 이 문서 나중에 좀더 다양하게 서술될것이다.

인라인 파라미터의 빠른 언급

이 문서에 나중에 제공되는 좀더 상세화된 설명에도 불구하고 인라인 파라미터에 대한 빠른 언급을 한다. 인라인 파라미터는 매핑 구문내부에서 사용될수 있다. 예를 들면 :

```
<insert id="insertProduct" >
```

```
        insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
        values (#id#, #description#)
</insert>
```

위 예제에서 인라인 파라미터는 *#id#* 와 *#description#* 이다. 각각은 구문 파라미터를 대체하는 자바빈즈 속성을 표현한다. Product클래스는 포함된 프라퍼티 토큰이 위치해 있는 구문내에 위치하는 값을 위해 읽게 되는 *id* 와 *description* 프라퍼티을 가진다. *id=5* 와 *description="dog"* 를 가지는 Product를 넘겨받은 구문은 다음처럼 수행된다.

```
    insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
    values (5, 'dog');
```

resultClass

resultClass속성값은 자바클래스의 전체경로를 포함(예를 들면 패키지를 포함한)한 이름이다. resultClass속성은 우리에게 ResultSetMetaData에 기반한 JDBC ResultSet에 자동매핑되는 클래스를 명시하도록 한다. 자바빈즈의 프라퍼티와 ResultSet의 칼럼이 매치될때마다 프라퍼티는 칼럼값과 함께 생성된다. 이것은 매우 짧고 달콤하게 매핑 구문를 쿼리한다. 예를 들면 :

```
<select id="getPerson" parameterClass="int"  resultClass="examples.domain.Person">
        SELECT
                PER_ID                  as id,
                PER_FIRST_NAME      as firstName,
                PER_LAST_NAME       as lastName,
                PER_BIRTH_DATE      as birthDate,
                PER_WEIGHT_KG       as weightInKilograms,
                PER_HEIGHT_M        as heightInMeters
        FROM PERSON
        WHERE PER_ID = #value#
</select>
```

위의 예제에서 Person클래스는 *id, firstName, lastName, birthDate, weightInKilograms, heightInMeters*를 포함하는 프라퍼티를 가진다. 칼럼별칭과 함께 대응되는 각각은 SQL select문에 의해 서술된다. 칼럼별칭은 데이터베이스 칼럼이름이 매치되지 않을때만 요구된다. 일반적으로는 요구되지 않는다. 실행되었을때 Person객체는 프라퍼티이름과 칼럼명에 기반해서 초기화되기 위해 매핑되는 결과 세트로부터 초기화되고 결과를 반환한다.

resultClass으로 자동매핑하는데는 몇 가지 제한점이 있다. 출력칼럼의 타입을 명시하는 방법은 없다. 관련된 데이터를 자동적으로 로드하는방법이 없고 ResultSetMetaData에 접근하는데 필요한 접근법내에서 하찮은 성능결과가 있다. 이 제한점 모드는 명시적인 resultMap를 사용함으로써 극복할수 있다. 결과 맵은 이 문서 나중에 좀더 상세하게 다루어질것이다.

resultMap

resultMap프라퍼티는 좀더 공통적으로 사용되고 이해하기 위해 가장 중요한 속성중에 하나이다. 이 resultMap속성값은 명시된 resultMap요소의 이름이다. resultMap속성을 사용하는 것은 당신에게 결과 세트로부터 데이터와 칼럼에 매핑되는 프라퍼티를 어떻게 꺼내는지 제어하도록 한다. resultClass속성을 사용하는 자동매핑접근법과는 달리 resultMap는 당신에게 칼럼타입을 명시하고 null값을 대체 그리고 복합 프라퍼티매핑(다른 자바빈즈, Collections 그리고 원시타입래퍼)을 허락한다.

resultMap 구조의 모든 상세정보는 이 문서 나중에 설명된다. 하지만 다음의 예제는 resultMap가 어떻게 구문에 관련되었는지 보여준다.

```
<resultMap id="get-product-result" class="com.ibatis.example.Product">
        <result property="id" column="PRD_ID"/>
        <result property="description" column="PRD_DESCRIPTION"/>
</resultMap>

<select id="getProduct" resultMap="get-product-result">
        select * from PRODUCT
</select>
```

위 예제에서 SQL쿼리로부터 ResultSet은 resultMap정의를 사용해서 Product인스턴스에 매핑할것이다. resultMap은 "id"프라퍼티가 "PRD_ID"칼럼과 "PRD_DESCRIPTION"칼럼에 의해 생성되는 "description"프라퍼티에 의해 생성될것이다. "select *"를 사용하는 것은 지원된다는 것에 주의하라. ResultSet내 반환칼럼 모두에 매핑할 필요는 없다.

cacheModel

cacheModel속성값은 정의된 cacheModel요소의 이름이다. cacheModel은 쿼리가 매핑 구문를 사용하기 위한 캐쉬를 서술하는데 사용된다. 각각의 쿼리매핑 구문는 다른 cacheModel이나 같은것을 사용할수 있다. cacheModel요소와 그것의 속성에 대한 모든 상세설명은 나중에 언급된다. 다음 예제는 어떻게 구문과 관련되는지 보여준다.

```
<cacheModel id="product-cache" type="LRU">
        <flushInterval hours="24"/>
        <flushOnExecute statement="insertProduct"/>
        <flushOnExecute statement="updateProduct"/>
        <flushOnExecute statement="deleteProduct"/>
        <property name="size" value="1000" />
</cacheModel>

<select id="getProductList"  parameterClass="int"  cacheModel="product-cache">
        select * from PRODUCT where PRD_CAT_ID = #value#
</select>
```

위 예제에서 캐쉬는 WEAK참조타입을 사용하는 products를 위해 정의되고 24시간마다 또는 관련된 update문이 수행될때마다 지워진다(flush) .

xmlResultName

매핑 결과를 XML문서로 직접적으로 만들 때 xmlResultName의 값은 XML문서의 가장 상위 요소의 이름이 될것이다. 예를 들면 :

```
<select id="getPerson" parameterClass="int"  resultClass="xml" xmlResultName="person">
        SELECT
                PER_ID                  as id,
                PER_FIRST_NAME    as firstName,
                PER_LAST_NAME     as lastName,
                PER_BIRTH_DATE    as birthDate,
                PER_WEIGHT_KG     as weightInKilograms,
                PER_HEIGHT_M      as heightInMeters
        FROM PERSON
        WHERE PER_ID = #value#
</select>
```

위 select 구문은 다음 구조의 XML객체를 생성할 것이다.

```
<person>
  <id>1</id>
  <firstName>Clinton</firstName>
  <lastName>Begin</lastName>
  <birthDate>1900-01-01</birthDate>
  <weightInKilograms>89</weightInKilograms>
  <heightInMeters>1.77</heightInMeters>
</person>
```

remapResults

remapResults 속성은 <statement>, <select>, 그리고 <procedure> 에서 사용가능하다. 이것은 선택적인 속성이고 디폴트는 false이다.

remapResults속성은 쿼리가 반환 칼럼의 다양한 세트를 가질때 true셋팅되어야만 한다. 다음 쿼리를 보자.

```
SELECT  $fieldList$
FROM     table
```

이전 예제에서 칼럼의 목록은 테이블이 언제나 같더라도 동적이다.

```
SELECT  *
FROM     $sometable$
```

이전 예제에서 테이블은 다를수 있다. Select절의 * 사용때문에, 결과적인 칼럼이름은 다를수 있다. 동적 요소는 하나의 쿼리 수행에서 다음 수행까지 변경하기 위한 목록을 야기할수있다.

resultset메타데이타를 알고/판단하기 위한 오버헤드가 명백하지 않기 때문에, iBATIS는 마지막 쿼리 수행에 반환된 것만을 기억할것이다. 이것은 위 예제와 비슷한 상황에서 문제를 발생시킨다.

만약 반환 칼럼이 변경된다면, remapResults를 true로 셋팅하라. 그렇지 않다면 메타데이타 검색의 오버헤드를 제거하기 위해 remapResults를 false로 셋팅하라.

resultSetType

SQL구문의 resultSetType을 명시하기 위해, 다음을 사용할수 있다.

- **FORWARD_ONLY**: 커서는 앞쪽으로만 이동한다.
- **SCROLL_INSENSITIVE**: 커서는 스크롤가능하지만 다른것에 의한 변경에는 대개 민감하지 않다.
- **SCROLL_SENSITIVE**: 커서는 스크롤가능하고 다른것에 의한 변경에 대개 민감하다.

resultSetType은 대개 요구되지 않는다. 그리고 서로 다른 JDBC드라이버는 같은 resultSetType셋팅을 사용하더라도 다르게 행동할것이다. (이를테면. Oracle은 **SCROLL_SENSITIVE**를 지원하지 않는다.).

fetchSize

SQL구문의 fetchSize를 셋팅하는 것은 수행될것이다. 이것은 JDBC드라이버에 데이터베이스 서버로의 왕복을 줄이기 위해 prefetching 힌트를 제공한다.

timeout (iBATIS 버전 2.2.0 나 그 이후의 버전)

구문을 위한 JDBC쿼리 타임아웃을 셋팅한다. 여기서 명시된 값은 SQLMapConfig.xml 파일의 "defaultStatementTimeout" 셋팅에 명시된 값을 무시할것이다. 디폴트 타임아웃을 명시하고 특정 구문을 위한 타임아웃을 원하지 않는다면, 타임아웃값을 0으로 셋팅하라. 명시된 값은 드라이버가 종료되도록 구문을 기다리는 초단위 값이다. 이 셋팅을 지원하지 않는 드라이버가 있다는 점에 주의하라.

파라미터 맵과 인라인 파라미터

당신이 위에서 본것처럼 parameterMap는 자바빈즈 프라퍼티를 구문의 프라퍼티에 매핑시키는 작업을 수행한다. 비록 parameterMaps가 외부형태내에 드물게 발생하더라도 그것들이 당신에게 인라인 파라미터를 이해하도록 도와준다는 것을 이해하라.

```
<parameterMap id="parameterMapName" [class="com.domain.Product"]>
    <parameter property ="propertyName" [jdbcType="VARCHAR"] [javaType="string"]
                                        [nullValue="-9999"]
                                        [typeName="{REF or user-defined type}"]
                                        [resultMap=someResultMap]
                                        [mode=IN|OUT|INOUT]
                                        [typeHandler=someTypeHandler]
                                        [numericScale=2]/>
    <parameter ...... />
    <parameter ...... />
</parameterMap>
```

[괄호**]**내의 부분은 옵션이다. parameterMap는 구문이 참조할 때 사용하는 구분자로써 단지 *id*속성만 필요하다. *Class* 속성은 옵션이지만 크게 사용이 추천되는 것이다. 구문의 parameterClass속성과 유사하게 class속성은 프레임워크가 성능을 위해 엔진을 최적화하는 것만큼 들어오는 파라미터를 체크하도록 한다.

<parameter> 요소

parameterMap은 구문의 파라미터에 직접 매핑하는 파라미터매핑의 어떤 숫자를 포함한다. 다음의 일부 섹션은 *property* 요소의 속성을 서술한다.

property

파라미터 맵의 property속성은 매핑 구문에 전달되는 파라미터객체의 자바빈즈 프라퍼티(get메소드)의 이름이다. 그 이름은 구문에 필요한 횟수에 의존하는것보다 좀더 사용될수 있다.

jdbcType

*jdbcType*속성은 이 프라퍼티에 의해 셋팅되는 파라미터의 칼럼타입을 명시적으로 정의하는데 사용된다. 몇몇 JDBC드라이버는 명시적인 드라이버 칼럼타입을 부르는 것 없이 어떤 작동을 위해 칼럼의 타입을 확인할수 없다. 이것의 완벽한 예제는 PreparedStatement.setNull(int parameterIndex, int sqlType) 메소드이다. 이 메소드는 정의하기 위한 타입을 요구한다. 몇몇 드라이버는 간단하게 Types.OTHER 또는 Types.NULL을 보냄으로써 함축되는 타입을 허락한다. 어쨌든 행위는 비일관적이고 몇몇 드라이버는 정의되기 위한 정확한 타입을 필요로한다. 그런 경우를 위해서 SQL Maps API는 parameterMap프라퍼티 요소의 *jdbcType* 속성을 사용하여 정의되기 위한 타입을 허락한다.

이 속성은 칼럼이 null이 가능할 때(nullable)만 요구된다. Type속성을 사용하는 다른 이유는 명시적으로 date타입을 정의하는 것이다. 자바는 단지 하나의 Date값타입(java.util.Date)을 가지는데 반해 대개의 SQL데이터베이스는 많은, 대개 최소 3가지 이상의 타입을 가진다. 당신의 칼럼 타입이 DATE나 DATETIME중에 하나로 명시적으로 정의하길 바랄지도 모르기 때문이다.

jdbcType 속성은 JDBC타입 클래스내 변수와 매치되는 어떤 문자열값에 셋팅될수 있다. 비록 이것은 그것들중에 어떤 것에 셋팅될수 있지만 몇몇 타입은 지원되지 않는다(이를 테면 blobs). 이 문서의 나중 섹션에서 프레임워크에 의해 지원되는 타입에 대해서 서술한다.

주의***!*** 대부분의 드라이버는 단지 null이 가능한 칼럼을 위해 정의되는 타입을 필요로 한다. 그러므로 그런 드라이버를 위해 당신은 null이 가능한 칼럼을 위해 타입을 정의할 필요가 있다.

주의***!*** 오라클 드라이버를 사용할 때 당신은 이것의 타입을 정의하지 않고서는 칼럼에 null값을 넣을 때 *"Invalid column type"* 에러를 보게될것이다.

javaType

javaType 속성은 셋팅되기 위한 파라미터의 자바 프라퍼티를 명시적으로 정의하기 위해 사용된다. 대게 이것은 리플렉션(reflection)을 통해 자바빈즈 프라퍼티로부터 파생된다. 하지만 Map과 XML매핑 같은 특정 매핑은 프레임워크를 위한 타입을 제공하지 않는다. 만약 javaType가 셋팅되지 않고 프레임워크도 어떤타입인지 구별할수 없다면 타입은 객체로 간주될것이다.

typeName

typeName 속성은 REF타입이나 사용자 정의 타입을 명시하기 위해 사용된다.

javadoc에 보면..

typeName 속성은 사용자-정의나 REF출력 파라미터를 위해 사용된다. 예를 들면, 사용자-정의 타입은 STRUCT, DISTINCT, JAVA_OBJECT, 그리고 명명된 배열 타입을 포함한다. 사용자-정의 파라미터를 위해, 파라미터의 전체 경로가 포함된 SQL타입명이 주어진다. 반면에 REF파라미터는 주어진 참조타입의 전체 경로가 포함된 타입명을 요구한다. JDBC 드라이버는 타입코드를 필요로 하지 않으며 타입명 정보는 이것을 무시한다. 이식가능하기 위해, 애플리케이션은 이러한 사용자정의와 REF파라미터를 위한 값을 제공해야만 한다. 비록 이것이 사용자-정의와 REF파라미터가 되더라도, 이 속성은

JDBC타입의 파라미터를 등록하기 위해 사용된다. 만약 파라미터가 사용자-정의나 REF타입을 가지지 않는다면, *typeName*파라미터를 무시된다.

nullValue

nullValue 속성은 어떤 유효한 값(프라퍼티 타입에 기초로 해서)에 셋팅할수 있다. null 속성은 null값 대체를 정의하기 위해 사용된다. 이것이 의미하는 것은 자바빈즈 프라퍼티내에서 검색되는 값인 NULL이 데이터베이스에 쓰여질것이라는것이다(들어오는 null값 대체의 상반된 행위). 이것은 당신에게 null값을 지원하지 않는 타입(이를 테면 int, double, float등등)을 위해 당신의 애플리케이션내에 "magic" null 숫자를 사용하도록 허락한다. 프라퍼티의 그런 타입은 적합한 null값을 포함할 때 NULL은 값 대신에 데이터베이스에 쓰여질것이다.

resultMap

저장 프로시저의 output파라미터의 값으로 java.sql.ResultSet의 인스턴스를 기대할때 resultMap요소를 명시한다. 이 값은 iBATIS로 하여금 객체 매핑을 위한 일반적인 결과 세트를 가능하게 할것이다.

mode

mode 속성은 저장 프로시저 파라미터의 타입을 명시한다. 사용가능한 값은 IN, OUT, INOUT이다.

typeHandler

typeHandler 속성은 디폴트 iBATIS타입대신에 이 프라퍼티를 위해 사용될 사용자 정의 타입 핸들러를 명시하기 위해 사용된다. 명시한다면, 값은 com.ibatis.sqlmap.engine.type.TypeHandler 인터페이스나 com.ibatis.sqlmap.client.extensions.TypeHandlerCallback 인터페이스를 구현하는 패키지 경로를 포함한 클래스명이어야만 한다. 이 값은 전역 타입 핸들러를 무시한다. 이 문서의 뒷부분에 사용자 정의 타입 핸들러에 대해 상세히 다룬다.

numericScale

(numericScale 은 iBATIS 버전 2.2.2나 그 이후 버전에서 사용가능하다.)

numericScale 속성은 NUMERIC 이나 DECIMAL 형태의 저장 프로시서 outout파라미터를 위한 크기를 지정하기 위해 사용된다. m*ode* 속성에 OUT, INOUT를 지정하고 *jdbcType*를 DECIMAL 이나 NUMERIC로 지정한다면, *numericScale* 값을 지정해야만 한다. 이 속성을 위한 값은 0보다 같거나 큰 정수값이어야만 한다.

<parameterMap> 예제

모든 구조를 사용하는 parameterMap의 예제가 다음과 같다.

```
<parameterMap  id="insert-product-param" class="com.domain.Product">
        <parameter          property="id"          jdbcType="NUMERIC"          javaType="int"
nullValue="-9999999"/>
        <parameter property="description" jdbcType="VARCHAR" nullValue="NO_ENTRY"/>
</parameterMap>

<insert  id="insertProduct"  parameterMap="insert-product-param">
        insert into PRODUCT (PRD_ID, PRD_DESCRIPTION) values (?,?)
</insert>
```

위 예제에서 자바빈즈 프라퍼티인 *id* 와 *description*는 목록화되는 순서대로 매핑 구문 인 *insertProduct* 의 파라미터에 적용될것이다. 그래서 *id*는 첫번째 파라미터(?)에 적용되고 *description* 는 두번째 파라미터에 적용된다. 만약에 순서가 반대라면 XML은 다음처럼 보일것이다.

```
<parameterMap id="insert-product-param" class="com.domain.Product">
        <parameter property="description" />
        <parameter property="id"/>
</parameterMap>

<insert id="insertProduct"  parameterMap="insert-product-param">
```

```
        insert into PRODUCT (PRD_DESCRIPTION, PRD_ID) values (?,?)
</insert>
```

주의*!* Parameter Map 이름은 정의된 SQL Map XML파일에 위치한다. 당신은 SQL Map(<sqlMap> root태그에 셋팅된)의 id와 함께 파라미터 Map의 id를 앞에 붙임으로써 다른 SQL Map XML파일내에 파라미터 Map을 참조할 수 있다. 예를 들면 다른 파일로부터 위의 파라미터 map를 참조하기 위해 참조하기 위한 전체이름은 "Product.insert-product-param"이 될것이다.

인라인 파라미터 맵

매우 상세한 설명에도 불구하고 parameterMaps을 선언하기 위한 위의 문법은 매우 장황하다. 파라미터 Maps을 위한 정의(definition)을 간단하게 하고 코드를 줄일수 있는 좀더 다양한 문법이 있다. 그 대안적인 문법은 자바빈즈 프라퍼티이름을 매핑 구문에 인라인시키는 것이다. 초기설정에 의해 명시적으로 정의된 parameterMap 이 없는 어떤 매핑 구문는 인라인 파라미터를 위해 파싱될것이다. 이전의 인라인 파라미터를 구현한 예제(이를 테면 Product)는 다음처럼 보일것이다.

```
<insert id="insertProduct" parameterClass="com.domain.Product">
        insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
        values (#id#, #description#)
</insert>
```

타입을 선언하는 것은 다음의 문법을 사용함으로써 인라인 파라미터로 할수 있다.

```
<insert id="insertProduct" parameterClass="com.domain.Product">
        insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
        values (#id:NUMERIC#, #description:VARCHAR#)
</insert>
```

타입을 선언하는 것과 null값 대체는 다음 문법을 사용함으로써 인라인 파라미터로 할수 있다.

```
<insert id="insertProduct" parameterClass="com.domain.Product">
        insert into PRODUCT (PRD_ID, PRD_DESCRIPTION)
        values (#id:NUMERIC:-999999#, #description:VARCHAR:NO_ENTRY#)
</insert>
```

주의*!* 인라인 파라미터를 사용할 때 당신은 타입정의없이 null값 대체를 명시할수 없다. 당신은 순서대로 파싱하기 위해 둘다 명시해야 한다.

주의*!* Null값의 완전한 투명성을 원한다면 당신은 이 문서의 나중에 설명되는것처럼 당신의 결과 맵s내에 null값 대체를 반드시 명시해야 한다.

주의*!* 당신이 많은 수의 타입 서술자와 null값 대체가 필요하다면 당신은 외부적인 것을 사용해서 코드를 정리할수 있어야 할것이다.

인라인 파라미터 맵 구문

iBATIS는 인라인 파라미터 맵을 위해 두가지의 문법을 지원한다. 하나는 간단한 문법이고 다른 하나는 좀더 상세하고 복잡한 문법이다.

간단한 문법은 다음과 같다.

#propertyName# - OR -
#propertyName:jdbcType# - OR -
#propertyName:jdbcType:nullValue#

이 문법의 예제는 위와 같다. *PropertyName* 요소는 파라미터 객체의 프라퍼티 명(또는 파라미터 객체가 String, Integer처럼 간단한 값이라면 파라미터 객체 자체의 값)이다. *JdbcType* 요소는 파라미터의 JDBC타입을 지정하기 위해 사용된다. 값은 java.sql.Types(VARCHAR, INTEGER, 등등.)의 목록중 하나의 타입이어야만 한다. 대개 *jdbcType* 요소는 값이 NULL일수 있거나 DATE나 TIME필드를 사용하도록 지정할 가능성이 있다면 필요하다. *NullValue* 요소는 위에서 언급된 것처럼 NULL대체값을 지정하기 위해 사용된다. *JdbcType* 을 지정하지 않는다면 *nullValue*를 지정할수 없다는 것에 주

의하라.
이 문법은 일반적인 형태의 파라미터 맵의 몇가지 고급 옵션을 사용할 필요가 없을때(예를 들어, 저장 프로시저를 호출할 때) 대부분 적절한다.

좀더 상세한 문법은 다음과 같다.

#propertyName,javaType=?,jdbcType=?,mode=?,nullValue=?,handler=?,numericScale=?#

"?"는 해당 속성에 대해 지정해야하는 값이다.

상세한 문법은 일반적인 파라미터 맵의 대부분의 값을 사용하도록 해준다. *PropertyName* 요소는 필수이고 나머지는 선택적이다. propertyName 요소가 가장 먼저 나와야 한다는 것을 제외하고는 이 값들은 특별한 순서를 가지지 않으므로 임의의 순서로 지정하면 된다. 각각의 속성들을 위해 사용되는 값들은 일반적인 형태의 파라미터 맵을 사용할때 사용된 값과 같다. 이 문법에 대해 유의하라. *Handler* 속성은 별칭이 등록되는 것처럼 타입 핸들러를 위한 별칭 형태의 이름을 사용할 것이다. 저장 프로시저를 호출하기 위해 사용된 이 분법의 예제는 다음과 같다.

```
<procedure id="callProcedure" parameterClass="com.mydomain.MyParameter">
	{call MyProcedure
	(#parm1,jdbcType=INTEGER,mode=IN#, #parm2,jdbcType=INTEGER,mode=IN#,
	 #parm3,jdbcType=DECIMAL,mode=OUT,numericScale=2#)}
</procedure>
```

원시타입 파라미터

파라미터처럼 사용하기 위해 자바빈을 쓰는 것은 언제나 필요하고 편리한 것은 아니다. 이런 경우에 당신은 직접적으로 파라미터를 사용하는것처럼 원시타입 래퍼객체(String, Integer, Date등등)를 사용하는 것을 환영할것이다. 예를 들면
:

```
<select id="insertProduct" parameter="java.lang.Integer">
	select * from PRODUCT where PRD_ID = #value#
</select>
```

PRD_ID가 숫자 타입이라고 가정하자. 호출이 되었을 때 java.lang.Integer객체를 전달할수 있는 매핑 구문를 만들것이다. **#value#** 파라미터는 Integer인스턴스의 값으로 대체될것이다. **"value"**라는 이름은 간단한 문법(이를 테면 괄호) 안의 요소이고 별명이 될수 있다. 결과 맵는 result처럼 원시타입을 잘 지원한다. 파라미터로 원시타입을 사용하는 방법에 대해서 좀더 다양한 정보를 위해서는 결과 맵섹션과 프로그래밍 SQL Maps(API)를 보라.

원시 타입은 좀더 간결한 코드를 위해서 별칭된다. 예를 들면 "int"는 "java.lang.Integer"대신에 사용될수 있다. 별칭 아래의 "파라미터 Map과 결과 맵을 위해 지원되는 타입" 이라는 제목의 테이블에서 이야기 된다.

Map 타입 파라미터

당신이 자바빈즈 클래스를 쓰는 것이 필요하지 않거나 편리하지 않은 위치에 있고 하나의 원시타입 파라미터을 쓰지는 않는다면 파라미터객체로 Map(이를 테면 HashMap, TreeMap)을 사용할수 있다. 예를 들면 :

```
<select  id="insertProduct"  parameterClass="java.util.Map">
	select * from PRODUCT
	where PRD_CAT_ID = #catId#
	and PRD_CODE = #code#
</select>
```

매핑 구문구현내에서는 차이점이 없다는 것을 알라. 위의 예제에서 만약 Map인스턴스가 statement를 위한 호출로 전달되었다면 Map은 "catId" 과 "code" 라는 이름의 키를 포함해야만 한다. 이 값은 Integer과 String과 같은 선호되는 타입이 되는 그런 키에 의해 참조된다. 결과 맵은 result처럼 Map타입을 아주 잘 지원한다. 파라미터처럼 Map타입을 사용하는것에 대한 좀더 상세한 정보를 위해서는 결과 맵섹션과 프로그래밍 SQL Map(API)를 보라.

Map 타입 역시 좀더 간결한 코드를 위해 별칭된다. 예를 들면 "map"는 "java.util.Map"을 대신할수 있다. 별칭은 아래의 "파라미터 Map과 결과 맵을 위해 지원되는 타입" 이라는 제목의 테이블에서 이야기 된다.

대체 문자열

iBATIS는 SQL을 실행하기 위해 언제나 JDBC prepared구문을 사용한다. JDBC prepared구문은 "파라미터 제조기(parameter markers)"를 사용해서 파라미터를 지원한다. 파라미터 맵과 인라인 파라미터 모두 iBATIS에 지정된 파라미터 대신에 파라미터 제조기로 SQL을 생성하도록 한다. 예를 들어, 구문을 다음처럼 만든다면

select * from PRODUCT where PRD_ID = #value#

iBATIS는 이 SQL문자열로 prepared구문을 생성할것이다.

select * from PRODUCT where PRD_ID = ?

데이터베이스는 대부분의 경우 파라미터 제조기를 허용하지만 SQL구문의 일부에 대해서만이다. 예를 들어, 다음과 같은 형태의 구문은 허용되지 않는다.

select * from ?

데이터베이스는 이 구문은 처리할수는 없을것이다. 왜냐하면 사용할 테이블의 무엇인지 알지 못하기 때문이다. 그래서 다음처럼 구문을 지정한다면,

select * from #tableName#

SQLException을 보게 될것이다.

이러한 몇가지 이슈를 극복하기 위해, iBATIS는 구문이 준비되기 전에 문자열을 SQL로 대체하기 위한 문법을 제공한다. 동적인 SQL구문을 생성하기 위해 대체 문자열 지원을 사용할수 있다. 대체 문자열의 예제는 다음과 같다.

select * from $tableName$

이러한 문법으로, iBATIS는 구문이 준비되기 전에 "tableName" 프라퍼티의 값을 SQL로 대체할것이다. 이러한 지원 기능으로, 문자열을 SQL구문의 일부로 대체할수 있다.

중요한 노트 ***1***: 이 지원기능은 String타입만을 대체한다. 그래서 Date나 Timestamp와 같은 복잡한 데이터 타입에는 적절하지 않다.

중요한 노트 ***2***: SQL select구문에서 테이블 이름, 칼럼목록을 변경하기 위해 이 기능을 사용한다면, 언제나 지정해야만 한다.

결과맵

결과 맵은 SQL Maps의 가장 중요한 컴포넌트이다. resultMap는 자바빈즈 프라퍼티를 매핑된 쿼리 구문을 실행함으로써 생산된 ResultSet의 칼럼에 매핑시키는 책임을 진다. resultMap의 구조는 다음과 같이 보인다.

```
<resultMap  id="resultMapName"  class="some.domain.Class"
     [extends="parent-resultMap"]
     [groupBy="some property list"]>
        <result property="propertyName"  column="COLUMN_NAME"
                     [columnIndex="1"]  [javaType="int"]  [jdbcType="NUMERIC"]
                     [nullValue="-999999"] [select="someOtherStatement"]
                     [resultMap="someOtherResultMap"]
                     [typeHandler="com.mydomain.MyTypehandler"]
                     />
        <result ....../>
        <result ....../>
        <result ....../>
</resultMap>
```

[괄호] 부분은 옵션이다. resultMap는 구문을 참조하기 위해 사용할 *id*속성을 가진다. resultMap는 클래스나 타입별칭의 전체경로를 포함한 이름인 *class*속성을 가진다. 이 클래스는 이것을 포함하는 result매핑에 기반하여 초기화되고 생성될것이다. ***Extends***속성은 resultMap에 기초한 다른 resultMap의 이름을 옵션적으로 셋팅할수 있다. 이것은 상위

resultMap의 모든 프라퍼티가 하위 resultMap의 부분을 포함하는것처럼 자바내에서 클래스를 확장하는것과 유사하다. 상위 resultMap의 프라퍼티는 하위 resultMap프라퍼티와 부모 resultMap가 자식 앞에서 정의되기 전에 언제나 추가된다. 상위/하위 resultMap를 위한 클래스는 같은 것을 필요로 하지 않을뿐 아니라 모든 것이 관련될 필요도 없다.

resultMap 은 자바빈즈를 ResultSet의 칼럼에 매핑시키는 어느정도의 프라퍼티 매핑을 포함할수 있다. 그런 프라퍼티 매핑은 문서내에서 정의하기 위해 적용될것이다. 관련 클래스는 각각의 프라퍼티, Map또는 XML을 위한 get/set메소드를 가진 자바빈즈와 호환되는 클래스여야만 한다.

주의*!* 칼럼은 결과 맵내에서 정의되기 위해서 명시적으로 읽을것이다.

다음의 섹션은 property요소의 속성들을 서술한다.

property

결과 맵의 *property* 속성은 매핑 구문에 의해 반환되는 결과 객체의 자바빈즈 프라퍼티(get메소드) 이름이다. 이름은 결과를 생성할 때 필요한 횟수에 의존적인 값보다 더 크게 사용될수 있다.

column

column 속성값은 프라퍼티를 생성하기 위해 사용될 값들로부터의 ResultSet내의 칼럼의 이름이다.

columnIndex

옵션적인(최소한의) 성능향상을 위해서 columnIndex 속성값은 자바빈즈 프라퍼티를 생성하기 위해 사용될 값으로부터의 ResultSet내의 칼럼의 인덱스이다. 이것은 애플리케이션의 99%정도엔 필요하지 않을것이고 유지를 위한 노력과 속도를 위해 가독성을 희생한다. 몇몇 JDBC드라이버는 다른것들이 동적으로 속도를 올려주는 동안 어떤 성능이득도 구체화하지 않을것이다.

jdbcType

jdbcType 속성은 자바빈즈 프라퍼티를 생성하는데 사용되는 ResultSet 칼럼의 데이터베이스 칼럼 타입을 명시적으로 정의하는데 사용된다. 비록 결과 맵이 null값과 함께 같은 어려움을 가지지 않는다고 하더라도 Date프라퍼티처럼 어떤 매핑 타입을 위해 유용할수 있는 타입을 정의한다. 자바는 오직 하나의 Date값 타입을 가지고 SQL 데이터베이스는 여러가지를 가지기 때문에 dates(또는 다른타입)타입을 정확하게 셋팅하는 것을 확신하는 몇몇경우에 필요하게 될것이다. 유사하게도 String타입은 VARCHAR, CHAR 또는 CLOB에 의해 생성될것이다. 그래서 그런 경우에 필요한 타입을 정의하라.

javaType

javaType 속성은 셋팅되는 프라퍼티의 자바프라퍼티타입을 명시적으로 정의하기 위해 사용된다. 대개 이것은 리플렉션(reflection)을 통해 자바빈즈 프라퍼티로부터 끌어낼수 있다. 하지만 Map와 XML매핑과 같은 매핑은 프레임워크를 위한 타입을 제공할수 없다. 만약 javaType가 셋팅되지 않고 프레임워크가 그 타입을 구분할수 없다면 타입은 객체로 가정되어 처리될것이다.

nullValue

nullValue 속성은 데이터베이스내에서 NULL값을 대신해서 사용되기 위한 값을 정의한다. 그래서 만약 ResultSet으로부터 NULL이 읽었다면 자바빈 프라퍼티는 NULL대신에 nullValue속성에 의해 정의된 값을 셋팅할것이다. null 속성값은 어떠한 값을 될수 있지만 프라퍼티타입을 위해서는 적절해야만 한다.

만약 당신의 데이터베이스가 NULL이 가능한 칼럼을 가진다면 당신은 당신의 애플리케이션이 다음처럼 결과 맵에서 그것을 정의할수 있는 변수값과 함께 NULL을 표시하기를 원한다.

```
<resultMap id="get-product-result" class="com.ibatis.example.Product">
        <result property="id" column="PRD_ID"/>
```

```
        <result property="description" column="PRD_DESCRIPTION"/>
        <result property="subCode" column="PRD_SUB_CODE" nullValue="-999"/>
    </resultMap>
```

위 예제에서 만약 PRD_SUB_CODE이 NULL로 읽혀진다면 subCode 프라퍼티는 -999라는 값으로 셋팅될것이다. 이것은 당신에게 데이터베이스내에서 NULL이 가능한 칼럼을 표현하기 위해 당신의 자바클래스내에 원시타입을 사용하도록 허락할것이다. 만약 당신이 updates/inserts같은 쿼리를 위한 작업을 수행하기를 원할 때 당신은 파라미터 맵에 nullValue를 정의해야만 한다는 것을 기억해라.

select

select 속성은 객체사이의 관계를 서술하고 자동적으로 복합 프라퍼티 타입을 로드하는데 사용된다. 구문 프라퍼티값은 다른 매핑 구문의 이름이 되어야만 한다. 데이터베이스 칼럼값은 구문 속성이 파라미터처럼 관계된 매핑 구문으로 전달하는것처럼 같은 property요소내에 정의된다. 그러므로 칼럼은 원시타입으로 지원이 되어야 한다. 지원되는 원시타입과 복합 프라퍼티 매핑/관계에 대한 상세정보는 이 문서 나중에 이야기 된다.

resultMap

resultMap 속성은 결과 매핑에서 재사용될수 있는 내포된 resultMap을 언급하기 위해 사용된다. 이 속성은 1:1관계나 1:N관계에서 사용될수 있다. 1:N관계라면, 프라퍼티는 Collection(List, Set, Collection 등등)이 되어야만 한다. iBATIS가 레코드를 그룹화하는 방법을 나타내는 resultMap요소의 groupBy속성을 지정해야만 한다. 1:1관계에서는, 프라퍼티가 어떤 타입이어도 상관없고 groupBy속성을 지정하든 하지 않든 문제가 되지 않는다. 몇가지 조인이 1:N이고 몇가지는 1:1인 경우에 groupBy속성을 사용할수 있다.

typeHandler

typeHandler 속성은 디폴트 iBATIS타입대신에 이 프라퍼티를 위해 사용될 사용자 정의 타입 핸들러를 명시하기 위해 사용된다. 명시한다면, 값은 com.ibatis.sqlmap.engine.type.TypeHandler 인터페이스나 com.ibatis.sqlmap.client.extensions.TypeHandlerCallback 인터페이스를 구현하는 패키지 경로를 포함한 클래스명이어야만 한다. 이 값은 전역 타입 핸들러를 무시한다. 이 문서의 뒷부분에 사용자 정의 타입 핸들러에 대해 상세히 다룬다.

내포하는 결과 맵

만약 당신이 명시적으로 정의된 resultMap의 재사용을 요구하지 않는다는 매우 간단한 요구사항을 가진다면 매핑 구문의 resultClass속성을 셋팅함으로써 결과 맵을 함축적으로 정의하는 빠른 방법이 있다. 이 묘기는 당신이 반환되는 결과 세트가 당신의 자바빈의 쓰기 가능한 프라퍼티 이름에 매치되는 칼럼이름(또는 라벨/별칭)을 가지는 것을 확실해야만 한다는것이다. 예를 들면 만약 우리가 위에서 서술된 Product클래스를 생각할 때 우리는 다음처럼 내포하는 결과 맵으로 매핑 구문를 생성할수 있다.

```
<select id="getProduct" resultClass="com.ibatis.example.Product">
    select
      PRD_ID as id,
      PRD_DESCRIPTION as description
     from PRODUCT
     where PRD_ID = #value#
</select>
```

위의 매핑 구문는 resultClass를 표기하고 Product클래스의 자바빈즈 프라퍼티에 매치되는 각각의 칼럼를 위한 별칭을 명시한다. 이것은 모두 필수(required) 이다. 결과 맵은 필요하지 않다. 여기서 교환(tradeoff)은 당신이 칼럼타입(대개 필수가 아닌)과 null값(또는 다른 어떤 프라퍼티 속성)을 정의하는 기회를 가지지 않는것이다. 많은 데이터베이스가 대소문자를 가리지 않기 때문에 내포된 결과 맵는 또한 가리지 않는다. 만약 당신의 자바빈이 두개의 프라퍼티를 가진다면 하나의 이름은 firstName이고 다른 것은 firstname이다(두개의 값은 대소문자의 차이이다). 그것들은 동일하고 당신은 내포된 결과 맵를 사용할수없을것이다(이것은 자바빈 클래스의 디자인에서 잠재적인 문제점을 파악하게 될것이다.). 게다가 resultClass를 통해 자동매핑을 하면 몇몇 성능에 관련된 부하가 발생할것이다. ResultSetMetaData에 접근하는 것은 몇몇 쓰여진 JDBC드라이버로는 느리게 만들수 있다.

원시타입의 결과 **(**이를 테면 **String, Integer, Boolean)**

자바빈 호환 클래스를 지원하기 위해 추가적으로 결과 맵은 String, Integer, Boolean 등등과 같은 간단한 자바타입 래퍼를 편리하게 생성할수 있다. 원시타입객체의 collection은 밑에서 이야기 되는 API(executeQueryForList()를 보라)들을 사용해서 가져올수 있다. 원시타입은 자바빈처럼 같은 방법으로 정확하게 매핑된다. 원시타입은 당신이 선호하는(대개 "value" 또는 "val") 이름형식의 어떤것처럼 될수 있는 하나의 프라퍼티만을 가질수 있다. 예를 들면 우리가 전체 Product 클래스 대신에 모든 product서술자(description)의 목록만을 로드하길 원한다면 맵은 다음처럼 보여질것이다.

```
<resultMap id="get-product-result" class="java.lang.String">
        <result property="value" column="PRD_DESCRIPTION"/>
</resultMap>
```

좀더 간단한 접근법은 매핑 구문안에서 간단하게 결과 클래스를 사용하는것이다.("as"키워드를 사용해서 "value"라는 칼럼별칭을 사용하는 것을 주의깊게 보라.)

```
<select  id="getProductCount" resultClass="java.lang.Integer">
        select count(1) as value
        from PRODUCT
</select>
```

Map Results

결과 맵s은 HashMap또는 TreeMap처럼 Map인스턴스를 편리하게 생성할수 있다. 그런 객체(Map의 List)의 collection은 아래에서 이야기되는 API(executeQueryForList()를 보라)들을 사용해서 가져올수 있다. Map타입은 자바빈과 같은 방법으로 정확하게 매핑된다. 하지만 자바빈 프라퍼티셋팅 대신에 Map의 key들은 대응되는 매핑칼럼을 위한 값을 참조하도록 셋팅한다. 예를 들면 만약 우리가 product의 값을 Map으로 빨리 로드시키길 원한다면 우리는 다음처럼 할것이다.

```
<resultMap id="get-product-result" class="java.util.HashMap">
        <result  property="id" column="PRD_ID"/>
        <result  property="code" column="PRD_CODE"/>
        <result  property="description" column="PRD_DESCRIPTION"/>
        <result  property="suggestedPrice" column="PRD_SUGGESTED_PRICE"/>
</resultMap>
```

위 예제에서 HashMap인스턴스는 Product데이터를 생성할것이다. 프라퍼티 이름 속성(이를 테면 "id")은 HashMap의 키가 될것이다. 매핑칼럼의 값은 HashMap의 값이 될것이다. 물론 당신은 Map타입을 가지고 내포된 결과 맵을 사용할수도 있다. 예를 들면 :

```
<select  id="getProductCount" resultClass="java.util.HashMap">
        select * from PRODUCT
</select>
```

위의 것은 반환된 ResultSet의 Map표현을 당신에게 줄것이다.

복합**(Complex)** 프라퍼티 **(**이를 테면 사용자에 의해 정의된 클래스의 프라퍼티**)**

이것은 선호하는 데이터와 클래스를 로드하는 방법을 알고있는 매핑 구문와 함께 관련된 resultMap프라퍼티에 의해 복합타입의 프라퍼티(사용자에 의해 생성된 클래스)를 자동적으로 생성하는 것은 가능하다. 데이터베이스내 데이터는 언제나 복합프라퍼티는 관계의 "many side"로부터이고 프라퍼티 자신은 관계의 "one side"로 부터이다라는 것을 고정하는 클래스에서 1:1관계 또는 1:M관계를 통해 표현된다. 예를 들면 :

```
<resultMap id="get-product-result" class="com.ibatis.example.Product">
        <result property="id" column="PRD_ID"/>
        <result property="description" column="PRD_DESCRIPTION"/>
        <result property="category" column="PRD_CAT_ID" select="getCategory"/>
</resultMap>

<resultMap id="get-category-result" class="com.ibatis.example.Category">
        <result property="id" column="CAT_ID"/>
```

```
        <result property="description" column="CAT_DESCRIPTION"/>
</resultMap>

<select  id="getProduct"  parameterClass="int" resultMap="get-product-result">
        select * from PRODUCT where PRD_ID = #value#
</select>

<select  id="getCategory"  parameterClass="int" resultMap="get-category-result">
        select * from CATEGORY where CAT_ID = #value#
</select>
```

위 예제에서 *Product*의 인스턴스는 *Category*타입의 *category*를 호출하는 프라퍼티를 가진다. Category는 복합사용자타입이기 때문에 JDBC는 그것을 생성하는 방법을 가지지 않는다. 프라퍼티매핑과 함께 다른 매핑 구문를 관련시킴으로써 우리는 그것을 생성하기 위한 SQL Map엔진을 위해 충분한 정보를 제공한다. *getProduct*를 수행하면 *get-product-result* 결과 맵이 *PRD_CAT_ID*칼럼내 반환되는 값을 사용해서 *getCategory*을 호출할것이다. *get-category-result* 결과 맵은 Category를 초기화할것이고 그것을 생성한다. 전체 Category인스턴스는 Product의 category프라퍼티로 셋팅한다.

N+1 조회(1:1) 피하기

위 솔루션을 사용할 때 문제점은 당신이 Product를 로드할때마다 두개(Product를 위해 하나 그리고 Category를 위해서 하나. 총 2개)의 SQL문이 실제적으로 구동된다는것이다. 이 문제는 하나의 Product를 로드할때는 큰문제가 아닌것처럼 보이지만 만약 10개의 Product를 로드하는 쿼리를 한다면 각각의 쿼리는 관련된 category를 로드하기 위한 Product를 위해서도 실행될것이다. 결과적으로 11번의 쿼리를 하게 된다. Product의 목록을 위해 하나, 관련된 Category를 로드하기 위해 반환되는 Product를 위해 하나씩(N+1 또는 이 경우엔 10+1=11)

해결법은 분리된 select 문 대신에 조인과 내포된(nested)프라퍼티 매핑을 사용하는 것이다. 여기에 그와 같은 상황을 사용한 예제가 있다.

```
<resultMap id="get-product-result" class="com.ibatis.example.Product">
        <result property="id" column="PRD_ID"/>
        <result property="description" column="PRD_DESCRIPTION"/>
        <result property="category.id" column="CAT_ID" />
        <result property="category.description" column="CAT_DESCRIPTION" />
</resultMap>

<select  id="getProduct"  parameterClass="int" resultMap="get-product-result">
        select *
        from PRODUCT, CATEGORY
        where PRD_CAT_ID=CAT_ID
        and PRD_ID = #value#
</select>
```

iBATIS 버전 2.2.0 또는 그 이후의 버전에서, 칼럼을 반복하는 대신에 1:1쿼리로 결과맵을 재사용할수 있다. 이 사용법의 예제는 다음과 같다.

```
<resultMap id="get-product-result" class="com.ibatis.example.Product">
        <result property="id" column="PRD_ID"/>
        <result property="description" column="PRD_DESCRIPTION"/>
        <result property="category" resultMap="get-category-result"  />
</resultMap>

<resultMap id="get-category-result" class="com.ibatis.example.Category">
        <result property="id" column="CAT_ID" />
        <result property="description" column="CAT_DESCRIPTION" />
</resultMap>

<select  id="getProduct"  parameterClass="int" resultMap="get-product-result">
        select *
        from PRODUCT, CATEGORY
```

```
        where PRD_CAT_ID=CAT_ID
        and PRD_ID = #value#
</select>
```

늦은**(Lazy)** 로딩 대 조인**(1:1)**

조인을 사용하는 것이 언제나 더 좋은 결과를 내지는 않는다는 것에 주의하는 것은 중요하다. 만약 당신이 관계객체에 접근하는 것이 거의 없는 상황이라면 조인을 피하는 것이 더 빠르고 모든 category프라퍼티의 로딩이 불필요하다. 이것은 outer조인을 포함하는 데이터베이스 디자인이나 null값이 가능하거나 인덱스가 없는 칼럼에는 사실이다. 이런 상황에서 늦은 로딩과 bytecode향상 옵션으로 sub-select솔류션을 사용하는 것은 좀더 향상된 결과를 보여준다. 일반적인 규칙은 연관된 프라퍼티에 접근하는 것을 좀더 하고자 할때만 조인을 사용하라. 반면에 늦은 로딩이 옵션이 아닐때에만 그것을 사용하라.

만약 당신이 사용할 방법을 결정하는데 문제가 있다면 걱정하지 마라. 그것은 문제도 아니다. 당신은 자바코드 충돌없이 이것을 항상 변경할수있다. 위의 두 예제는 같은 객체형태의 결과를 보이고 정확하게 같은 메소드 호출을 사용해서 로드된다. 만약 당신이 캐쉬를 가능하게 하면 단지 하나의 고려사항은 *separate select*(조인이 아닌) 솔루션을 사용하는 것이 반환되는 캐쉬된 인스턴스내에 결과를 보이게 된다.

복합 **Collection** 프라퍼티

복합 객체의 목록을 표현하는 프라퍼티를 로드하는 것은 가능하다. 데이터베이스내의 데이터는 M:M관계나 1:M관계에 의해 표현될것이다. 객체목록을 로드하는 것은 구문에 어떤 변경사항도 주지 않는다. SQL Map프레임워크가 비즈니스객체내에서 리스트처럼 프라퍼티를 로드하기위해 요구되는 단 하나의 차이점은 java.util.List 또는 java.util.Collection 타입이 되어야 한다는것이다. 예를들면 Category가 Product 인스턴스 목록을 가진다면 매핑은 다음처럼 보일것이다. (Category가 java.util.List타입의 "productList"라고 불리는 프라퍼티를 가진다고 가정하자.)

```
<resultMap id="get-category-result" class="com.ibatis.example.Category">
        <result property="id" column="CAT_ID"/>
        <result property="description" column="CAT_DESCRIPTION"/>
        <result property="productList" column="CAT_ID" select=" getProductsByCatId"/>
</resultMap>

<resultMap id="get-product-result" class="com.ibatis.example.Product">
        <result property="id" column="PRD_ID"/>
        <result property="description" column="PRD_DESCRIPTION"/>
</resultMap>

<select id="getCategory"  parameterClass="int" resultMap="get-category-result">
        select * from CATEGORY where CAT_ID = #value#
</select>

<select  id="getProductsByCatId"  parameterClass="int" resultMap="get-product-result">
        select * from PRODUCT where PRD_CAT_ID = #value#
</select>
```

N+1 조회**(1:M** 과 **M:N)** 피하기

이것은 위의 1:1 상황과 유사하다. 하지만 굉장히 많은 데이터를 포함할 때 좀더 큰 걱정거리가 될것이다. 위 해결법과 함께 문제는 당신이 Category를 로드할때마다 두개의 SQL문(하나는 Category를 위한 하나이고 하나는 Products에 대한 목록을 위한 것)은 실질적으로 수행된다. 이 문제는 하나의 Category를 로드할 때 평범한것처럼 보이지만 10개의 Category를 로드하는 쿼리문을 실행할때는 각각의 쿼리가 Product의 목록을 로드하기 위한 각각의 Category를 위해서 수행될것이다. 결과적으로 11개의 쿼리가 수행된다. 하나는 Category 목록을 위한것이고 각각의 Product 관련 목록을 반환하는 각각의 Category를 위한 것이다(N+1 또는 이 경우엔 10+1=11). 이 상환을 더욱 나쁘게 만들려면 우리는 굉장히 많은 데이터를 다루면 된다.

1:N 과 **M:N** 해결법

iBATIS는 이 문제를 해결한다. 다음은 그 예제이다.

```
<sqlMap namespace="ProductCategory">

<resultMap id="categoryResult" class="com.ibatis.example.Category" groupBy="id">
        <result property="id" column="CAT_ID"/>
        <result property="description" column="CAT_DESCRIPTION"/>
        <result property="productList" resultMap="ProductCategory.productResult"/>
</resultMap>

<resultMap id="productResult" class="com.ibatis.example.Product">
        <result property="id" column="PRD_ID"/>
        <result property="description" column="PRD_DESCRIPTION"/>
</resultMap>

<select id="getCategory"  parameterClass="int" resultMap="categoryResult">
        select C.CAT_ID,  C.CAT_DESCRIPTION, P.PRD_ID, P.PRD_DESCRIPTION
        from CATEGORY C
        left outer join PRODUCT P
        on C.CAT_ID = P.PRD_CAT_ID
        where CAT_ID = #value#
</select>
</sqlMap>
```

당신이 호출할때,

```
List myList = queryForList("ProductCategory.getCategory", new Integer(1002));
```

...메인 쿼리는 수행되고 결과는 " com.ibatis.example.Category "타입의 bean인 *myList*변수에 저장된다. List내 각각의 객체는 같은 쿼리로부터 생성되는 List인 " productList "프라퍼티를 가질것이다. 하지만 하위 목록내 bean을 생성하는 " productResult " 결과 맵을 사용한다. 그래서 당신은 하위목록을 포함하는 목록으로 종료하고 오직 하나의 데이터베이스 쿼리만이 수행된다.

가장 중요한 항목은 "categoryResult" 결과 맵에서

groupBy="id"

...속성이고 ...

```
<result property="productList" resultMap="ProductCategory.productResult"/>
```

...프라퍼티 매핑이다. 다른 중요한 사항은 productList프라퍼티를 위한 결과매핑이 명명공간을 인식(이것은 작동하지 않는 "productResult "이 될것이다.)하는것이다.

이 접근법을 사용하여, 당신은 N+1 문제를 풀수 있다.

중요한 노트**:** groupBy를 queryForPaginatedList() API와 조합하는 것은 정의된 행위가 아니며 기대하는 것과 다른 결과를 반환할지도 모른다. 왠만하면 이런 두가지 아이디어를 조합하는 것을 시도하지 말아달라. groupBy를 사용한다면, queryForList 나 queryForObject 메소드를 사용해야만 한다.

내포된 프라퍼티는 java.util.Collection의 구현체이 될수있다. 하지만 프라퍼티를 위한 getter와 setter는 간단할것이고 내부적인 속성에 접근할수있다. iBATIS는 프라퍼티에 접근하기 위해 get메소드를 반복적으로 호출할것이다. 그리고 결과세트를 처리할때 프라퍼티의 add()메소드를 호출한다. getter와 setter로 어떤 특별한 것을 할려고 하지 말라. 이를테면 List의 내부 배열을 포함하도록 시도하는 것과 같은 행위는 결과적으로 iBATIS가 처리하는 것을 실패하도록 만들것이다. iBATIS가 객체를 일괄적으로 가져와서 한번에 set메소드를 호출하는 공통적으로 잘못된 개념이 있다. get메소드가 null을 반환할때 iBATIS가 set메소드를 호출하는 경우는 아니다. 이 경우 iBATIS는 프라퍼티의 디폴트 구현체를 생성하고 결과객체에 새로운 객체를 셋팅한다. 새롭게 생성된 객체는 언제가 비어있을것이다. iBATIS가 프라퍼티를 가져오기 위해 get메소드를 호출하고 add메소드를 호출할것이기 때문이다.

늦은**(Lazy)** 로딩 대 조인**(1:M and M:N)**

먼저 이야기된 1:1상황처럼 조인을 사용하는 것이 언제나 더 좋다는 것이 아니라는 것을 아는 것은 중요하다. 이것은 대량의 데이터로 인하여 개별적인 값 프라퍼티를 위한 것보다 collection프라퍼티에서 좀더 사실적이다. 만약 당신이 관련된 객체에 접근하는 것이 드문 상황(이를 테면 Category클래스의 *productList* 프라퍼티*)*이라면 이것은 조인과 product목록의

필요없는 로딩을 피한다면 정말 빠르게 될것이다. 이것은 outer조인과 null이 가능하고 아니면 또는 인덱스가 없는 칼럼을 포함한 데이터베이스 디자인에는 특별히 사실이다. 이런 상황에서 늦은(lazy)로딩과 bytecode향상옵션으로 sub-select솔루션을 사용하는 것은 좀더 향상시켜준다. 일반적인 규칙은 연관된 프라퍼티에 접근하는 것을 좀더 하고자 할때만 조인을 사용하라. 반면에 늦은 로딩이 옵션이 아닐때에만 그것을 사용하라.

먼저 언급했던 것 처럼 만약 당신이 어떤 방법을 사용해야 하는지 결정하는데 문제가 있다면 걱정하지 마라. 어떤 방법을 사용할지에 대해서 걱정하는 것은 필요없는 일이다. 당신은 당신의 자바코드에 충돌없이 그것을 변화시킬수 있다. 위의 두 예제는 같은 객체형태의 결과를 보이고 정확하게 같은 메소드 호출을 사용해서 로드된다. 만약 당신이 캐쉬를 가능하게 하면 단지 하나의 고려사항은 *separate select*(조인이 아닌) 솔루션을 사용하는 것이 반환되는 캐쉬된 인스턴스내에 결과를 보이게 된다.

복합 키 또는 다중 복합 파라미터 프라퍼티

당신은 위 예제에서 column 속성에 의해 resultMap내에 정의된 것처럼 사용되어지는 것은 하나의 키라는 것이 언급되었다. 이것은 단지 하나의 키만이 관계된 매핑 구문에 관련 될 수 있다는 것을 제안했다. 어쨌든 관계된 매핑 구문에 전달할 다중 칼럼을 허락하는 대안적인 문법이 있다. 이것은 복합키 관계가 존재하는 상황이나 당신이 간단하게 #value#와 다른 이름의 파라미터를 사용하고자 할 때 편리하다. Column 속성이 간단{param1=column1, param2=column2, …, paramN=columnN}할 때 대안적인 문법이다. PAYMENT테이블이 Customer ID와 Order ID를 둘다 키로 할 때 다음의 예제를 보고 생각해보라.

```
<resultMap id="get-order-result" class="com.ibatis.example.Order">
        <result property="id" column="ORD_ID"/>
        <result property="customerId" column="ORD_CST_ID"/>
        …
        <result property="payments" column="{itemId=ORD_ID, custId=ORD_CST_ID}"
              select="getOrderPayments"/>
</resultMap>

<select id="getOrderPayments" resultMap="get-payment-result">
        select * from PAYMENT
        where PAY_ORD_ID  = #itemId#
        and PAY_CST_ID  = #custId#
</select>
```

옵션적으로 당신은 그것들이 파라미터처럼 같은 순서로 정렬되는 것처럼 칼럼이름을 정의할 수 있다. 예를 들면

{ORD_ID, ORD_CST_ID}

언제나 처럼 이것은 읽기와 유지라는 것의 영향과 함께 미세한 성능 획득이 있다.

중요***!*** 현재의 SQL Map프레임워크는 순환하는 관계를 자동으로 해석하지 않는다. 부모/자식 관계(트리)를 구현할 때 이것을 알고 있어라. 쉬운 대안은 간단하게 부모객체를 로드하기 않는 경우를 위한 하나 또는 "N+1 피하기" 해결법에서 서술된 조인을 사용하는 경우를 위한 두번째 결과 맵를 정의하는것이다.

주의***!*** 몇몇 JDBC드라이버(이를 테면 내장된 PointBase)는 동시에 다중 ResultSet(커넥션마다)을 지원하지 않는다. 그런 드라이버는 SQL Map엔진이 다중 ResultSet 커넥션을 요구하기 않기 때문에 복잡한 객체 매핑과는 작동하지 않을것이다. 다시 말해 조인을 사용하는거 대신에 이것을 해석할수 있다.

주의***!*** 결과 맵 이름은 언제나 그것들이 정의된 SQL Map XML파일에 위치한다. 당신은 SQL Map의 이름을 결과 맵의 이름앞에 위치시킴으로써 다른 SQL Map XML파일내의 결과 맵를 참조할수 있다.

만약 당신이 *JDBC*를 위해 *MS*의 *SQL Server2000* 드라이버를 사용한다면 당신은 수동 트랜잭션 모드인 동안 다중 *statement*를 수행하기 위해 *connection url*에 *SelectMethod=Cursor*을 추가할 필요가 있을지도 모른다.*(MS*의 지식 기반 기사 *313181*을 보라. *http://support.microsoft.com/default.aspx?scid=kb%3Ben-us%3B313181).*

파라미터 맵과 결과 맵을 지원하는 타입들

파라미터와 결과를 위해 iBATIS 프레임워크에 의해 지원되는 자바타입은 다음과 같다.

자바타입	자바빈/맵 프라퍼티 매핑	결과 클래스 / 파라미터 클래스***	타입 별칭**
boolean	YES	NO	boolean
java.lang.Boolean	YES	YES	boolean
byte	YES	NO	byte
java.lang.Byte	YES	YES	byte
short	YES	NO	short
java.lang.Short	YES	YES	short
int	YES	NO	int/integer
java.lang.Integer	YES	YES	int/integer
long	YES	NO	long
java.lang.Long	YES	YES	long
float	YES	NO	float
java.lang.Float	YES	YES	float
double	YES	NO	double
java.lang.Double	YES	YES	double
java.lang.String	YES	YES	string
java.util.Date	YES	YES	date
java.math.BigDecimal	YES	YES	decimal
* java.sql.Date	YES	YES	N/A
* java.sql.Time	YES	YES	N/A
* java.sql.Timestamp	YES	YES	N/A

* *java.sql. date* 타입 사용은 좋지않다*(discouraged).* 대신에 *java.util.Date* 를 사용하는 것이 제일좋다*.*

** .파라미터나 *result*클래스를 정의할 때 타입별칭은 전체경로의 클래스명에 두는 것이 좋다*.*

******* *int, boolean and float* 와 같은 원시타입은 *iBATIS* 데이터베이스 레이어가 완전한 객체지향접근법을 사용하는것처럼 직접적으로 원시타입을 지원하지는 않는다*.* 그러므로 모든 파라미터와 *result*는 그들의 상위레벨에서 객체가 되어야 한다*.* 부수적으로 *JDK1.5*의 *autoboxing*기능은 잘 사용되기 위해 원시타입을 허락한다*.*

사용자 정의 타입 핸들러 생성하기

타입은 TypeHandlerCallback인터페이스의 사용을 통해 iBATIS내에서 확장될수 있다. 당신 자신의 타입 핸들러를 생성하기 위해, TypeHandlerCallback을 구현한 클래스를 생성할 필요가 있다. 사용자정의 타입 핸들러를 사용하여 당신은 지원되지 않는 타입을 다루거나 지원되는 타입을 다른 방법으로 다뤄서 프레임워크를 확장할수 있다. 예를들면, 당신은 적절한 BLOB지원을 구현하는 사용자 정의 타입 핸들러를 사용하거나 대개 0/1대신에 "Y"와 "N"을 사용하는 boolean을 다루기 위해 이것을 사용할수 있다.

다음은 "Yes"와 "No"를 사용하는 boolean핸들러의 간단한 예제이다.

```
public class YesNoBoolTypeHandlerCallback implements TypeHandlerCallback {

  private static final String YES = "Y";
  private static final String NO = "N";

  public Object getResult(ResultGetter getter)
    throws SQLException {
    String s = getter.getString();
    if (YES.equalsIgnoreCase(s)) {
      return new Boolean (true);
    } else if (NO.equalsIgnoreCase(s)) {
```

```
      return new Boolean (false);
    } else {
      throw new SQLException (
        "Unexpected value " + s + " found where " + YES + " or " + NO + " was expected.");
    }
  }

  public void setParameter(ParameterSetter setter, Object parameter)
    throws SQLException {
    boolean b = ((Boolean)parameter).booleanValue();
    if (b) {
      setter.setString(YES);
    } else {
      setter.setString(NO);
    }
  }

  public Object valueOf(String s) {
    if (YES.equalsIgnoreCase(s)) {
      return new Boolean (true);
    } else {
      return new Boolean (false);
    }
  }
```

iBATIS내 사용하기 위한 이러한 타입을 선언하기 위해, 당신은 sqlMapConfig.xml내 다음의 문법을 사용한다.

```
<typeHandler
 javaType="boolean"
 jdbcType="VARCHAR"
 callback="org.apache.ibatis.sqlmap.extensions.YesNoBoolTypeHandlerCallback"/>
```

iBATIS 가 java타입과 jdbc타입간의 이전을 다루는것을 알고 난뒤에, 특정 타입 핸들러 콜백은 작성된다. 선택적으로, <result> 매핑에 타입 핸들러를 지정하거나 명시적인 인라인 파라미터 맵으로 개별적인 프라퍼티를 위한 타입 핸들러를 지정할수 있다.

캐싱된 매핑 구문 결과

매핑 구문 쿼리로부터의 결과는 statement 요소내의 cacheModel파라미터를 정의함으로써 간단하게 캐쉬될수 있다. 캐쉬 모델은 당신의 SQL Map내에서 정의된 설정된 캐쉬다. 캐쉬 모델은 다음처럼 cacheModel요소를 사용해서 설정된다.

```
<cacheModel id="product-cache" type ="LRU" readOnly="true" serialize="false">
  <flushInterval hours="24"/>
  <flushOnExecute statement="insertProduct"/>
  <flushOnExecute statement="updateProduct"/>
  <flushOnExecute statement="deleteProduct"/>
  <property name="cache-size" value="1000" />
</cacheModel>
```

위의 캐쉬 모델은 LRU(Least Recently Used) 방식을 사용해서 "product-cache" 라는 이름의 캐쉬 인스턴스를 생성할것이다. *type*속성값은 전체경로의 클래스명이거나 아래처럼 구현을 포함하는 것의 별칭이다. flush요소에 기초로 하여 캐쉬 모델 내에서 정의된다. 이 캐쉬는 24시간 마다 삭제된다. interval요소내에서 *hours, minutes, seconds* 또는 *milliseconds*단위로 설정이 되어서 삭제된다. 캐쉬는 추가적으로 *insertProduct*, *updateProduct*, 또는*deleteProduct* 매핑 구문가 수행될때마다 삭제된다. 캐쉬를 위해 "flush on execute"요소의 숫자값이 정의될수 있다. 몇몇 캐쉬 구현체은 위에서 보여지는 'cache-size' 같은 추가적인 프라퍼티를 필요로한다. LRU 캐쉬의 경우에 크기는 캐쉬내 저장되기 위한 항목의 갯수로 결정된다. 캐쉬 모델이 설정되었을 때 당신은 매핑 구문에 의해 사용되기 위한 캐쉬 모델을 정의할수 있다. 예를 들면

```
<select id="getProductList"  cacheModel="product-cache">
    select * from PRODUCT where PRD_CAT_ID = #value#
</select>
```

읽기전용 대 읽기/쓰기

프레임워크는 읽기전용과 읽기/쓰기 캐쉬를 모두 지원한다. 읽기전용 캐쉬는 모든 유저에 의해 공유되어서 좀더 큰 성능향상을 보여준다. 어쨌든 읽기전용 캐쉬로부터 읽어들인 객체는 변경할수 없다. 대신에 새로운 객체는 업데이트를 위해 데이터베이스(또는 읽기/쓰기 캐쉬)로부터 읽어야만 한다. 반면에 정정(retrieval)및 변경을 위한 객체를 사용할 경우에는 읽기/쓰기 캐쉬가 추천된다(이를 테면 필수이다). 읽기전용 캐쉬를 사용하기 위해서는 캐쉬 모델 요소에 *readOnly="true"* 를 셋팅하라. 기초설정값은 읽기전용(true)이다.

직렬화가능한 읽기/쓰기 캐쉬

당신이 동의한다면 서술된 것처럼 세션당 캐쉬는 전역 애플리케이션 성능에 자그마한 이익을 준다. 읽기/쓰기 캐쉬의 다른 타입은 전체애플리케이션이 직렬화가능한 읽기/쓰기 캐쉬라면 성능향상을 보여준다. 이 캐쉬는 각각의 세션에 캐쉬된 객체의 다른 인스턴스를 반환할것이다. 그러므로 각각의 세션은 안전하게 반환된 인스턴스를 변경할수 있다. 여기서 의미론적인 차이점을 알아보자면 당신은 언제나 캐쉬로부터 반환된 같은 인스턴스를 기대하겠지만 캐쉬내에서 당신은 다른 것을 얻게 될것이다. 또한 직렬화가능한 캐쉬로부터 저장된 모든 객체는 직렬화가능해야만 한다. 이것은 직렬화가능한 캐쉬로 조합된 늦은(lazy) 로딩기능을 사용하기에는 어려울것이라는 것을 의미한다. 왜냐하면 늦은(lazy) 프록시는 직렬화가능하지 않기 때문이다. 캐쉬의 조합을 해결하는 가장 좋은 방법은 늦은(lazy)로딩과 테이블 조인을 간단히 시도하는것이다. 직렬화가능한 캐쉬를 사용하기 위해서는 기초설정 캐쉬모델이 읽기전용이고 직렬화가능하지 않기 때문에 *readOnly="false"* 와 *serialize="true"* 로 셋팅하라. 읽기전용 캐쉬는 직렬화되지 않을것이다.

캐쉬 타입들

캐쉬 모델은 다른 타입의 캐쉬를 지원하기 위해서 플러그인형태의 프레임워크를 사용한다. 그 구현은 cacheModel요소의 *type*속성값내에 정의된다. 이 정의된 클래스 이름은 CacheController인터페이스의 구현이나 아래에서 논의되는 4가지 별칭중에 하나가 되어야만 한다. 게다가 설정 파라미터는 cacheModel내에 포함된 property요소를 통해 구현체로 전달될수 있다. 현재 배포판에는 4가지 구현체를 포함하고 있다. 그들은 다음과 같다.

"MEMORY" (com.ibatis.db.sqlmap.cache.memory.MemoryCacheController)

MEMORY 캐쉬는 캐쉬행위를 관리하기 위해서 참조타입을 사용한다. 그것은 가비지컬렉터(garbage collector)가 캐쉬내에 머물러 있는지 아닌지 효과적으로 결정한다. MEMORY캐쉬는 객체 재사용의 일정한 패턴이 없는 애플리케이션또는 메모리가 충분하지 않은 애플리케이션을 위한 좋은 선택이다.

MEMORY 구현은 다음처럼 설정된다.

```
<cacheModel id="product-cache" type="MEMORY">
        <flushInterval hours="24"/>
        <flushOnExecute statement="insertProduct"/>
        <flushOnExecute statement="updateProduct"/>
        <flushOnExecute statement="deleteProduct"/>
        <property name="reference-type" value="WEAK" />
</cacheModel>
```

단지 하나의 프라퍼티가 MEMORY캐쉬 구현에 의해 인식된다. 'reference-type' 라는 이름의 프라퍼티는 STRONG, SOFT 또는 WEAK의 값으로 셋팅되어야만 한다. 그 값들은 JVM내에 유효한 여러가지 메모리 참조타입에 대응된다.

다음의 테이블은 MEMORY캐쉬를 위해 사용될수 있는 다른 참조타입을 서술한다. 참조타입을 핵심을 좀더 이해하기 위해서는 "reachability" 에 대한 정보를 위한 java.lang.ref부분의 JDK문서를 보기를 바란다.

WEAK *(default)*	이 참조타입은 대부분의 경우에 가장 좋은 선택이고 참조타입을 정의하지 않는다면 디폴트로 설정되는 값이다. 이것은 대개의 결과에 성능을 향상시킬것이다. 하지만 할당된 다른 객체내에서 사용되는 메모리를 완전히 제거할것이다. 그 결과는 현재 사용중이지 않다는 것을 가정한다.
SOFT	이 참조타입은 결과물이 현재 사용중이지 않고 메모리가 다른 객체를 위해 필요한 경우에 메모리가 바닥나는 가능성을 제거할것이다. 어쨌든 이것은 할당되고 좀더 중요한 객체에 유효하지 않는 메모리에 대해 대부분 공격적인 참조타입이 아니다.

STRONG	이 참조타입은 명시적을 캐쉬가 삭제될때까지 메모리내에 저장된 결과물을 보증한다. 이것은 1) 매우 작음, 2) 절대적으로 정적, and 3) 매우 종종 사용되는 결과에 좋다. 장점은 특수한 쿼리를 위해 매우 좋은 성능을 보인다. 단점은 결과물에 의해 사용되는 메모리가 필요할 때 다른 객체를 위해 메모리를 반환하지 않는다.

"LRU" (com.ibatis.db.sqlmap.cache.lru.LruCacheController)

LRU 캐쉬는 객체가 자동으로 캐시로부터 어떻게 삭제되는지 결정하기 위해 Least Recently Used(가장최근에 적게 사용된) 알고리즘을 사용한다. 캐쉬가 가득 찼을 때 가장 최근에 접근된 객체는 캐쉬로부터 삭제된다. 이 방법은 종종 참조되는 특수한 객체가 있을 때 이것은 가장 최근에 삭제된 변경과 함께 캐쉬내에 남을것이다. LRU캐쉬는 오랜시간동안 하나이상의 사용자에게 특별한 객체가 사용되는 패턴을 가지는 애플리케이션을 위해서 좋은 선택이다(이를 테면 페이지 처리된 목록 사이에 앞 페이지 뒷 페이지를 탐색하는, 특수한 검색키.. 등등).

LRU 구현은 다음처럼 설정된다.

```
<cacheModel id="product-cache" type="LRU">
        <flushInterval hours="24"/>
        <flushOnExecute statement="insertProduct"/>
        <flushOnExecute statement="updateProduct"/>
        <flushOnExecute statement="deleteProduct"/>
        <property name="size" value="1000" />
</cacheModel>
```

단지 하나의 프라퍼티만이 LRU캐쉬에 의해 인식된다. 'size"라는 이름의 프라퍼티는 한번에 캐쉬내에 고정되는 객체의 최대갯수를 표현하는 숫자값으로 설정해야 한다. 여기서 기억해야할 중요한 것은 하나의 문자열 인스턴스로부터 자바빈즈의 ArrayList 해당되는 어떠한 객체로도 될수 있다는것이다. 그래서 메모리 포화의 위험이 있다면 당신의 캐쉬내 너무 많이 저장하지 마라.

"FIFO" (com.ibatis.db.sqlmap.cache.fifo.FifoCacheController)

FIFO 캐쉬는 객체가 캐쉬로부터 자동적으로 어떻게 삭제될지 결정하기 위해 First In First Out(먼저 들어온 것을 먼저 보낸다.) 알로리즘을 사용한다. 캐쉬가 가득찼을 때 가장 오래된 객체는 캐쉬로부터 삭제될것이다. FIFO캐쉬는 특수한 쿼리가 빠른 성공내에서 적은 수로 참조되는 패턴을 사용할 때 좋다. 하지만 나중에 몇몇시점에서는 가능하지 않다.

FIFO 구현은 다음처럼 설정된다.

```
<cacheModel id="product-cache" type="FIFO">
        <flushInterval hours="24"/>
        <flushOnExecute statement="insertProduct"/>
        <flushOnExecute statement="updateProduct"/>
        <flushOnExecute statement="deleteProduct"/>
        <property name="size" value="1000" />
</cacheModel>
```

단지 하나의 프라퍼티가 FIFO캐쉬에 의해 인식된다. . 'size"라는 이름의 프라퍼티는 한번에 캐쉬내에 고정되는 객체의 최대갯수를 표현하는 숫자값으로 설정해야 한다. 여기서 기억해야할 중요한 것은 하나의 문자열 인스턴스로부터 자바빈즈의 ArrayList 해당되는 어떠한 객체로도 될수 있다는것이다. 그래서 메모리 포화의 위험이 있다면 당신의 캐쉬내 너무 많이 저장하지 마라.

"OSCACHE" (com.ibatis.db.sqlmap.cache.oscache.OSCacheController)

OSCACHE 캐쉬는 OSCache 2.0 캐쉬엔진을 위한 플러그인이다. 이것은 설정가능하고 구분가능하고 유연성이 있다.

OSCACHE 구현은 다음처럼 설정된다.

```
<cacheModel id="product-cache" type="OSCACHE">
        <flushInterval hours="24"/>
        <flushOnExecute statement="insertProduct"/>
        <flushOnExecute statement="updateProduct"/>
        <flushOnExecute statement="deleteProduct"/>
</cacheModel>
```

OSCACHE 구현은 설정을 위해 어떠한 프라퍼티요소도 사용하지 않는다. 대신에 OSCache인스턴스는 클래스패스 가장 상위에 위치시켜야 하는 표준적인 *oscache.properties*을 사용해서 설정된다. 당신은 알고리즘, 캐쉬크기, 영속성접근법(메모리, 파일등등) 그리고 클러스터링을 설정할수 있다.

좀더 많은 정보를 위해서는 OSCache문서를 참조하라. OSCache와 그 문서는 다음의 Open Symphony 웹사이트에서 찾을수 있다.

http://www.opensymphony.com/oscache/

동적인 매핑 구문

JDBC를 사용해서 직접적으로 작동할 때 매우 공통적인 문제는 동적 SQL이다. 파라미터값 뿐만 아니라 파라미터와 칼럼이 모두 포함된 변경을 하는 SQL문에서 작업을 하는 것은 매우 어렵다. 전형적인 해결법은 조건적인 if-else문과 지겨운 문자열 연결 덩어리를 사용하는 것이다. 요구되는 결과는 종종 쿼리가 예제 객체와 유사한 객체를 찾기위해 빌드될수 있는 예제에 의해 쿼리된다. SQL Map API는 어떤 매핑 구문요소에 적용될수 있는 상대적으로 훌륭한 해결법을 제공한다. 이것은 간단한 예제이다.

```
<select id="dynamicGetAccountList"
                cacheModel="account-cache"
                resultMap="account-result" >

    select * from ACCOUNT

        <isGreaterThan prepend="and" property="id" compareValue="0">
            where  ACC_ID = #id#
        </isGreaterThan>

    order by ACC_LAST_NAME

</select>
```

위 에제에서 파라미터빈의 "id"프라퍼티의 상태에 의존해서 생성될수 있는 두가지 가능한 statement가 있다. 만약 파라미터가 0보다 크다면 statement는 다음처럼 생성될것이다.

```
  select * from ACCOUNT where ACC_ID = ?
```

또는 파라미터가 0이거나 더 작다면 statement는 다음처럼 보일것이다.

```
  select * from ACCOUNT
```

이것의 즉각적인 유용함은 좀더 복잡한 상황이 발생하기 전까지 명백하게 되지는 않을것이다. 예를 들면 다음은 좀더 복잡한 예제이다.

```
 <select id="dynamicGetAccountList"
                resultMap="account-result" >
  select * from ACCOUNT
  <dynamic prepend="WHERE">
    <isNotNull prepend="AND" property="firstName"
      open="(" close=")">
      ACC_FIRST_NAME = #firstName#
      <isNotNull prepend="OR" property="lastName">
        ACC_LAST_NAME = #lastName#
```

```
      </isNotNull>
    </isNotNull>
    <isNotNull prepend="AND" property="emailAddress">
      ACC_EMAIL like #emailAddress#
    </isNotNull>
    <isGreaterThan prepend="AND" property="id" compareValue="0">
      ACC_ID = #id#
    </isGreaterThan>
  </dynamic>
  order by ACC_LAST_NAME
</select>
```

상황에 의존적으로 위 동적 statement로부터 각각 다른 16가지의 SQL문이 생성될 수 있다. if-slse구조와 문자열연결을 코딩하는 것은 덩어리지고 수백라인의 로드를 요구할수 있다.

동적 statement를 사용하는 것은 당신의 SQL문에 몇몇 조건적인 태그를 추가하는것처럼 간단하다. 예를 들면

```
<select id="someName"
              resultMap="account-result" >
  select * from ACCOUNT
  <dynamic prepend="where">
    <isGreaterThan prepend="and" property="id" compareValue="0">
      ACC_ID = #id#
    </isGreaterThan>
    <isNotNull prepend=”and" property="lastName">
      ACC_LAST_NAME = #lastName#
    </isNotNull>
  </dynamic>
  order by ACC_LAST_NAME
 </select>
```

위의 statement에서 <dynamic>요소는 동적인 SQL부분을 구분한다. dynamic요소는 옵션이고 포함된 조건이 statement에 덧붙여지지 않는다면 “WHERE”과 같은 구문이 포함되지 않을 경우에 “WHERE”같은 구문을 관리하는 방법을 제공한다. statement부분은 statement내에 포함될 SQL코드를 포함할지 안 할지 결정하는 조건적 요소를 제한없이 포함할 수 있다. 조건적인 요소의 모든 것은 쿼리로 전달되는 파라미터 객체의 상태에 기초하여 작동한다. 동적인 요소와 조건적인 요소 모두 “prepend”속성을 가진다. prepend속성은 필요하다면 부모요소의 prepend에 의해 오버라이딩되기 위해 자유로운 코드의 일부이다. 위 예제에서 “where” 구문은 첫번째 true조건구문을 오버라이딩 할것이다. 이것은 SQL문이 정확하게 빌드되는 것을 확인하기 위해 필요하다. 예를 들면 첫번째 true조건의 경우에서 AND문이 필요없고 사실 statement이 끝날것이다. 다음의 섹션은 다양한 종류의 요소(바이너리 조건을 포함, 단일성분의 조건과 iterate)를 서술한다.

Dynamic 요소

dynamic태그는 다른 동적 sql요소를 포장하고 결과적인 내용물을 위해 prepend, open 또는 close를 수행하기 위한 제공되는 간단한 태그이다. 이 태그를 사용할때, removeFirstPrepend속성 기능이 강제로 수행된다.

이항연산 속성**:**
- prepend – statement에 붙을 오버라이딩 가능한 SQL부분(옵션)
- open – 결과적인 전체내용물을 열기위한 문자열(옵션)
- close – 결과적인 전체내용물을 닫기위한 문자열(옵션)

<dynamic>	Wrapper tag that allows for an overall prepend, open and close.

이항연산 요소

이항연산 요소는 정적값또는 다른 프라퍼티값을 위한 프라퍼티값과 비교한다. 만약 결과가 true라면 몸체부분은 SQL 쿼리에 포함된다.

이항연산 속성**:**
- prepend – statement에 붙을 오버라이딩 가능한 SQL부분(옵션)

property – 비교되는 프라퍼티(필수)
compareProperty – 비교되는 다른 프라퍼티(필수 또는 compareValue)
compareValue – 비교되는 값(필수 또는 compareProperty)
removeFirstPrepend – 첫번째로 내포된 내용을 생성하는 요소의 prepend를 제거(true|false, 선택)
open – 결과적인 전체내용물을 열기위한 문자열(선택)
close – 결과적인 전체내용물을 닫기위한 문자열(선택)

<isEqual>	프라퍼티와 값 또는 다른 프라퍼티가 같은지 체크.
<isNotEqual>	프라퍼티와 값 또는 다른 프라퍼티가 같지 않은지 체크.
<isGreaterThan>	프라퍼티가 값 또는 다른 프라퍼티보다 큰지 체크.
<isGreaterEqual>	프라퍼티가 값 또는 다른 프라퍼티보다 크거나 같은지 체크.
<isLessThan>	프라퍼티가 값 또는 다른 프라퍼티보다 작은지 체크.
<isLessEqual>	프라퍼티가 값 또는 다른 프라퍼티보다 작거나 같은지 체크. 사용법 예제: **<isLessEqual prepend="AND" property="age" compareValue="18"> ADOLESCENT = 'TRUE' </isLessEqual>**

단항연산 요소

단항연산 요소는 특수한 조건을 위해 프라퍼티의 상태를 체크한다.

단항연산 속성**:**
prepend – statement에 붙을 오버라이딩 가능한 SQL부분(옵션)
property – 체크되기 위한 프라퍼티(필수)
removeFirstPrepend – 태그를 생성하는 첫번째 내포 내용의 prepend제거(옵션)
open – 결과적인 전체내용물을 열기위한 문자열(옵션)
close – 결과적인 전체내용물을 닫기위한 문자열(옵션)

<isPropertyAvailable>	프라퍼티가 유효한지 체크(이를 테면 파라미터빈의 프라퍼티이다.)
<isNotPropertyAvailable>	프라퍼티가 유효하지 않은지 체크(이를 테면 파라미터의 프라퍼티가 아니다.)
<isNull>	프라퍼티가 null인지 체크
<isNotNull>	프라퍼티가 null이 아닌지 체크
<isEmpty>	Collection, 문자열 또는 String.valueOf() 프라퍼티가 null이거나 empty("" or size() < 1)인지 체크
<isNotEmpty>	Collection, 문자열 또는 String.valueOf() 프라퍼티가 null이 아니거나 empty("" or size() < 1)가 아닌지 체크. 사용 예제: **<isNotEmpty prepend="AND" property="firstName" > FIRST_NAME=#firstName# </isNotEmpty>**

다른 요소

Parameter Present: 파라미터 객체가 존재하는지 체크
Parameter Present Attributes:
prepend – statement에 붙을 오버라이딩 가능한 SQL부분(옵션)
removeFirstPrepend – 태그를 생성하는 첫번째 내포 내용의 prepend제거(옵션)
open – 결과적인 전체내용물을 열기위한 문자열(옵션)
close – 결과적인 전체내용물을 닫기위한 문자열(옵션)

<isParameterPresent>	파라미터 객체가 존재(not null)하는지 보기위해 체크.

<isNotParameterPresent>	파라미터 객체가 존재하지(null) 않는지 보기위해 체크. 사용 예제: **<isNotParameterPresent prepend="AND">** **EMPLOYEE_TYPE = 'DEFAULT'** **</isNotParameterPresent>**

Iterate: 이 태그는 collection을 반복하거나 리스트내 각각을 위해 몸체부분을 반복한다.
Iterate Attributes:
prepend – the statement에 붙을 오버라이딩 가능한 SQL부분(옵션)
property – 반복되기 위한 java.util.List타입의 프라퍼티(필수)
open – 반복의 전체를 열기 위한 문자열, 괄호를 위해 유용하다. (옵션)
close – 반복의 전체를 닫기 위한 문자열, 괄호를 위해 유용하다. (옵션)
conjunction – 각각의 반복 사이에 적용되기 위한 문자열, AND 그리고 OR을 위해 유용하다. (옵션)
removeFirstPrepend – 태그를 생성하는 첫번째 내포 내용의 prepend제거(옵션)

<iterate>	java.util.Collection 이나 java.util.Iterator 또는 배열 구현체인 프라퍼티에 대해 반복적으로 처리 사용법 예제: **<iterate prepend="AND" property="userNameList"** **open="(" close=")" conjunction="OR">** **username=#userNameList[]#** **</iterate>** collection은 매핑 구문에 파라미터로 전달할때 반복적으로 처리하는 것이 가능하다. 사용 예제: **<iterate prepend="AND" open="(" close=")" conjunction="OR">** **username=#[]#** **</iterate>** 주의: iterator요소를 사용할 때 리스트 프라퍼티의 끝에 중괄호[] 를 포함하는 것은 중요하다. 중괄호는 문자열처럼 리스트를 간단하게 출력함으로부터 파서를 유지하기 위해 리스트처럼 객체를 구별한다.

더 상세한 <iterate> 요소 사용 노트:

첫번째 예제에서, "userNameList[]"는 목록에서 현재 항목을 가져오는 작업이 된다. 다음처럼 목록형태의 항목에서 프라퍼티를 조회하도록 이 작업을 수행할수 있다.

```
<iterate prepend="AND" property="userList"
         open="(" close=")" conjunction="OR">

   firstname=#userList[].firstName#  and
   lastname=#userList[].lastName#

</iterate>
```

iBATIS 버전 2.2.0에서, iterate요소는 복잡한 조건을 생성하기 위해 내포될수 있다. 다음은 그 예제이다.

```
<dynamic prepend="where">
  <iterate property="orConditions" conjunction="or">
  (
    <iterate property="orConditions[].conditions"
          conjunction="and">
        $orConditions[].conditions[].condition$
        #orConditions[].conditions[].value#
    </iterate>
  )
  </iterate>
</dynamic>
```

이 예제는 파라미터 객체가 객체의 List인 "orConditions" 프라퍼티를 가진다고 가정한다. List안에 있는 각각의 객체는 "conditions"이라고 불리는 List프라퍼티를 포함한다. 그래서 파라미터 객체로 List안에 List를 가진다. "orConditions[].conditions[].condition" 라는 문구는 "외부 루프에서 현재 객체의 conditions프라퍼티인 내부 목록의 현재 요소에서 condition프라퍼티를 가져온다. "라는 것을 의미한다. iterate요소를 내포하는 레벨에 관련해서는 어떤 제약도 없다. 또한 "현재 항목" 이라는 작업은 다른 동적 요소에 대한 입력값처럼 사용될수 있다.
<iterate> 요소를 사용하는 *removeFirstPrepend* 함수는 다른 요소와는 조금 다르다. *removeFirstPrepend*를 true로 지정한다면, 내용을 생성하는 첫번째 내포된 속성은 자체의 prepend를 제거할것이다. 이러한 현상은 전체 루프에서 오직 한번만 발생할것이다. 이것은 대부분의 환경에서 올바른 현상이다.
몇가지 환경에서, *removeFirstPrepend* 함수가 한번보다는 루프에 각각의 반복작업마다 작동할지도 모른다. 이 경우, *removeFirstPrepend*를 위한 값처럼 iterate를 지정하라. 이 함수는 iBATIS 버전 2.2.0이나 그 이후 버전에서만 사용가능하다.

간단한 동적 **SQL**요소

위에서 논의된 전체 동적 매핑 구문 API의 힘에도 불구하고 때때로 당신은 당신의 SQL이 동적이기 위해 간단하고 작은 규모가 될 필요가 있다. 이것을 위해 SQL statement와 statement는 조항에 의한 순서로 동적으로 구현하도록 도와주는 동적 SQL요소를 포함할수 있다. 이 개념은 인라인 파라미터 maps처럼 잘 작동한다. 하지만 조금 다른 문법을 사용한다. 다음예제를 보라.

```
<select  id="getProduct" resultMap="get-product-result">
        select * from PRODUCT order by $preferredOrder$
</select>
```

위 예제에서 *preferredOrder* 동적 요소는 파라미터객체(파라미터 map처럼)의 *preferredOrder*프라퍼티값에 의해 대체될수 있다. 차이점은 이것이 SQL statement의 기초적인 변화이다. 이것은 간단하게 파라미터 값을 셋팅하는것보다 좀더 큰 일이다. 동적 SQL 요소내에서 만들어지는 실수는 보안, 성능과 안정위험을 소개할수 있다. 간소한 동적 SQL요소가 선호적으로 사용되는 것을 확인하기 위해서 너무 과다하게 체크되는 것을 주의해라. 당신 디자인의 꺼림은 데이터베이스가 당신의 비즈니스객체모델을 침해하는 것을 정의하기 위해 잠재적이다. 예를 들면 당신은 당신의 비즈니스 객체 프라퍼티나 또는 당신의 JSP페이지에 필드값처럼 마치기 위해서 조항에 의해서 정렬되기 위한 경향의 칼럼명을 원하지 않을지도 모른다.

간단한 동적 요소는 <statements>내에 포함될수 있고 스스로 SQL statement를 변경할 필요가 있을 때 직접 손으로 작업할수 있다. 예를 들면

```
<select  id="getProduct" resultMap="get-product-result">
        SELECT * FROM PRODUCT
        <dynamic prepend="WHERE">
           <isNotEmpty property="description">
              PRD_DESCRIPTION   $operator$   #description#
           </isNotEmpty>
        </dynamic>
</select>
```

위의 예제에서 파라미터객체의 *operator* 프라퍼티는 $operator$ 토큰으로 대체되기 위해 사용된다. 만약 *operator* 프라퍼티가 ;like'와 같고 *description* 프라퍼티가 '%dog%' 와 같다면 SQL문은 다음처럼 생성된다.

```
        SELECT * FROM PRODUCT WHERE PRD_DESCRIPTION LIKE '%dog%'
```

데이터 매퍼로 프로그래밍하기**: The API**

SqlMapClient API는 간단하고 최소한으로 되어 있다. 이것은 프로그래머에게 4가지(SQL Map를 설정하기, SQL update문(insert와 delete문을 포함해서) 수행하기, 하나의 객체를 위한 쿼리 수행하기, 그리고 객체의 리스트를 위한 쿼리 수행하기) 중요한 기능을 할 수 있는 능력을 제공한다.

설정

SQL Map을 설정하는 것은 당신이 SQL Map XML정의파일과 SQL Map설정파일을 생성하는것이다. SqlMapClient인스턴스는 SqlMapClientBuilder를 사용해서 빌드된다. 이 클래스는 buildSqlMap()이라는 하나의 중요한 정적 메소드를 가진다. buildSqlMap()메소드는 간단하게 sqlMap-config.xml의 내용을 읽을수 있는 Reader나 InputStream 인스턴스를 가져온다.

```
String resource = "com/ibatis/example/sqlMap-config.xml";
Reader reader = Resources.getResourceAsReader (resource);
SqlMapClient sqlMap =  SqlMapClientBuilder.buildSqlMap(reader);

String resource = "com/ibatis/example/sqlMap-config.xml";
InputStream inputStream = Resources.getResourceAsStream (resource);
SqlMapClient sqlMap =
SqlMapClientBuilder.buildSqlMap(inputStream);
```

The differences in these methods primarily relates to character encoding and internationalization issues. See the section on internationalization for more details.

트랜잭션

디폴트에 의하면 SqlMapClient인스턴스에서 어느 executeXxxx()메소드를 호출하는 것은 자동커밋/롤백(auto-commit/rollback)을 하는것이다. 이것은 executeXxxx()의 각각의 호출이 하나의 작업단위가 되는 것을 의미한다. 이것은 정말로 간단하다. 하지만 하나의 작업단위로 수행해야할 많은 수의 statement(이를 테면 그룹처럼 모두 성공하거나 실패하는것)를 가진다면 좋은생각이 아니다. 이것은 트랜잭션이 작동하는 시점이다.

만약 당신이 전역(SQL Map설정파일에 의해 설정된) 트랜잭션을 사용한다면 당신은 자동커밋을 사용할수 있고 여전히 작업단위 행위를 달성할수 있다. 어쨌든 이것은 트랜잭션 범위 경계는 성능원인에 영향을 끼친다. 그리고 이것은 connection pool과 데이터베이스 연결 초기화에 트래픽을 감소시킨다.

SqlMapClient인터페이스는 트랜잭션경계를 지정하기 위해 메소드를 가진다. 트랜잭션이 시작되고 SqlMapClient인터페이스의 다음과 같은 메소드를 사용함으로써 커밋되고/되거나 롤백된다.

```
public void startTransaction () throws SQLException
public void commitTransaction () throws SQLException
public void endTransaction () throws SQLException
```

트랜잭션을 시작함으로써 당신은 connection pool로부터 connection을 가져오고 SQL쿼리와 update를 가져오기 위해 열린다.

트랜잭션을 사용하는 예제는 다음과 같다.

```
private Reader reader = new Resources.getResourceAsReader(
                                    "com/ibatis/example/sqlMap-config.xml");
private SqlMapClient sqlMap = SqlMapClientBuilder.buildSqlMap(reader);

public updateItemDescription (String itemId, String newDescription)
     throws SQLException {
   try {
     sqlMap.startTransaction ();
     Item item =  (Item) sqlMap.queryForObject ("getItem", itemId);
     item.setDescription (newDescription);
     sqlMap.update ("updateItem", item);
     sqlMap.commitTransaction ();
   } finally {
     sqlMap.endTransaction ();
```

```
    }
}
```

endTransaction()가 에러에도 불구하고 호출되는 방법에 주의하라. 이것은 정화(cleanup)을 확실히 하기 위해서 중요한 단계이다. 그 규칙은 만약 당신이 startTransaction()을 호출한다면 endTransaction()을 받드시 호출해야 한다. (당신이 커밋을 하거나 하지않더라도)

주의**!** 트랜잭션은 내포될수 없다. 한 개이상의 같은 쓰레드로부터 .startTransaction()을 호출하는 것은 commit()나 rollback()을 호출하기 전에 던져질 예외를 야기한다. 반면에 각각의 쓰레드는 SqlMapClient 인스턴스마다 적어도 하나의 열려있는 트랜잭션 가질수 있다.

주의**!** SqlMapClient 트랜잭션은 저장 트랜잭션 객체를 위해 자바의 ThreadLocal을 사용한다. 이것은 각각의 startTransaction()을 호출하는 쓰레드가 그들의 트랜잭션을 위해 유일한 Connection객체를 가질것이라는걸 의미한다. 데이터소스로 connection을 반환(또는 connection을 닫는)하는 유일한 방법은 commitTransaction()이나 endTransaction()을 호출하는것이다. 아무것도 하지않으면 당신의 pool이 connection이 바닥나거나 잠기게 되는 현상을 야기할것이다.

자동 트랜잭션

명시적으로 트랜잭션을 사용하는 것이 매우 추천되는 사항임에도 불구하고 간단한 요구사항(대개는 읽기전용)을 위해 사용되는 간략화된 의미론이 있다. 만약 당신이 startTransaction(), commitTransaction() 그리고 endTransaction()메소드를 사용해서 명시적으로 트랜잭션의 경계를 지정하지 않는다면 그들은 위에서 보여진것처럼 트랜잭션범위의 밖에서 statement를 수행할때마다 자동으로 호출될것이다. 예를 들면

```
private Reader reader = new Resources.getResourceAsReader(
                                        "com/ibatis/example/sqlMap-config.xml");
private SqlMapClient sqlMap = SqlMapClientBuilder.buildSqlMap(reader);

public updateItemDescription (String itemId, String newDescription)
      throws SQLException {
   try {
      Item item =  (Item) sqlMap.queryForObject ("getItem", itemId);
      item.setDescription ("TX1");
      // No transaction demarcated, so transaction will be automatic (implied)
      sqlMap.update ("updateItem", item);
      item.setDescription (newDescription);
      item.setDescription ("TX2");
      // No transaction demarcated, so transaction will be automatic (implied)
      sqlMap.update("updateItem", item);
   } catch (SQLException e) {
      throw (SQLException) e.fillInStackTrace();
   }
}
```

주의**!** 자동 트랜잭션을 사용할 때 주의하라. 그들이 매력이 있음에도 불구하고 당신은 데이터베이스에 하나의 update보다 더 많은 작업량이 요구된다면 문제를 일으킬것이다. 위 예제에서 "updateItem"의 두번째 호출은 실패한다면 item description은 처음 새로운 "TX1"의 description으로 여전히 업데이트 될 것이다.(이를 테면 이것은 트랜잭션 행위가 아니다.)

전역**(**분산된**)** 트랜잭션

SQL Maps 프레임워크는 전역 트랜잭션을 잘 지원한다. 분산 트랜잭션처럼 알려져 있는 전역 트랜잭션은 당신에게 같은 작업단위에서 다중 데이터베이스(또는 JTA호환 자원) update를 가능하게 할 것이다.(이를 테면 다중 데이터베이스 update는 그룹처럼 성공하거나 실패할수 있다.)

외부적인/프로그래적인 전역 트랜잭션

당신은 외부적으로(-프로그래적(손에 의해 코딩되는)) 또는 공통적인 EJB처럼 다른 프레임워크를 구현함으로써 전역 트

랜잭션 관리를 선택할수 있다. EJB를 사용하면 당신은 EJB배치서술자에서 선언적인 트랜잭션 경계를 지정할수 있다. 하지만 이것은 이 문서의 범위를 벗어난다. 외부적또는 프로그래밍적 전역 트랜잭션을 지원하기 위해서 당신은 SQL Map 설정파일에서 <transactionManager> *type*속성을 "EXTERNAL"으로 설정해야 한다. 외부적으로 제어되는 전역 트랜잭션을 사용할 때 SQL Map트랜잭션 제어 메소드는 다소 불필요하다. 외부 트랜잭션 관리자에 의해 트랜잭션의 시작, 커밋, 롤백이 제어되기 때문이다. 어쨌든 SqlMapClient의 startTransaction(), commitTransaction() 그리고 endTransaction() 메소드를 사용해서 당신의 트랜잭션의 경계를 지정하는 것은 성능향상에 도움을 준다. 이런 메소드를 계속적으로 사용함으로써 당신은 일관적인 프로그래밍 패러다임을 유지할 것이다. Connection pool으로부터 connection을 위한 요청 횟수를 제거할 수 있을것이다. 다른 애플리케이션 서버와 트랜잭션 관리자는 다른 규칙을 사용한다. 반면에 간단한 고려사항은 전역 트랜잭션을 사용하기 위해 SQL Map코드를 요구되는 변경사항이 없다.

관리되는 전역 트랜잭션

SQL Map프레임워크는 당신을 위해 전역 트랜잭션를 관리할수 있다. 관리되는 전역 트랜잭션을 사용하기 위해서는 당신은 SQL Map설정파일에 <transactionManager> *type* 속성을 "JTA"로 설정해야만 하고 "UserTransaction"을 SqlMapClient인스턴스가 UserTransaction인스턴스를 찾는곳에 전체 JNDI이름에 셋팅해야 한다. 전체 설정방법에 대해서는 <transactionManager>부분을 보라.

전역 트랜잭션를 위한 프로그래밍은 크게 다르지 않다. 어쨌든 몇가지 작은 고려사항이 있다. 예제를 보라.

```
try {
  orderSqlMap.startTransaction();
  storeSqlMap.startTransaction();

  orderSqlMap.insertOrder(…);
  orderSqlMap.updateQuantity(…);

  storeSqlMap.commitTransaction();
  orderSqlMap.commitTransaction();
} finally {
  try {
    storeSqlMap.endTransaction()
  } finally {
    orderSqlMap.endTransaction()
  }
 }
```

이 예제에서 두 개의 다른 데이터베이스를 사용하는 것을 가정하는 두 개의 SqlMapClient인스턴스가 있다. 트랜잭션을 사용하기 위해 사용되는 첫번째 SqlMapClient(orderSqlMap)는 역시 전역 트랜잭션을 시작할 것이다. 그 후, 다른 모든 행위는 같은 SqlMapClient(orderSqlMap)이 commitTransaction()과 endTransaction()을 호출할때까지 전역 트랜잭션의 한부분처럼 간주된다. 전역 트랜잭션이 커밋되는 시점에 모든 작업이 승인된다.

경고*!* 이것들이 간단하게 모임에도 불구하고 당신이 전역트랜잭션을 지나치게 사용하지 않도록 하는 것은 중요하다. 당신의 애플리케이션서버와 데이터베이스 드라이버를 위해 추가적인 복잡한 설정이 요구되는 만큼 성능에 문제가 있을수 있다. 이것이 쉬워 보이지만 당신은 몇가지 어려움을 겪게 될것이다. EJB는 당신이 분산 트랜잭션을 요구하는 작업을 위해 세션 EJB를 사용을 좀더 쉽게 하기 위해서 좀더 많은 지원과 사용을 돕는 툴이 필요하다는 것을 기억해라. SQL Map전역 트랜잭션을 사용하는 JPetStore라는 예제 애플리케이션은 www.ibatis.com에서 찾을수 있을것이다.

다중 쓰레드 프로그래밍

iBATIS는 다중 쓰레드 프로그래밍을 지원하지만 몇가지 고려해야 할 사항이 있다.
첫번째로 고려해야할 사항은 트랜잭션이 쓰레드에서 전체적으로 포함되어야만 한다는 것이다. 다른 방법으로 설명하면, 트랜잭션은 쓰레드 경계에 대해 가로지를수 없다. 이러한 이유로, 전체 작업단위를 완료하기 위해 쓰레드를 시작하는 것이 좋은 생각이다. 각각의 작업 단위에 밀접한 쓰레드를 보장할수가 없다면 쓰레드 풀이 트랜잭션을 시작하고 완료하기 위해 기다리는 것은 좋은 생각이 아니다.
다른 고려사항은 각각의 쓰레드마다 오직 하나의 활성화된 트랜잭션을 가지도록 하는 것이다. 하나의 쓰레드에 하나 이상의 트랜잭션을 실행하도록 코드를 작성할수 있지만 트랜잭션은 순차적으로 되어야만 한다. 그리고 동시에 열려서는 안된다. 다음은 하나의 쓰레드에 여러개의 쓰레드가 존재하는 예제이다.

```
try {
```

```
  sqlMap.startTransaction();
  // execute statements for the first transaction
  sqlMap.commitTransaction();
} finally {
  sqlMap.endTransaction();
}

try {
  sqlMap.startTransaction();
  // execute statements for the second transaction
  sqlMap.commitTransaction();
} finally {
  sqlMap.endTransaction();
}
```

중요한 것은 쓰레드 내에서 한번에 오직 하나의 트랜잭션만이 활성화되어 있어야만 한다는 것이다. 물론, 구문별 자동 트랜잭션은 각각이 다른 트랜잭션이다.

iBATIS 클래스 로딩

(이 부분의 정보는 iBATIS 버전 2.2.0이나 그 이후 버전에서만 정확하다.)

iBATIS는 클래스를 로드하기 위해 *com.ibatis.common.resources.Resources* 클래스의 메소드를 사용한다. 클래스 로딩을 위해, 가장 중요한 메소드는 *classForName(String)* 이다. 이 메소드는 iBATIS의 클래스 로딩에서 가장 상위에 위치한다. 디폴트로 이 메소드는 다음과 같은 작업을 수행한다.

1. 현재 쓰레드의 컨텍스트 클래스 로더에서 클래스를 로드하도록 시도
2. 에러가 발생하면, *Class.forName(String)* 로 클래스를 호드하도록 시도

이 메소드는 대부분의 환경에서 잘 작동한다. 만약 몇가지 이유로, 이 메소드가 당신의 환경에서 작동하지 않는다면, *Resources.setDefaultClassLoader(ClassLoader)* 라는 정적 메소드를 호출하여 사용할 다른 클래스 로더를 지정할수 있다. 이 메소드 호출로 클래스 로더를 제공한다면, iBATIS는 지정된 클래스 로더로부터 모든 클래스를 로드하도록 시도할것이다. 사용자 정의 클래스 로더를 제공하고자 한다면, iBATIS의 다른 작업전에 메소드를 호출해야만 한다.

일괄처리

만약 당신이 수행할 많은 수의 쿼리아닌 statement(insert/update/delete)를 가진다면 당신은 추가적인 최적화를 위해서 네트워크 트래픽을 줄이고 JDBC드라이버를 허락하는 배치 같은 작업을 수행하길 원할지도 모른다. 배치를 사용하는 것은 SQL Map API를 사용하면 간단하다. 배치의 경계를 지정하기 위해서 두가지 간단한 메소드를 제공한다.

```
try {
  sqlMap.startTransaction();
  sqlMap.startBatch();
  // ... execute statements in between
  int rowsUpdated = sqlMap.executeBatch();  //optional
  sqlMap.commitTransaction();
} finally {
  sqlMap.endTransaction();
}
```

executeBatch()를 호출함으로써 모든 배치 statement는 JDBC드라이버를 통해 수행될것이다. 일괄처리가 시작된다면 커밋 작업이 일괄처리를 자동으로 실행할것이기 때문에 executeBatch() 를 호출하는 것은 선택적인 사항이다. 그래서 영향을 받는 레코드의 개수를 알고 싶다면 executeBatch()를 호출할수 있거나 건너뛰고 commitTransaction()를 호출한다. 일괄처리로 처리해야 할 작업이 매우 많다면, 일괄처리의 처음부터 끝까지 일정하게 커밋하도록 원할것이다. 예를 들어, 1000 레코드를 입력한다면, 큰 트랜잭션을 생성한 것을 유지하기 위해 100개의 레코드마다 커밋을 하고자 할것이다. 일정한 커밋을 하고자 한다면 각각의 일정한 커밋을 실행한 뒤에 startBatch()를 호출해야만 하는 것을 아는게 중요하다. 왜냐하면 커밋은 일괄처리를 실행하고 끝낼것이기 때문이다. 다음은 예제이다.

```
try {
```

```
  int totalRows = 0;
  sqlMap.startTransaction();

  sqlMap.startBatch();
  // … insert 100 rows
  totalRows += sqlMap.executeBatch();  //optional
  sqlMap.commitTransaction();

  sqlMap.startBatch();
  // … insert 100 rows
  totalRows += sqlMap.executeBatch();  //optional
  sqlMap.commitTransaction();

  sqlMap.startBatch();
  // … insert 100 rows
  totalRows += sqlMap.executeBatch();  //optional
  sqlMap.commitTransaction();

  // etc.

} finally {
  sqlMap.endTransaction();
}
```

일괄처리에 대한 중요한 노드:

1. 일괄처리는 명시적인 트랜잭션내에서 언제나 내포되어야만 한다. 명시적인 트랜잭션을 사용할수 없을때 일괄처리를 하지 않는다면 iBATIS는 각각의 구문을 개별적으로 실행할것이다.
2. 일괄처리 경계에서 매핑구문을 실행할지도 모른다. 다른 매핑 구문(이를테면, insert하고 update)을 실행한다면, iBATIS는 실행된 마지막 구문의 SQL문에 기초하여 "하위 일괄처리"로 처리되는 것을 막을것이다. 예를 들어, 다음의 코드를 보자.

   ```
   try {
     sqlMap.startTransaction();
     sqlMap.startBatch();

     sqlMap.insert("myInsert", parameterObject1);
     sqlMap.insert("myInsert", parameterObject2);
     sqlMap.insert("myInsert", parameterObject3);
     sqlMap.insert("myInsert", parameterObject4);

     sqlMap.update("myUpdate", parameterObject5);
     sqlMap.update("myUpdate", parameterObject6);

     sqlMap.insert("myInsert", parameterObject7);
     sqlMap.insert("myInsert", parameterObject8);
     sqlMap.insert("myInsert", parameterObject9);

     sqlMap.executeBatch();
     sqlMap.commitTransaction();
   } finally {
     sqlMap.endTransaction();
   }
   ```

   iBATIS는 이 일괄처리 작업을 3개의 하위 일괄처리로 나누어서 실행할것이다. 3개의 일괄처리 중 하나는 4개의 insert구문이고 다음은 2개의 update구문, 그리고 마지막은 3개의 insert구문이다. 비록 마지막 3개의 insert구문은 첫번째 4개와 같지만 update구문이 insert 구문들 사이에 있기 때문에 iBATIS는 다른 하위 일괄처리로 실행할것이다.

3. executeBatch() 메소드는 일괄처리로 수정된 레코드의 전체 개수인 int타입의 값을 반환한다. 하위 일괄처리가

있다면, iBATIS는 전체 개수에 하위 일괄처리로 처리된 레코드의 개수를 추가할것이다. JDBC드라이버가 일괄처리로 수정된 레코드의 전체 개수를 반환하는데 실패하는 경우 executeBatch() 메소드는 레코드들이 수정이 되더라도 0을 반환할것이다. 오라클 드라이버가 좋은 예제이다.

IBATIS 버전 2.2.0이나 그 이후 버전에서, 일괄처리를 실행하기 위해 *executeBatchDetailed* 라는 다른 메소드를 사용할 수 있다. 이 메소드는 executeBatch 메소드와 같지만, 레코드 개수에 대한 좀더 상세한 정보를 반환한다. *executeBatchDetailed* 메소드는 각각의 하위 일괄처리를 위한 BatchResult 객체의 List를 반환한다. 각각의 BatchResult객체는 하위 일괄처리가 실행될때 JDBC 드라이버가 반환하는 int[]처럼 하위 일괄처리와 관련된 구문에 대한 정보를 포함한다. java.sql.BatchUpdateException 가 발생한다면, 메소드는 하위 일괄처리가 성공한 이전의 BatchResult 객체의 List처럼 예외를 야기한 구문에 대한 정보를 포함하는 BatchException을 던질것이다.

SqlMapClient API를 통해 구문을 실행하기

SqlMapClient는 이것에 관련된 모든 매핑 구문를 수행하기 위한 API를 제공한다. 그 메소드들은 다음과 같다.

public Object insert(String statementName, Object parameterObject)
throws SQLException

public Object insert(String statementName) throws SQLException

public int update(String statementName, Object parameterObject)
throws SQLException

public int update(String statementName) throws SQLException

public int delete(String statementName, Object parameterObject)
throws SQLException

public int delete(String statementName) throws SQLException

public Object queryForObject(String statementName,
Object parameterObject)
throws SQLException

public Object queryForObject(String statementName) throws SQLException

public Object queryForObject(String statementName,
Object parameterObject, Object resultObject)
throws SQLException

public List queryForList(String statementName, Object parameterObject)
throws SQLException

public List queryForList(String statementName) throws SQLException

public List queryForList(String statementName, Object parameterObject,
int skipResults, int maxResults)
throws SQLException

public List queryForList(String statementName, int skipResults, int maxResults)
throws SQLException

void queryWithRowHandler (String statementName,
Object parameterObject, RowHandler rowHandler)
throws SQLException

void queryWithRowHandler (String statementName, RowHandler rowHandler)
throws SQLException

```
public PaginatedList queryForPaginatedList(String statementName,
        Object parameterObject, int pageSize)
        throws SQLException

public PaginatedList queryForPaginatedList(String statementName,
        int pageSize) throws SQLException

public Map queryForMap (String statementName, Object parameterObject,
        String keyProperty)
        throws SQLException

public Map queryForMap (String statementName, Object parameterObject,
        String keyProperty, String valueProperty)
        throws SQLException

public void flushDataCache()

public void flushDataCache(String cacheId)
```

각각의 경우에 매핑 구문의 이름은 첫번째 파라미터로 넘겨진다. 이 이름은 위에서 서술된 <statement>요소의 name 속성에 대응된다. 추가적으로 파라미터객체는 옵션적으로 전달할수 있다. null파라미터객체는 만약 기대되는 파라미터가 없다면 전달될수 있다. 행위의 남겨진 차이점은 아래에서 간단하게 설명된다.

insert(), update(), delete() : 이 메소드들은 update statement를 위해 특별히 의미된다. 밑의 쿼리 메소드중에 하나를 사용해서 update statement를 수행하는 것은 불가능하다. 어쨌든 이것은 애매한 의미이고 드라이버에 의존적이다. executeUpdate()의 경우에 statement는 간단하게 수행되고 영향을 받는 많은 수의 row가 반환된다.

queryForObject() : executeQueryForObject()의 두가지 버전이 있다. 하나는 새롭게 할당된 객체를 반환하는 것이고 다른 하나는 파라미터처럼 전달된 미리할당된 객체를 사용하는것이다. 후자의 경우 하나의 statement보다 많은 수에 의해 생성되는 객체에 유용하다.

queryForList() : queryForList()에는 네가지 버전이 있다. 첫번째는 쿼리를 실행하고 쿼리로부터 모든 결과를 반환한다. 두번째는 첫번째와 같지만 파라미터 객체를 받지 않는다. 세번째는 스킵되는 결과물의 수(이를 테면 시작지점)를 지정할수 있고 반환되는 레코드의 최대갯수도 지정할수 있다. 이것은 전체데이터를 반환하고 싶지 않은 굉장히 큰 데이터셋과 작업을 할 때 가치가 있다. 네번째는 세번째와 같지만 파라미터 객체를 받지 않는다.

queryWithRowHandler(): 이 메소드는 대개의 칼럼과 rows보다 result객체를 사용해서 row에 의해 결과를 처리하도록 한다. 이 메소드는 전형적인 이름과 파라미터 객체를 넘기지만 RowHandler를 가진다. row핸들러는 RowHandler인터페이스를 구현하는 클래스의 인스턴스이다. RowHandler인터페이스는 다음처럼 오직 하나의 메소드만 가진다.

```
public void handleRow (Object valueObject);
```

이 메소드는 데이터베이스로부터 반환되는 각각의 row를 위한 RowHandler에서 호출될것이다. 이것은 쿼리결과를 처리하기 위해 깔끔하고 간단하며 확장가능한 방법이다. RowHandler의 사용법 에제를 위해 아래 섹션의 예제를 보라. 리스트 파라미터는 queryForList()메소드로부터 반환될 List인터페이스의 인스턴스이다. 당신은 리스트의 result객체의 아무것도, 몇몇 또는 모든 것을 추가할수도 있다. 만약 당신이 100만개의 row로 작업을 한다면 리스트에 그것들을 모두 넣는 것은 좋은 생각이 아니다.

queryForPaginatedList(): 이것은 이전, 다음 버튼으로 데이터를 탐색할 때 데이터의 일부를 관리할수 있는 리스트를 반환하는 매우 유용한 메소드이다. 이것은 쿼리로부터 반환된 레코드의 일부만을 표시하는 사용자 인터페이스를 구현하는데 유용하다. 10,000개의 필드를 반환하는 검색엔진이 있지만 한번에 100개만 표시해야 하는 예제가 있다. PaginatedList인터페이스는 페이지를 통한 탐색(nextPage(), previousPage(), gotoPage())과 페이지의 상태를 체크(isFirstPage(), isMiddlePage(), isLastPage(), isNextPageAvailable(), isPreviousPageAvailable(), getPageIndex(), getPageSize())하는 메소드를 포함한다. 유효한 레코드의 총 개수가 PaginatedList인터페이스로부터 접근가능하지 않더라도 이것은 기대되는 결과갯수를 세는 두번째 statement를 간단히 수행함으로써 쉽게 달성할 수 있을 것이다. 반면에 PaginatedList를 사용하면 지나친 부하가 발생할수도 있다.

queryForMap(): 이 메소드는 리스트로 결과의 collection을 로드하는 대안을 제공한다. 대신에 이것은 keyProperty처럼 전달된 파라미터에 의해 결과를 키(key)화된 map으로 로드한다. 예를 들면 만약 Employee객체의 collection을 로드

한다면 당신은 그것들을 employeeNumber프라퍼티에 의해 키(key)화된 map으로 로드할것이다. Map의 값은 전체 employee객체가 될수도 있고 valueProperty라고 불리는 두번째 파라미터내 정의된 employee개체로 부터의 다른 프라퍼티가 될수도 있다. 예를들면 당신은 employee숫자에 의해 키(key)화된 employee이름의 map을 원할지도 모른다. result객체처럼 Map타입을 사용하는 개념을 사용하는 메소드를 혼란스러워하지 마라. 이 메소드는 result객체가 자바빈즈나 Map인지에따라 사용될수 있다.

flushDataCache(): 이 메소드는 데이터 캐시를 버리는 프로그램을 사용한 방식을 제공한다. 인자 없는 메소드는 모든 데이터 캐시를 제거할것이다. 인자로 cacheId를 가지는 메소드는 명명된 데이터 캐시를 제거할것이다. 후자의 경우를 위해, 당신은 명명공간을 사용하여 cachaId를 명시할 필요가 있다. (비록 당신이 **useStatementNamespaces** 를 **false** 로 셋팅하더라도).

Example 1: Executing Update (insert, update, delete)

```
sqlMap.startTransaction();

Product product = new Product();
product.setId (1);
product.setDescription ("Shih Tzu");

Integer primKey = (Integer)sqlMap.insert ("insertProduct", product);

sqlMap.commitTransaction();
```

Example 2: Executing Query for Object (select)

```
sqlMap.startTransaction();

Integer key = new Integer (1);

Product product = (Product)sqlMap.queryForObject ("getProduct", key);

sqlMap.commitTransaction();
```

Example 3: Executing Query for Object (select) With Preallocated Result Object

```
sqlMap.startTransaction();

Customer customer = new Customer();

sqlMap.queryForObject("getCust", parameterObject, customer);
sqlMap.queryForObject("getAddr", parameterObject, customer);

sqlMap.commitTransaction();
```

Example 4: Executing Query for List (select)

```
sqlMap.startTransaction();

List list = sqlMap.queryForList ("getProductList");

sqlMap.commitTransaction();
```

Example 5: Auto-commit

```
// When startTransaction is not called, the statements will
// auto-commit.  Calling commit/rollback is not needed.
Integer primKey = (Integer)sqlMap.insert ("insertProduct", product);
```

Example 6: Executing Query for List (select) With Result Boundaries

```
sqlMap.startTransaction();

List list = sqlMap.queryForList ("getProductList", 0, 40);

sqlMap.commitTransaction();
```

Example 7: Executing Query with a RowHandler (select)

```
public class MyRowHandler implements RowHandler {
  private SqlMapClient sqlMap;

  public MyRowHandler(SqlMapClient sqlMap) {
    this.sqlMap = sqlMap;
  }

  public void handleRow (Object valueObject)
                          throws SQLException {
    Product product = (Product) valueObject;
    product.setQuantity (10000);
    sqlMap.update ("updateProduct", product);
  }

}

sqlMap.startTransaction();

RowHandler rowHandler = new MyRowHandler(sqlMap);
sqlMap.queryWithRowHandler ("getProductList", rowHandler);

sqlMap.commitTransaction();
```

Example 8: Executing Query for Paginated List (select)

```
PaginatedList list =
        sqlMap.queryForPaginatedList ("getProductList", 10);

list.nextPage();
list.previousPage();
```

Example 9: Executing Query for Map

```
sqlMap.startTransaction();

Map map = sqlMap.queryForMap ("getProductList", null, "productCode");

sqlMap.commitTransaction();
```

```
Product p = (Product) map.get("EST-93");
```

SqlMap로깅하기

SqlMap 프레임워크는 내부적으로 log factory를 사용하여 로깅 정보를 제공한다. 내부적인 로그 팩토리는 다음의 로그 구현체중 하나로 로깅 정보를 위임할것이다.

1. Jakarta Commons Logging
2. Log4J
3. JDK 로깅 (JRE 1.4 나 그 이상이 요구됨)

내부적으로 iBATIS 로그 팩토리가 런타임시 체크한 사항에 기초하여 로깅 솔루션을 선택된다. IBATIS 로그 팩토리는 처음으로 찾아진(위 3가지 구현체중 순서대로 체크해서) 로깅 구현체를 사용할것이다. 위 구현체중 하나도 찾지 못하면 로깅은 사용되지 않는다.
많은 환경들이 애플리케이션 서버의 클래스패스의 일부로 JCL을 다룬다. 그런 환경에 대해 아는 것이 중요하고 iBATIS는 로깅 구현체로 JCL을 사용할것이다. 웹스피어와 같은 환경에서 웹스피어가 JCL의 자체적인 구현체를 제공하기 때문에 Log4J설정은 무시될것이다. iBATIS가 Log4J설정을 무시하기 때문에 망가뜨릴수도 있다. (사실, iBATIS는 JCL을 사용하기 때문에 Log4J 설정을 무시하는 것이다.)
JCL이 클래스패스에 포함된 환경에서 애플리케이션이 구동중이지만 다른 로깅 구현체를 사용할것이라면 다음 메소드(이 메소드는 iBATIS 버전 2.2.0이나 그 이후 버전에서 사용가능하다.)를 호출하여 다른 로깅 구현체를 선택할수 있다.

```
com.ibatis.common.logging.LogFactory.selectLog4JLogging();
com.ibatis.common.logging.LogFactory.selectJavaLogging();
```

이 메소드중에 하나를 호출한다면, 다른 iBATIS 메소드를 호출하기 전에 해야만 한다. 또한 이러한 메소드는 요청된 로그 구현체를 전환하기만 할것이다. 예를 들어, Log4J로깅을 선택하지만 Log4J가 런타임시 사용가능하지 않다면, iBATIS는 Log4J를 무시할것이고 로깅 구현체를 찾기 위해 일반적인 알고리즘을 사용할것이다.
Jakarta Commons Logging의 스펙은 Log4J와 JDK1.4로깅 API는 이 문서의 범위를 넘어선다. 어쨌든 아래 설정된 예제로 당신은 시작할수 있을것이다. 만약 이 프레임워크에 대해서 좀더 상세한 정보를 알기를 원한다면 당신은 다음 위치에서 좀더 다양한 정보를 얻을수 있을것이다.

Jakarta Commons Logging
- http://jakarta.apache.org/commons/logging/index.html

Log4J
- http://jakarta.apache.org/log4j/docs/index.html

JDK 1.4 로깅 API
- http://java.sun.com/j2se/1.4.1/docs/guide/util/logging/

Log 설정

iBATIS는 iBATIS패키지의 클래스가 아닌 다른 클래스도 모두 로깅한다. iBATIS 로깅 구문을 보기 위해, java.sql 패키지의 클래스에 대해서 로깅을 활성화해야만 한다. 다음 클래스에 대해 활성화해야만 한다.

- java.sql.Connection
- java.sql.PreparedStatement
- java.sql.Resultset
- java.sql.Statement

로깅 구현체를 사용할때 의존성은 어떻게 처리해야 하는가.? 우리는 Log4J를 사용하는 방법을 보여줄것이다.

commons 로깅 서비스의 설정은 하나이상의 추가적인 설정파일(이를 테면 log4j.properties)과 새로운 JAR파일(이를 테면 log4j.jar)을 포함하는 방법으로써 매우 간단하다. 다음의 예제설정은 제공자처럼 Log4J를 사용하여 전체 로깅서비스를 설정할것이다. 여기엔 두가지 단계가 있다.

단계 **1: Log4J JAR** 파일 추가하기**.**

우리는 Log4J를 사용하기 때문에 우리는 우리의 애플리케이션에 JAR파일이 유효한지 확인할 필요가 있다. 기억해라 Commons Logging은 추상 API이다. 이것은 이것의 구현체을 제공한다는 것을 의미하지 않는다. 그래서 Log4J를 사용하기 위해서는 당신은 JAR파일을 당신의 애플리케이션 클래스패스에 추가할 필요가 있다. 당신은 위 URL로부터 Log4J를 다운로드 받거나 iBATIS프레임워크와 함께 포함된 JAR을 사용할수도 있다. 웹또는 기업용 애플리케이션을 위해 당신은 WEB-INF/lib밑에 log4j.jar를 추가할수 있다. 또는 standalone애플리케이션을 위해 JVM클래스패스 시작 파라미터에 간단히 추가할수도 있다.

단계 **2: Log4J** 설정하기

Log4J설정은 간단하다. Commons Logging처럼 당신은 클래스패스 루트에 properties파일을 추가할것이다. 이 파일이 log4j.properties이고 다음처럼 보일것이다.

log4j.properties

```
 1  # Global logging configuration
 2  log4j.rootLogger=ERROR, stdout
 3
 4  # SqlMap logging configuration...
 5  #log4j.logger.com.ibatis=DEBUG
 6  #log4j.logger.com.ibatis.common.jdbc.SimpleDataSource=DEBUG
 7  #log4j.logger.com.ibatis.sqlmap.engine.cache.CacheModel=DEBUG
 8  #log4j.logger.com.ibatis.sqlmap.engine.impl.SqlMapClientImpl=DEBUG
 9  #log4j.logger.com.ibatis.sqlmap.engine.builder.xml.SqlMapParser=DEBUG
10  #log4j.logger.com.ibatis.common.util.StopWatch=DEBUG
11  #log4j.logger.java.sql.Connection=DEBUG
12  #log4j.logger.java.sql.Statement=DEBUG
13  #log4j.logger.java.sql.PreparedStatement=DEBUG
14  #log4j.logger.java.sql.ResultSet=DEBUG
15
16  # Console output...
17  log4j.appender.stdout=org.apache.log4j.ConsoleAppender
18  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
19  log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
```

위 파일은 에러일때만 리포팅을 로깅하는 최소한의 설정이다. 2번째라인에서 log4j가 errors만을 stdout appender에 리포팅하는 설정을 보여준다. Appender은 출력(이를 테면 console, file, database 등등)을 모으는 컴포넌트이다. 리포팅레벨을 최대화 하기 위해서는 우리는 다음처럼 2번째라인을 변경할수 있다.

```
log4j.rootLogger=DEBUG, stdout
```

2번째라인을 위처럼 바꿈으로써 Log4J는 지금 'stdout'(console) appender로 모든 로깅 이벤트를 리포팅한다. 만약 당신이 finer레벨로 로깅을 원한다면 당신은 위의 'SqlMap logging configuration'섹션(5에서 12번째 라인)을 사용해서 시스템에 대한 로그를 각각의 클래스에 설정할수 있다. 우리가 DEBUG레벨로 콘솔에 PreparedStatement행위를 로그하기를 원한다면 우리는 다음처럼 11번째라인을 간단하게 변경할것이다.

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

log4j.properties파일내의 남은 설정은 이 문서의 범위를 벋어나는것이지만 appender를 설정하는것이다. 어쨌든 당신은 Log4J웹사이트에서 추가적인 정보를 찾을수 있다.

한 페이지의 자바빈즈 과정

SqlMaps 프레임워크는 자바빈즈의 확실한 이해를 요구한다. SqlMap에 관련된만큼 자바빈즈 API가 충분하지는 않다. 그래서 여기서 당신이 전에 그것들을 경험해보지 못했다면 자바빈즈에 대해서 빠르게 소개를 하겠다.

자바빈이 무엇이냐.? 자바빈은 접근하는 명명메소드를 위한 엄격한 관습을 고수하고 클래스의 상태를 변화시키는 클래스이다. 이것을 말하는 다른 방법은 자바빈즈가 "getting"과 "setting" 프라퍼티를 위한 어떤 관습을 따른다는것이다. 자바빈즈의 프라퍼티는 메소드정의에 의해(필드에 의해서가 아닌) 정의된다. "set"으로 시작되는 메소드는 쓰기가능한 프라퍼

티(이를 테면 setEngine처럼)이다. "get"으로 시작되는 메소드는 읽기가능한 프라퍼티(이를 테면 getEngine)이다. Boolean프라퍼티를 위해 읽기가능한 프라퍼티 메소드는 "is"로 시작(이를 테면 isEngineRunning)한다. Set메소드는 반환타입을 정의하지 않는다. 그리고 프라퍼티를 위한 선호하는 타입(이를 테면 String)의 파라미터하나만을 가질뿐이다. Get메소드는 선호하는 타입을 반환해야 한다. 그리고 파라미터를 가지지 않을수도 있다. 같음에도 불구하고 자바빈즈는 Serializable인터페이스를 구현한다. 자바빈즈는 다른 기능을 지원하고 인자없는 생성자를 가져야만 한다. 하지만 SQL Map의 컨텍스트내에서 중요하지 않고 웹애플리케이션 컨텍스트내에서 중요하지 않다.

자바빈즈의 예제이다.

```
public class Product implements Serializable {

  private String id;
  private String description;

  public String getId() {
     return id;
  }
  public void setId(String id) {
     this.id = id;
  }

  public String getDescription() {
     return description;
  }
  public void setDescription(String description) {
     this.description = description;
  }
}
```

주의**!** 주어진 프라퍼티를 위해 ***get***과 ***set***프라퍼티의 데이터타입을 섞지 말라**.** 예를 들면 ***"account"***의 숫자형 프라퍼티를 위해 다음처럼 ***getter***와 ***setter***양속을 위한 같은숫자타입을 사용해야 한다**.**

```
public void setAccount (int acct) {….}
public int getAccount () {….}
```

"int"타입을 모두 사용할때는 주의하라. Get메소드로부터 "long"의 반환은 문제를 야기하게 될것이다.

주의**!** 유사하게도 당신은 getXxxx() 그리고 setXxxx()라는 이름의 메소드를 단지 하나만 가지도록 확인해라. 갖가지 형태의 메소드를 가지도록 하라. 당신은 좀더 특별하게 그것들의 명명을 할수 있을것이다.

주의**!** 대안적인 getter문법은 boolean타입의 프라퍼티를 위해 존재한다. Get메소드는 isXxxxx()라는 형태의 명명이 될것이다. "is"또는 "get"메소드를 가지도록 하라.

축하한다 당신은 이 과정을 통과했다.

Okay, 두번째 페이지

Side Bar: 객체 도식 탐색 **(**자바빈즈 프라퍼티**, Map, List)**

이 문서를 통해 당신은 객체가 struts또는 다른 어떤 자바빈즈 호환 프레임워크를 사용하는 사람들에게 친숙한 특별한 문법을 통해 접근할수 있다는 것을 보았다. SqlMap프레임워크는 자바빈즈 프라퍼티, map(key/value) 그리고 리스트를 통해서 탐색될 객체그래프를 허락한다. 다음의 탐색을 보라.

```
Employee emp = getSomeEmployeeFromSomewhere();
((Address) ( (Map)emp.getDepartmentList().get(3) ).get ("address")).getCity();
```

Employee객체의 프라퍼티는 다음처럼 SqlMapClient프라퍼티(ResultMap, ParameterMap 등등)내에서 탐색 될수 있다.

"departmentList[3].address.city"

중요: 이 문법은 iBATIS가 동적 SQL요소를 지원하기 위해 사용된 프라퍼티에만 적용한다. 이것은 <result> 나 <parameter> 매핑의 프라퍼티와는 작동하지 않을것이다.

Resources (com.ibatis.common.resources.*)

Resources클래스는 클래스패스로부터 자원을 로드하는 것을 쉽게 만드는 메소드를 제공한다. ClassLoaders를 다루는 것은 애플리케이션서버/컨테이너내에서 특별히 변경될수 있다. 그 Resource클래스는 때때로 지루한 작업과 함께 간단하게 시도될수 있다.

자원파일의 공통적인 사용은 :

- 클래스패스로부터 SQL Map 설정파일(이를 테면 sqlMap-config.xml) 로드하기.
- 클래스패스로부터 다양한 *.properties파일 로드하기
- 기타

자원을 로드하기 위해 많은 다른 방법이 있다.

- Reader 처럼 : 간단한 읽기전용 텍스트 데이터를 위해
- Stream 처럼 : 간단한 읽기전용 바이너리또는 텍스트 데이터를 위해
- File 처럼 : 읽기/쓰기 바이너리 또는 텍스트 파일을 위해
- Properties 파일처럼 : 읽기전용 설정 프라퍼티파일을 위해

위 스키마중에 하나를 사용해서 자원을 로드하는 Resource클래스의 다양한 메소드는 다음과 같다(위 스키마순서대로).

```
Reader getResourceAsReader(String resource);
InputStream getResourceAsStream(String resource);
File getResourceAsFile(String resource);
Properties getResourceAsProperties(String resource);
```

각각의 경우에 자원을 로드하기 위해 사용되는 ClassLoader는 Resource클래스를 로드하는것처럼 같게 될것이다. 또는 실패하였을 때 시스템 클래스로더가 사용될것이다. 이 이벤트에서 당신은 ClassLoader가 다루기 힘든(이를 테면 어떤 애플리케이션 서버내에서) 환경에서 사용하기 위한 ClassLoader(이를 테면 당신의 애플리케이션 클래스중에 하나로부터 ClassLoader를 사용하는)를 정의할 수 있다. 위의 각각의 메소드는 첫번째 파라미터처럼 ClassLoader를 가지는 자매(sister) 메소드를 가진다. 그들은 다음과 같다.

```
Reader getResourceAsReader (ClassLoader classLoader, String resource);
Stream getResourceAsStream (ClassLoader classLoader, String resource);
File getResourceAsFile (ClassLoader classLoader, String resource);
Properties getResourceAsProperties (ClassLoader classLoader, String resource);
```

resource 파라미터에 의해 명명된 자원은 전체 패키지명에 더해서 전체 파일명/자원명 이어야만 한다. 예를 들면 당신이 만약 'com.domain.mypackage.MyPropertiesFile.properties' 과 같은 클래스패스의 자원을 가진다면 당신은 다음 코드를 사용해서 Resource클래스를 사용하여 프라퍼티 파일을 로드 할 수 있다.(여기서 자원은 "/" 로 시작하지 않는다는 것에 조심하라.)

```
String resource = "com/domain/mypackage/MyPropertiesFile.properties";
Properties props = Resources.getResourceAsProperties (resource);
```

유사하게도 당신은 Reader처럼 클래스패스로부터 SqlMap설정파일을 로드할수 있다. 이것은 우리의 클래스패스에서의 간단한 프라퍼티 패키지이다(properties.sqlMap-config.xml).

```
String resource = "properties/sqlMap-config.xml";
Reader reader = Resources.getResourceAsReader(resource);
SqlMapClient sqlMap = SqlMapClientBuilder.buildSqlMap(reader);
```

Resources 국제화

노트: 이 부분의 정보는 iBATIS 2.3 이나 그 이후 버전을 위한 것이다.

iBATIS에서, 국제화에 관련한 중요 부분은 XML설정파일이다. 파일이 예외적인 인코딩을 사용하거나 시스템 디폴트 인코딩이 XML파일의 인코딩과 일치하지 않다면, 에러가 발생할수 있다. iBATIS는 이러한 이슈를 위해 두가지 방법을 제공한다.

Character Readers를 사용한 국제화

Reader를 사용할때, iBATIS는 파일을 인코딩하기 위해 InputStreamReader를 사용할것이다. 디폴트로 이 클래스는 시스템의 디폴트 인코딩을 사용하고 파일의 실제 인코딩은 무시한다. 몇몇 환경에서, 시스템의 디폴트 인코딩은 XML파일에 유리한 유니코드 인코딩과는 잘 작동하지 않는다. Reader로 iBATIS XML파일을 파싱할때 인코딩 이슈가 발생한다면, XML파일의 인코딩에 일칠시키기 위해 디폴트 인코딩을 변경할수 있다. 예를 들면:

```
String resource = “properties/sqlMap-config.xml”;
Resources.setCharset(Charset.forName('UTF-8”)); // change the default encoding
Reader reader = Resources.getResourceAsReader(resource);
SqlMapClient sqlMap = SqlMapClientBuilder.buildSqlMap(reader);
```

“setCharset” 메소드는 이후 “getResourceAsReader”를 호출하기 위해 사용되는 인코딩을 변경할것이다. 시스템 디폴트 인코딩으로 다시 돌아가고자 한다면, “setCharset(null)”를 호출해주면 된다.

Byte Input Streams을 사용한 국제화

XML설정파일을 읽기 위해 byte InputStream사용한다면, 종종 파서는 파일 인코딩을 자동으로 알아낼수 있을것이다. 그래서 종종 이 두개의 메소드를 사용하는것이 가장 좋은 선택이다. 이 메소드를 사용하는 예제는 다음과 같다.

```
String resource = “properties/sqlMap-config.xml”;
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlMapClient sqlMap = SqlMapClientBuilder.buildSqlMap(inputStream);
```

이 메소드는 문자열 인코딩을 위한 파서의 네이티브 지원에 의존한다. 이 메소드로 에러를 본적이 있다면, 디폴트 스키마를 무시하는 방법에 대한 정보를 가진 파서의 문서를 보라.

SimpleDataSource (com.ibatis.common.jdbc.*)

SimpleDataSource 클래스는 JDBC 2.0호환 DataSource의 간단한 구현체이다. 이것은 connection pooling기능의 편리한 세트를 제공하고 매우 가볍고 이식가능한 connection pooling솔루션과 완벽하게 동기화된다. SimpleDataSource는 다른 JDBC DataSource구현체처럼 사용되고 다음 URL에서 찾을 수 있는 JDBC표준확장API(JDBC Standard Extensions API)의 부분처럼 문서화되었다.

http://java.sun.com/products/jdbc/jdbc20.stdext.javadoc/

주의***!:*** JDBC 2.0 API는 표준 J2SE1.4.x의 부분처럼 포함되어 있다.

주의***!:*** SimpleDataSource는 매우 편리하고 효율적이며 실제적이다. 어쨌든 큰 기업용이나 특별한 상황에 치명적인 애플리케이션를 위해서 이것은 기업용 수준의 DataSource구현(애플리케이션 서버와 상업적인 O/R매핑 툴과 함께)을 사용하는경우에는 추천된다.

SimpleDataSource 생성자는 설정프라퍼티의 개수를 가져오는 프라퍼티 파라미터를 요구한다. 다음의 테이블은 프라퍼티의 이름과 서술이다. 단지 “JDBC.” 프라퍼티만 요구된다.

Property Name	Required	Default	Description
JDBC.Driver	Yes	n/a	JDBC 드라이버 클래스명
JDBC.ConnectionURL	Yes	n/a	JDBC connection URL.
JDBC.Username	Yes	n/a	데이터베이스에 로그인하기 위한 유저명
JDBC.Password	Yes	n/a	데이터베이스에 로그인하기 위한 패스워드
JDBC.DefaultAutoCommit	No	드라이버에 의존	Pool에 의해 생성된 모든 connection을 위한 autocommit셋팅

Pool.MaximumActiveConnections	No	10	주어진 시간에 열릴수 있는 connection의 최대수
Pool.MaximumIdleConnections	No	5	pool내에 저장될 idle connection의 수
Pool.MaximumCheckoutTime	No	20000	Connection이 “checked out”될수 있는 시간의 최대길이
Pool.TimeToWait	No	20000	클라이언트가 connection을 기다린다면 이것은 connection을 요청하도록 시도하는 것을 반복하기 전에 쓰레드가 기다려야 하는 최대시간이다. 이 시간내에서 connection이 pool로 반환이 될것이고 이 쓰레드에게 알려줄것이다. 대개 쓰레드는 이 프라퍼티가 정의된 값만큼 기다리지는 않을것이다.(이것은 간단히 말해 최대시간이다.)
Pool.PingQuery	No	n/a	ping쿼리는 connection을 테스트하기 위해 데이터베이스에 대해서 실행할것이다. Connection을 신뢰하지 않는 환경에서 이것은 pool이 좋은 connection을 반환할것을 보증하기 위한 ping쿼리를 사용하는데 유용하다. 어쨌든 이것은 성능에 약간 영향을 미친다. ping쿼리를 설정하는데 주의를 하고 너무 많이 테스트하지 말라.
Pool.PingEnabled	No	false	ping쿼리의 enable또는 disable상태로 만들기. 대부분의 애플리케이션은 ping쿼리가 필요하지 않다.
Pool.PingConnectionsOlderThan	No	0	이 프라퍼티값(milliseconds 단위)보다 오래된 connection은 쿼리를 사용하여 테스트될것이다. 이것은 일정시간(이를 테면 12시간)후에 당신의 데이터베이스환경이 공통적으로 connection을 없앤다면 유용하다.
Pool.PingConnectionsNotUsedFor	No	0	이 프라퍼티값보다 오래된 활성화되지 않은 connection은 ping쿼리를 사용해서 테스트될 것이다. 이것은 당신의 데이터베이스환경이 일정시간동안 활성화되지 않은 후에 공통적으로 connection을 없앤다면 유용하다.
*Driver.**	No	N/A	많은 JDBC드라이버는 기타의 프라퍼티를 제공함으로써 추가적인 기능을 제공한다. 당신의 JDBC드라이버에 이러한 프라퍼티를 넘기기위해서 당신은 프라퍼티이름에 “Driver”이라는 접두사를 정의할수 있다. 예를 들면 당신의 드라이버가 “compressionEnabled”라는 이름의 프라퍼티를 가진다면 당신은 “Driver.compressionEnabled=true”를 셋팅함으로써 SimpleDataSource을 설정할수 있다. 주의 : 이런 프라퍼티는 dao.xml과 SqlMap-config.xml파일내에 작동한다.

예제 **: SimpleDataSource** 사용하기

```
// properties usually loaded from a file
DataSource dataSource = new SimpleDataSource(props);
```

Connection conn = dataSource.getConnection();
// ... database queries and updates
conn.commit();
// connections retrieved from SimpleDataSource will return to the pool when closed
conn.close();

ScriptRunner (com.ibatis.common.jdbc.*)

ScriptRunner클래스는 데이터베이스 스키마를 생성하거나 디폴트내지 테스트데이터를 입력하는 것과 유사한 작업을 수행하는 SQL 스크립트를 실행할 때 매우 유용하다. ScriptRunner를 그대로 논의하는것보다 다음의 예제로 어떻게 간단하게 사용하는지 보라.

예제 스크립트 **: initialize-db.sql**

```
--Creating Tables – Double hyphens are comment lines
CREATE TABLE SIGNON (USERNAME VARCHAR NOT NULL, PASSWORD VARCHAR NOT NULL,
UNIQUE(USERNAME));
--Creating Indexes
CREATE UNIQUE INDEX PK_SIGNON ON SIGNON(USERNAME);
--Creating Test Data
INSERT INTO SIGNON VALUES('username','password');
```

예제 사용법 **1:** 현재 존재하는 **connection**을 사용하기

Connection conn = getConnection(); //some method to get a Connection
ScriptRunner runner = new ScriptRunner ();
runner.runScript(conn,
Resources.getResourceAsReader("com/some/resource/path/initialize.sql"));
conn.close();

예제 사용법 **2:** 새로운 **connection**을 사용하기

ScriptRunner runner = new ScriptRunner (“com.some.Driver”, “jdbc:url://db”, “login”, “password”); runner.runScript(conn, new FileReader("/usr/local/db/scripts/ initialize-db.sql"));

예제 사용법 **2:** 프라퍼티로부터 새로운 **connection**을 사용하기

Properties props = getProperties (); // some properties from somewhere
ScriptRunner runner = new ScriptRunner (props);
runner.runScript(conn, new FileReader("/usr/local/db/scripts/ initialize-db.sql"));

위 예제에서 사용되는 프라퍼티 파일(Map)은 다음과 같은 프라퍼티를 반드시 포함해야 한다.

driver=org.hsqldb.jdbcDriver
url=jdbc:hsqldb:.
username=dba
password=whatever
stopOnError=true

당신이 유용하게 사용할수 있는 몇몇 메소드는 :

```
// if you want the script runner to stop running after a single error
scriptRunner.setStopOnError (true);


// if you want to log output to somewhere other than System.out
scriptRunner.setLogWriter (new PrintWriter(…));
```

CLINTON BEGIN MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AS TO THE INFORMATION IN THIS DOCUMENT.

© 2004 Clinton Begin. All rights reserved. iBATIS and iBATIS logos are trademarks of Clinton Begin.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.