iBATIS

Data Mapper
(a.k.a SQL Maps)

Version 2.0

Developer Guide

November 30, 2006

翻訳：iBatisマニュアル日本語訳グループ
(ibatis_manual_japanese_translation_group@googlegroups.com)

目次

序文

iBATIS Data Mapper フレームワークは、リレーショナルデータベースへアクセスするのに必要な Java コードを著しく減少させる手助けをします。iBATIS は、とてもシンプルな XML の記述で JavaBeans を SQL ステートメントにマップします。シンプルなのは、他のフレームワークやオブジェクトリレーショナルツールよりも iBATIS が優れているところです。iBATIS Data Mapper を使うには、なじみやすい JavaBeans，XML と SQL だけが必要です。テーブルの結合や複雑なクエリーを実行するのに複雑なスキーマを必要しません。Data Mapper を使用することで、精通している SQL の全ての機能が利用できます。

Data Mapper (com.ibatis.sqlmap.*)

コンセプト

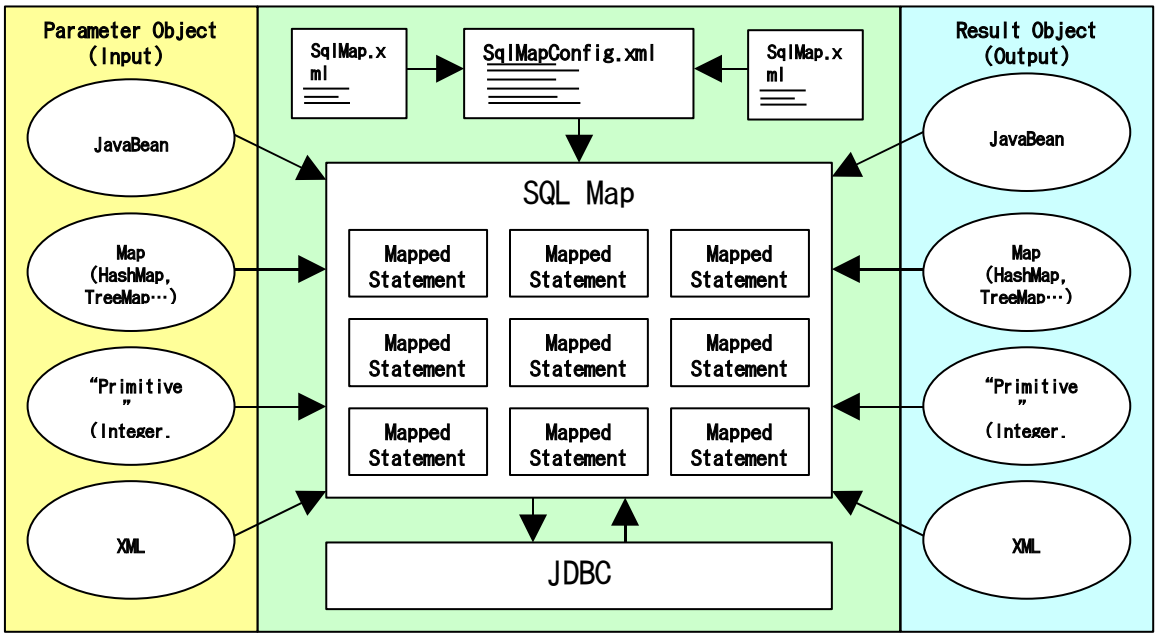
iBATIS Data Mapper API は、プログラマが JavaBeans のオブジェクトを PreparedStatement パラメータと ResultSets に簡単にマップできます。Data Mapper の考え方は、シンプルです。コードの 20%だけを使用して 80%の JDBC 機能を提供するシンプルなフレームワークを提供します。

どのように動作するか？

Data Mapper は、XML を使用して JavaBeans、Map の実装、プリミティブラッパータイプ、そして、XML さえも SQL ステートメントにマップするためにシンプルなフレームワークを提供します。下記は、ライフサイクルの高レベルな記述です：

1. JavaBean、Map、もしくはプリミティブラッパーのいずれかのオブジェクトをパラメータとして提供してください。パラメータオブジェクトは、クエリー内の update ステートメントまたは、where 句などの入力値として使用されます。
2. mapped statement を実行します。このステップは、魔法がおきるところです。Data Mapper フレームワークは、PreparedStatement インスタンスを作成して提供されたパラメータをセットし、ResultSet から結果オブジェクトを構築します。
3. 更新の場合、更新した行数を返します。検索の場合、単一のオブジェクトかオブジェクトのリストを返します。パラメータのように結果オブジェクトは、JavaBean、Map、プリミティブラッパータイプ、または、XML になることができます。

下記のダイアグラムは、記述したフローを示します。



インストール

iBATIS Data Mapper フレームワークのインストールは、classpath に適切なファイルを単に置くだけです。JVM 起動時に(java -cp 引数) で指定するか、Web アプリケーションの/WEB-INF/lib に置いてください。Java classpath については、本ドキュメントの範囲外となります。もし、Java かclasspath(もしくは、ともに) 不慣れであれば下記のリソースを参照してください：

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/

iBATIS は、下記の単一の JAR ファイルで配布されます。ファイル名は、この形になります：

ibatis-*version*.*build*.jar

例えば、**ibatis-2.3.0.677.jar となります。**

一般的には、この単一の JAR ファイルをアプリケーションのクラスパスに含めれば十分です。

JAR ファイルと依存性

フレームワークが多くの依存性を持ちすぎると、アプリケーションへの組み込みや他のフレームワークとの連携が難しくなります。2.0 のキーポイントの 1 つは、依存性の管理と減少です。したがって、もし JDK1.4 で iBATIS を動作させるのであれば、iBATIS は、他の依存性を持ちません。オプショナルの JAR ファイルは、下記のウェブサイトからダウンロードできます。それらは、機能によってカテゴライズされています。下記にオプショナルのパッケージが必要な時をまとめています。

記述	使用するとき	依存
レガシー JDK サポート	JDK1.4 より前の JDK を利用していてアプリケーションサーバがこれらの JAR を提供していない時にこれらのオプショナルのパッケージを必要とします。	**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 後方互換性	iBATIS の DAO フレームワーク(1.x)使用しているか古い Data Mapper(1.x)を使用しているときに、単に JAR ファイルを含めるだけで使いつづけられます。	**iBATIS DAO 1.3.1** **http://sourceforge.net/projects/ibatisdb/**
実行時バイトコード拡張	lazy loading とリフレクションのパフォーマンスを改善するために CGLIB 2.0 を有効にしたい時	**CGLIB 2.0** **http://cglib.sf.net**
データソース実装	Jakarta DBCP コネクションプールを使用したい時	**DBCP 1.1** **http://jakarta.apache.org/commons/dbcp/**
分散キャッシング	集中もしくは、分散されたキャッシュサポートのために OSCache を使用したい時	**OSCache 2.0.1** **http://www.opensymphony.com/oscache/**
ロギング	Log4J ロギングを使用したい時	**Log4J 1.2.8** **http://logging.apache.org/log4j/docs/**
ロギング	Jakarta Commons Logging を使用したい時	**Jakarta Commons Logging** **http://jakarta.apache.org/commons/logging**

1.xからのアップグレード

アップグレードすべきかどうか？

アップグレードすべきかどうか決めるベストな方法は、それらを試すことです。いくつかのアップグレードする方法があります。

1. バージョン2.0は、1.xリリースとほぼ完全に後方互換性を持つようにメンテナンスされています。そのため、JARファイルを単に置き換えるだけで十分な人もいます。このアプローチは、最もシンプルですがメリットが一番少ないです。XMLファイル、またはJavaコードを変更する必要がありません。非互換性がいくつか見付かるかもしれません。

2. 二つ目のオプションは、XMLファイルを2.0の仕様に変換することです。しかし、1.x Java APIを使いつづけなければなりません。いくつかのマッピングファイル間で互換性の問題が発生するかもしれませんが安全な解決策です。下記に記述しているXMLを変換するためのAntタスクはフレームワークに含まれています。

3. 三つ目のオプションは、（2.と同じく）XMLファイルの変換とJavaコードの変換をすることです。Javaコードの変換ツールはないので、手作業で変換しなければなりません。

4. 最後のオプションは、アップグレードをしないことです。アップグレードが困難であれば1.xリリースで動作しているシステムをそのままにします。1.xアプリケーションを離れて新しいアプリケーションのみ2.0ではじめることは悪いアイデアではないかもしれません。もちろん、いずれにしろ古いアプリケーションに（ある基準以上の）大規模なリファクタリングがされる予定ならば、Data Mapperのアップグレードも同様に行った方が良いかもしれません。

1.Xから2.XへのXML設定ファイルの変換

2.0フレームワークは、Antで動作させるXMLドキュメントコンバータを含んでいます。XMLドキュメントを変換することは、1.xコードが自動的にすぐさま古いXMLファイルを自動的に変換するため完全に任意です。しかしながら、ひとまずアップグレードに不安がないのであればファイルを変換させることは良い考えです。(1.x Java APIを使いつづけたとしても)いくつかの互換性の問題をくぐりぬければ新機能のアドバンテージを得られます。

build.xmlのAntタスクは以下のようになります:

```
<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>
```

ご覧のとおり、Ant Copyタスクと同じような働きをします、そして実際の所Antタスクを拡張しています。だから、Copyタスクでできることは、行うことができます。(詳細については、Ant Copyタスクのドキュメントを参照してください。)

JAR ファイル:古きを捨て、新しきを得る

アップグレードする時に、(古い)iBATIS ファイルを削除して新しいファイルに置き換えることは良い考えです。他のコンポーネント、もしくはフレームワークがまだ古い JAR ファイルを必要としていないか確認してください。JAR ファイルのほとんどは、状況に応じて任意の依存であることに気を付けてください。JAR ファイルと依存に関する詳細情報は上記の説明を見てください。

古い JAR ファイルと新しい JAR ファイルを下記のテーブルにまとめています。

古いファイル	新しいファイル
ibatis-db.jar *1.2.9b リリース以降、このファイルは下記の３つのファイルに分割されました。* ibatis-common.jar ibatis-dao.jar ibatis-sqlmap.jar	Ibatis-*version.build*.jar ***(必須)***
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 ***(オプショナル)*** commons-collections-2-1.jar ***(オプショナル)*** commons-dbcp-1-1.jar ***(オプショナル)*** commons-pool-1-1.jar ***(オプショナル)*** oscache-2-0-1.jar ***(オプショナル)*** jta-1-0-1a.jar ***(オプショナル)*** jdbc2_0-stdext.jar ***(オプショナル)*** xercesImpl-2-4-0.jar ***(オプショナル)*** xmlParserAPIs-2-4-0.jar ***(オプショナル)*** xalan-2-5-2.jar ***(オプショナル)*** log4j-1.2.8.jar ***(オプショナル)*** cglib-full-2-0-rc2.jar ***(オプショナル)***

ガイドの残りは、Data Mapper フレームワークの使い方を紹介します。

SQLマップ設定ファイル (http://ibatis.apache.org/dtd/sql-map-config-2.dtd)

Data Mapperは、データソースの詳細設定、Data Mapperとスレッド管理のような他のオプションを設定するために、XML設定ファイルを用いて設定を行います。下記は、SQLマップ設定ファイルの例です。：

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">

<!-- Always ensure to use the correct XML header as above! -->
<sqlMapConfig>

  <!-- The properties (name=value) in the file specified here can be used placeholders in this
config
        file (e.g. "${driver}". The file is relative to the classpath and is completely
optional. -->
  <properties resource=" examples/sqlmap/maps/SqlMapConfigExample.properties " />

  <!-- These settings control SqlMapClient configuration details, primarily to do with
transaction
        management. They are all optional (more detail later in this document). -->
  <settings
    cacheModelsEnabled="true"
    enhancementEnabled="true"
    lazyLoadingEnabled="true"
    maxRequests="128"
    maxSessions="10"
    maxTransactions="5"
    useStatementNamespaces="false"
    defaultStatementTimeout="5"
    statementCachingEnabled="true"
    classInfoCacheEnabled="true"
  />

  <!-- This element declares a factory class that iBATIS will use for creating result objects.
       This element is optional (more detail later in this document). -->
  <resultObjectFactory type="com.mydomain.MyResultObjectFactory" >
      <property name="someProperty" value="someValue"/>
  </resultObjectFactory>

  <!-- Type aliases allow you to use a shorter name for long fully qualified class names. -->
  <typeAlias alias="order" type="testdomain.Order"/>

  <!-- Configure a datasource to use with this SQL Map using SimpleDataSource.
       Notice the use of the properties from the above resource -->
  <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>

  <!-- Identify all SQL Map XML files to be loaded by this SQL map.  Notice the paths
        are relative to the classpath.  For now, we only have one... -->
  <sqlMap resource="examples/sqlmap/maps/Person.xml" />
</sqlMapConfig>
```

このドキュメントの下記のセクションでSQL Map設定ファイルのさまざまな説明をしています。

<properties>エレメント

SQL Mapは、SQL Map設定ファイルと関連する標準のJavaプロパティ(名前=値)を受け入れる一つの<properties>エレメントを持つことができます。プロパティファイルの名前付けされた値は、SQL Map設定ファイルと全てのData Mapperの中で参照できるようになります。例えば、プロパティファイルが下記を含んでいるとします。：

```
driver=org.hsqldb.jdbcDriver
```

設定ドキュメントによって参照されるSQL Map設定ファイルかSQL Mapは、
org.hsqldb.jdbcDriverと置き換えられるプレースホルダーを使用できます。例えば：

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

これは、ビルド、テスト、デプロイに、とても便利です。複数の環境のリコンフィグや(例.Ant)設定のための自動化ツールの使用を容易にします。プロパティファイルは、classpath(resource属性を使う)または、特定の有効なURL(url属性を使う)からロードできます。下記は、固定パスファイルをロードして使用する例です。：

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

<settings>エレメント

<settings>エレメントは、様々なオプションの設定とXMLファイルを使用して構築するSqlMapClientの最適化をできます。settingsエレメントと全ての属性は完全に任意です。サポートされている属性と振舞いを下記のテーブルに記述します。：

maxRequests	これは、一度にSQLを実行するスレッドの最大数です。最大数を超えるスレッドは、別のスレッドが完全に実行するまでブロックされます。DBMSごとに異なる制限を持ちますが、制限を持たないデータベースはありません。通常最低でもmaxTransaction(下記参照)の10倍とします。この値は、常にmaxSessionとmaxTransactionより大きくします。同時リクエスト数の最大数を減らすとパフォーマンスの向上が、よく見受けられます。 *サンプル: maxRequests="256"* *デフォルト: 512*

maxSessions	一時にアクティブになることができるセッション（もしくはクライアント）の数です。セッションは、プログラム的に要求した明示的なセッションか（例：ステートメントの実行まで）SqlMapClient インスタンスが使うスレッドです。この値は、maxTransaction 以上の値で maxRequest より小さい値を常に指定するべきです。同時セッションの最大数を減らすとメモリ占有量を減少させることができます。 *サンプル: maxSessions="64"* *デフォルト: 128*
maxTransactions	一度に SqlMapClient.startTransaction()に入ることができるスレッドの最大数。最大数以上のスレッドは、他のスレッドが存在している間ブロックされます。異なる DBMS は、異なる制限を持ちます。しかし、それらの制約を持たないデータベースはありません。この値は、常に maxSession 以下でかつ、maxRequest よりも、とても少なくなければなりません。最大同時トランザクション数を減らすことで、しばしばパフォーマンスを向上させることができます。 *サンプル: maxTransactions="16"* *デフォルト: 32*
cacheModelsEnabled	この設定は、SqlMapClient のための全てのキャッシュモデルを全体的に有効、または無効にします。デバッグに役立てることができます。 *サンプル: cacheModelsEnabled="true"* *デフォルト: true (enabled)*
lazyLoadingEnabled	この設定は、SqlMapClient のための全ての lazy loading を全体的に有効、または無効にします。デバッグに役立てることができます。 *サンプル: lazyLoadingEnabled="true"* *デフォルト: true (enabled)*
enhancementEnabled	この設定は、lazy loading 拡張をするだけではなく、容易に最適化した JavaBeans プロパティアクセスするための実行時のバイトコード拡張を有効にします。 *サンプル: enhancementEnabled="true"* *デフォルト: false (disabled)*
useStatementNamespaces	この設定が有効であれば、常に SqlMap 名とステートメント名の完全修飾子でマップされたステートメントを参照しなければなりません。例： queryForObject("sqlMapName.statementName"); *サンプル: useStatementNamespaces="false"* *デフォルト: false (disabled)*

defaultStatementTimeout	(iBATIS 2.2.0 以降) この設定は、全てのステートメントのための JDBC クエリータイムアウトとして適用される integer の値。この値は、マップされたステートメントの”statement”の属性値で上書きできます。この値を指定しなければマップされたステートメントの”statement”属性で上書きしない限りクエリータイムアウトはセットされません。指定した値は、ドライバがステートメントの実行まで待てる秒数です。全てのドライバがこの設定をサポートしてないことに注意してください。
classInfoCacheEnabled	この設定が有効であれば、iBATIS は、キャッシュされたクラスを維持します。もし多くのクラスが再利用されるのであれば起動時間の大幅な減少をもたらすでしょう。 *サンプル: classInfoCacheEnabled=“true”* *デフォルト: true (enabled)*
statementCachingEnabled	(iBATIS 2.3.0 以降) この設定が有効であれば、iBATIS は、prepared statements のローカルキャッシュを維持します。これは、大幅なパフォーマンスの改善をもたらすことができます。 *サンプル: statementCachingEnabled=“true”* *デフォルト: true (enabled)*

<resultObjectFactory>エレメント

Important: この機能は、iBATIS バージョン 2.2.0 以降で利用可能です。

resultObjectFactory エレメントは、SQL ステートメントの実行結果からクラスを作成するためのファクトリクラスを指定できます。このエレメントはオプショナルです。エレメントを指定しなければ、iBATIS は、結果オブジェクトを作成するために内部メカニズム (class.newInstance())を使うでしょう。

これらのケースにおいて iBATIS は、結果オブジェクトを作成します。

1. ResultSet から返された行をマップする時(最も共通したケース)
2. resultMap の中の result エレメントでネストした select ステートメントを使用する時。もし、ネストした select ステートメントで parameterClass を宣言していれば、ネストした select ステートメントを*実行する前に、iBATIS はクラスのインスタンスを作成、代入して利用可能にします。*
3. ストアドプロシージャを実行する時。iBATIS は、OUTPUT パラメータのオブジェクトを作成します。
4. ネストした result map を処理する時。ネストした result map が、N+1 クエリーを回避するために、groupBy サポートを同時に使用するのであれば、オブジェクトは、通常 Collection, List, Set の実装クラスとなります。もし望むのであれば、それらのインタフェースのカスタム実装を resultObjectFactory を通じて提供できます。ネストした result map と 1:1 で結合する場合において、iBAITS は、このファクトリを通じて指定したドメインオブジェクトのインスタンスを作成します。

ファクトリの実装を選択するのであればファクトリクラスは、
com.ibatis.sqlmap.engine.mapping.result.ResultObjectFactory を実装しなければなりません。そして、ファクトリの実装クラスは、public なデフォルトコンストラクタを持たなければなりません。ResultObjectFactory インタフェースは、2 つのメソッドを持ちます。1 つはオブジェクトの作成。そして、あと 1 つは、設定で指定されたプロパティ値を受け入れます。

例えば、*resultObjectFactory* の設定エレメントは、このように指定します：

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

それから、result object factory クラスのコードは以下のようにするべきでしょう：

```
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...
  }
```

```
}
```

設定において、iBATISは、各プロパティを指定するために、一度setPropertyメソッドを呼び出します。createInstanceメソッドが、いずれかの呼出しで処理される前に全てのプロパティは、セットされます。

iBATISは、上記のオブジェクトの作成が必要なケースに、都度createInstanceメソッドを呼び出します。もし、createInstanceからnullが返されたらiBATISは、通常の方法(class.newInstance())を使用してオブジェクトの作成を試みます。もし、java.util.Collectionかjava.util.Listの作成要求からnullが返された場合、iBATISは、java.util.ArrayListを作成します。iBATISは、オブジェクト作成要求があったコンテキストに現在のステートメントidを知らせます。

<typeAlias>エレメント

typeAliasエレメントは、完全修飾された長いクラス名を、指定した短い名前で参照できるようにします。例：

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

SQL Map Config ファイルで、いくつかのエイリアスが事前に定義されています。
事前定義されているものは、以下です：

トランザクションマネージャ　エイリアス	
JDBC JTA EXTERNAL	com.ibatis.sqlmap.engine.transaction.jdbc.JdbcTransactionConfig com.ibatis.sqlmap.engine.transaction.jta.JtaTransactionConfig com.ibatis.sqlmap.engine.transaction.external.ExternalTransactionConfig
データソースファクトリ　エイリアス	
SIMPLE DBCP JNDI	com.ibatis.sqlmap.engine.datasource.SimpleDataSourceFactory com.ibatis.sqlmap.engine.datasource.DbcpDataSourceFactory com.ibatis.sqlmap.engine.datasource.JndiDataSourceFactory

<transactionManager>エレメント

1.0変換時の注意点：Data Mapper 1.0は、複数のデータソース設定が許可されていました。これは、扱いにくく悪いプラクティスを招きました。従って2.0は、単一のデータソースのみ使用可能となっています。複数の配備/設定には、システムごとに異なる設定、またはSQL Mapを作成するときパラメータとして、複数のプロパティファイルを使用することを推奨します。(下記のJava APIセクション参照)

<transactionManager>エレメントは、SQL Mapのためにトランザクション管理サービスを設定できます。type属性に、使用するトランザクションマネージャを指定します。その値は、クラス名かtypeAliasを指定できます。JDBC, JTA, EXTERNALの3つのトランザクションマネージャは、フレームワークに含まれています。

JDBC - Connectionのcommit()とrollback()メソッド経由で、JDBCトランザクションコントロールが可能です。

JTA - このトランザクションマネージャは、(他のデータベースまたは、トランザクションリソースを含めることができる)広いトランザクションスコープの一部として、SQL Mapを含むことができるJTAグローバルトランザクションを使用します。 この設定は、JNDIリソースからuser transactionを見つけられるUserTransationプロパティセットを必要とします。この設定のサンプルは、下記のJNDIデータソースサンプルを見てください。

EXTERNAL – 自分自身で、トランザクションを管理することを可能にします。データソースの設定をできますが、トランザクションは、フレームワークライフサイクルの一部としてコミット、またはロールバックされません。これはData Mapperの外部のアプリケーションの一部でトランザクションを管理しなくてはならないことを意味します。この設定も（例.読み取り専用）トランザクションがないデータベースの役に立ちます。

〈transactionManager〉エレメントは、trueかfalseとなるオプショナルのcommitRequired属性も使用できます。通常iBATISは、insert，update，delete操作を実行しない限りトランザクションをcommitしません。これは、明示的にcommitTransaction()メソッドを読んだとしてもです。この振舞は、いくつかのケースにおいて問題があります。もしinsert,update，delete操作しなくても常にトランザクションをcommitしたいのであれば、commitRequired属性をtrueをセットしてください。この属性が便利なときの例：

1. 更新の結果行を返し、かつデータを更新するストアドプロシージャを呼出したい時。このケースにおいて、queryForList()操作で、そのプロシージャを呼び出します。そのためiBATISは、通常そのトランザクションをコミットしません。そのために、その更新はロールバックされます。
2. WebSphere環境で、コネクションプールとJNDI〈datasource〉とJDBCかJTAトランザクションを使用するケース。WebSphereは、プールされたコネクションにおける全てのトランザクションをコミットすることが必要です。さもなければコネクションはプールに戻りません。

EXTERNALトランザクションマネージャを使っている時は、commitRequired属性の効果がないことに気を付けてください。

いくつかのトランザクションマネージャは、設定ファイルの拡張が可能です。下記のテーブルはそれらのトランザクションマネージャで利用可能な拡張プロパティを記述しています。

トランザクションマネージャ	*プロパティ*	
EXTERNAL	*プロパティ*	*説明*
	DefaultAutoCommit	trueであれば、もしベースとなるデータソースによって値が提供されていないとしても、各トランザクションのベースとなるコネクションに対して、setAutoCommit(true)が呼び出されます。 falseか指定されていないければ、もしベースとなるデータソースによって値が提供されていないとしても各トランザクションのベースとなるコネクションに対して、setAutoCommit(false)が呼び出されます。 この振舞いは、SetAutoCommitAllowedプロパティで、上書きできます。
	SetAutoCommitAllowed	trueか指定されていなければ、”DefaultAutoCommit”プロパティで指定されている動作となります。 falseであれば、iBATISは、いかなるケースにおいてもsetAutoCommitを呼び出しません。WebSphereのようにいかなる状況でもsetAutoCommitメソッドを呼び出すべきでない環境で役に立ちます。

トランザクションマネージャ	*プロパティ*
JTA	*プロパティ* / *説明* UserTransaction / このプロパティは必須です。 user transaction の値。多くのケースでは、”java:comp/UserTransaction”をセットするべきことに注意してください。

<dataSource>エレメント

トランザクションマネージャの一部として含まれているのは、dataSource エレメントと SQL Map のために使われるデータソースを設定するプロパティのセットです。現在、フレームワークと一緒に３つのデータソースファクトリが提供されています。しかし、独自のファクトリを作成することもできます。含まれている DataSourcefactory の実装は、以降でそれぞれの詳細を説明しています。そして、それぞれの設定例を記しています。

SimpleDataSourceFactory

SimpleDataSourceFactoryは、コンテナが提供するデータソースがないケースにおいて、プールしたデータソースの基本的な実装を提供します。それは、iBATIS SimpleDataSourceコネクションプール実装をベースにしています。

```
<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”プレフィックスを持つプロパティに気を付けてください。それらは、JDBCドライバの基本的なプロパティに追加されます。

DbcpDataSourceFactory

この実装は、DataSource APIを経由してコネクションプーリングサービスを提供するためにJakarta DBCP(データベースコネクションプール)を使用します。このデータソースは、アプリケーション/webコンテナがデータソースを実装していない時か、スタンドアロンアプリケーションを動作させるためのものです。設定において希望する指定DBCPプロパティを指定することによってiBATISは、DBCPデータソースのプロパティへの直接的なアクセスを提供します。例えば：

```
<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>
```

利用可能な全てのプロパティは、ここから参照できます:

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

“Driver”プレフィックスを持つプロパティに気を付けてください。それらはJDBCドライバの基本的なプロパティに追加されます。

iBATISは、下記の柔軟ではないレガシー設定オプションもサポートしています。しかしながら、上記の設定オプションの使用を推奨します。

```
<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>
```

示されたプロパティは、レガシー設定オプションを使用している時だけ、iBATISに認識されるプロパティです。“Driver”プレフィックスを持つプロパティに、気を付けてください。それらはJDBCドライバの基本的なプロパティに追加されます。

JndiDataSourceFactory

この実装は、コンテナのJNDIコンテキストからデータソースの実装を利用します。これは、アプリケーションサーバからコネクションプールが提供されている場合に、標準的に使用されます。JDBC DataSource実装にアクセスする標準的な方法は、JNDIを経由する方法です。 JndiDataSourceFactoryは、JNDI経由でデータソースにアクセスするための

機能を提供します。データソースの設定において、指定しなければならない設定パラメータは以下です：

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

上記の設定は、通常のトランザクション管理を使用します。しかし、コンテナが管理するリソースで、下記のようにグローバルなトランザクションを設定したいかもしれません。

```
<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 プロパティは、UserTransaction インスタンスを見つけられる JNDI のロケーションを指し示します。Sql Map が他のデータベースとトランザクショナルリソースを巻き込めるスコープの広いトランザクションに参加するために、JTA トランザクション管理は必須です。

“context”プレフィックスを持つプロパティを追加することによって、lookup の前に JNDI コンテキストプロパティに追加できます。例:

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

<sqlMap>エレメント

sqlMapエレメントは、SQL Mapまたは、別のSQL Map設定ファイルを、明示的に含めるために使われます。それぞれのSQL Map XMLファイルは、SqlMapClientインスタンスによって使われるために宣言する必要があります。SQL Map XMLファイルは、classpath、もしくはURLからストリームリソースとしてロードされます。（ある限りの）ありとあらゆるData Mapperを指定しなければなりません。下記は、サンプルです:

```
<!-- 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" />
```

次のいくつかのセクションで SQL Map XML ファイルの構造を詳しく記述します。

SQL Map XMLファイル
(http://ibatis.apache.org/dtd/sql-map-config-2.dtd)

上記の例で、Data Mapperの最もシンプルな形を見ました。SQL Mapドキュメント構造で利用できる他のオプションがあります。これは、より多くの機能を使用するmapped statementの例です。

```
<?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>
```

多すぎますか？フレームワークは、多くのことをしているにも関わらず、シンプルなselectステートメントのために、多くの余分なこと(XML)をしているように見えるかもしれません。心配しないでください。以下は、上記を簡単にしたバージョンです。

```
<?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>
```

現在、これらのステートメントは、SQL Mapの振舞いにおいて全く同じではありません。始めに、後者のステートメントは、キャッシュを定義していません。その為に、全てのリクエストはデー

タベースに発行されます。次に、後者はオーバーヘッドを伴うフレームワークの自動マッピング機能を利用しています。しかしながら、２つのステートメントともにJavaコードから同じように実行されます。従ってシンプルなやり方で始めて、より詳細なマッピングが必要となった時に移行できます。最もシンプルなソリューション始めることは、最近の方法論においてはベストプラクティスです。

単一のSQL Map XMLファイルは、多くのセッションモデル、parameter map、Result Mapとステートメントを含むことができます。アプリケーションのために、ふさわしいステートメントとマップを組織化(論理的なグループと)してください。

Mapped Statements

Data Mapperのコンセプトは、mapped statementsを中心としています。Mapped statementsは、parameter map(入力)とResult Map(出力)を持つSQLステートメントを構成することことができます。シンプルなケースであれば、パラメータと結果を直接マップされたステートメントに設定できます。mapped statementマップされたステートメントは、メモリでキャッシュしている一般的な結果にキャッシュモデルを使うように設定することもできます。

```
<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>
```

ステートメントは、*insert*, *update*, *delete*, *select*, *procedure*、もしくは*statement*となります。上記のステートメントにおいて[]で示されている箇所はオプションの設定です。そして、いくつかのケースにおいては、組み合わせて使用することもできます。ルールに則りシンプルな形でマップされたステートメントは、このようになります。:

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

上記の例では、明示的でありませんが、任意のSQLを実行するためにSQL Mapフレームワークを簡単に使用したいのであれば便利です。しかしながら、parameter mapとResult Mapを使用してJavaBeansマッピング機能を使用することは、本当の力であってより一般的です。以降のセクションで、構造と属性、そして、それらがどのようにmapped statementに影響するのかについて記述します。

Statementの種類

<statment>エレメントは、SQLステートメントのどの種類でも使用することが一般的な”catch all”ステートメントです。一般的に、一つ以上のstatementエレメントを指定するには良い考えです。より特化したエレメントは、より直感的なXML DTDと通常の<statement>ではできない追加機能を提供します。下記のテーブルは、statementエレメントとそれらがサポートする属性と機能をまとめたものです。:

ステートメント エレメント	属性	子要素	メソッド
<statement>	id parameterClass resultClass parameterMap resultMap cacheModel resultSetType fetchSize xmlResultName remapResults timeout	All dynamic elements	insert update delete All query methods
<insert>	id parameterClass parameterMap timeout	All dynamic elements <selectKey>	insert update delete
<update>	id parameterClass parameterMap timeout	All dynamic elements	insert update delete
<delete>	id parameterClass parameterMap timeout	All dynamic elements	insert update delete
<select>	id parameterClass resultClass parameterMap resultMap cacheModel resultSetType fetchSize xmlResultName remapResults timeout	All dynamic elements	All query methods
<procedure>	id parameterClass resultClass parameterMap resultMap cacheModel fetchSize xmlResultName remapResults timeout	All dynamic elements	insert update delete All query methods

The SQL

SQL は、明らかにマップの中で最も重要な部分です。データベースと JDBC ドライバで有効な SQL を使用できます。どんなファンクションも利用可能です。そして、ドライバがサポートしていれば複数のステートメントさえ送ることができます。単一のドキュメントに SQL と XML を混ぜるので特別な意味をもつ文字の意味が重複してしまいます。最も共通しているのは、大なり小なり(<>)シンボルです。それらは、SQL においても必須ですし XML タグの予約シンボルです。簡単な解決策は、他の特別な XML タグでくくり SQL に記述することです。標準的な XML CDATA セクションを使用することによって特別な文字として解釈させないことで解決できます。例えば：

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

SQL フラグメントの再利用

SqlMap を書いている時に、しばしば重複した SQL のフラグメントが出てきます。例えば、FROM 句もしくは、条件句です。iBATIS は、それらを再利用するためにシンプルでパワフルなタグを提供します。シンプルさを目的として、いくつかのアイテムを得たいとして、それらの数を取得したいとしましょう。通常、このように書きます：

```
<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>
```

重複を取り除くために、<sql>と<include>タグを使用します。<sql>タグは、フラグメントの再利用をするために、それぞれのフラグメントを含む<include>タグをステートメントの中に含みます。例えば：

```
<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>タグは、ネームスペースに配慮しています。そのために、別のマップにフラグメントがあっても参照できます。(しかしながら、iBATIS が、SqlMap をロードするために含められるフラグメントを、事前にロードしておかなければなりません)

フラグメントは、含められてからクエリーとして処理されます。そのため、パラメータも使用できます：

```
<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>
```

キーの自動生成

多くのリレーショナルデータベースシステムは、主キーの自動生成をサポートしています。この機能は、（常にではないが）しばしば適切な機能です。Data Mapper は、<insert>ステートメント中の<selectKey>を通じてキーの自動生成をサポートします。事前(Oracle)と事後(SQL Server)の両方のキー生成をサポートしています。２つの例です：

```
<!—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>
```

もし、insert ステートメントより前に selectKey があれば、insert の前に実行されます。そうでなかれば、insert の後に実行されます。始めの例では、（シーケンスにとって適切なように）insert ステートメントの前に実行される Oracle の selectKey の例を示しています。SQL Server の例は、（identity カラムにとって適切なように)insert ステートメントの後に実行される selectKey の例を示しています。

iBATIS バージョン 2.2.0 以降では、希望するのであればステートメント実行の順序を明確に指定することがでいます。selectKey エレメントは、ステートメントの順序を明確にするためにセットする type 属性をサポートしています。type 属性の値は、pre か post のどちらかです。値の意味は、insert ステートメントの前が後に実行することです。もし、type 属性を指定したのであれば、selectKey エレメントの場所に関係無く指定した値が採用されます。例えば、エレメントが insert ステートメントの後にあっても、insert の前に実行できます。

```
<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>属性リファレンス:

<selectKey>属性	*説明*
resultClass	<selectKey>ステートメント実行の結果として、生成されるべき Java クラス(通常 Integer または、Long)
keyProperty	<selectKey>ステートメント実行の結果として、パラメータオブジェクトにセットされるプロパティ。もし、指定されていなければ、iBATIS は、データベースから返されたカラム名をベースに、プロパティを見つけようとします。もし、プロパティが見付からなければ、プロパティはセットされないでしょう。しかし、iBATIS は、<insert>ステートメントの結果として生成されたキーを依然として返すでしょう。
type	“pre” または “post”。もし、指定されば、select key ステートメントは、関連する insert ステートメントの前(pre)または、後(post)に実行されることを意味する。 もし、指定しなければ、insert ステートメントの中のエレメントの場所から、実行順序を推測します。(もし、SQL の前に置かれていれば、selectKey は、statement の前に実行されます) この属性は、iBATIS versions 2.2.0 以降でのみ利用可能です。

ストアドプロシージャ

ストアドプロシージャは <procedure> エレメントによってサポートされます。 次の例ではどのようにストアドプロシージャが出力パラメータとともに利用されるかを示します。

```
<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>
```

上記のプロシージャを呼び出すことにより、２つの列(データベーステーブル)のメールアドレス、同様にパラメータオブジェクト(Map)の内容が交換されます。 このパラメータオブジェクトは <parameter> エレメントの *mode* 属性が“INOUT” または “OUT” に設定されている場合のみ変更されます。それ以外の場合は変更されません。明らかに不変のパラメータオブジェクト(例. String)は変更できません。

Note! 常に標準の JDBC ストアドプロシージャ構文を使うように心がけましょう。詳しい情報は JDBC CallableStatement documentation を参照してください。

parameterClass

parameterClass 属性は完全修飾名された Java クラス(すなわち package を含む)の値を指定します。この parameterClass 属性はオプショナルですが、強く推奨されます。 これは限定されたパラメータによるステートメントの実行のみを許可するためと、同時にフレームワークのパフォーマンスを最適化します。 もしあなたが parameterMap 属性を使う場合、 parameterClass 属性を指定する必要はありません。例えば、あなたがオブジェクト型(例. instanceof) “examples.domain.Product” をパラメータとして許可したいのであれば、このように記述できます：

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

IMPORTANT: 下位互換性に対しては任意ですが、parameter class を常に指定することは強く推奨されています（もちろんパラメータが不要な場合を除いて)。 **このクラスを指定することによりより良いパフォーマンスを達成できるでしょう、なぜならフレームワークはあらかじめ型を知ることにより、処理を最適化する可能性があるからです。**

特定の parameterClass が無い場合、適切なプロパティ(get/set メソッド)を持ついずれかの JavaBean も、パラメータとして利用可能です。これはいくつかの状況で非常に有益に成り得ます

parameterMap

parameterMap 属性の値は〈parameterMap〉エレメント(下記参照)として定義された名前を指定します。 このparameterMap 属性は稀に parameterClass 属性(上記)とインライン Parameters 属性(後述)のために使われることがあります。しかし、XML の純粋性と一貫性があなたの関心事であるか、または、より説明的な parameterMap（例.ストアドプロシージャのため）を必要とするなら、これは良いアプローチです。

Note! Dynamic mapped statements（後述）は インライン Parameters のみをサポートし、parameter map には機能しません。

parameterMap の目的は JDBC PreparedStatement の値トークンと一致する、順序付きのパラメータのリストを定義することです。例えば ：

```
<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>
```

この上記の例では、この parameter map は２つのパラメータが順に SQL ステートメントの値トークン(“?”)にマッチすることを表しています。つまり始めの “?” は “id” のプロパティ値により置き換えられ、二つ目は “description” プロパティとなります。parameter map とそのオプションの詳細については、この文書で後ほど解説します。

インライン Parameters クイックグランス

さらなる詳細は、この文書の後ほどで解説しますが、ここでは手短にインライン parameter の紹介をします。インライン parameter は mapped statement の中で使用されます。例えば ：

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

上記の例において、インライン parameter は *#id#* と *#description#* です。それぞれステートメントの適切な場所へ配置させるために使われる JavaBean のプロパティを表します。上記の例では、Product クラス(前述の例で使用しました)は、関連するプロパティトークンが位置するステートメント中で置き換えられる値として読み込まれる *id* と *description* プロパティを持ちます。 つまりステートメントが Product（*id=5* , *description=" dog"* ）により実行される時, このステートメントは以下のように実行されることでしょう:

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

resultClass

resultClass 属性は完全修飾名された Java クラス(すなわち package を含む)の値です。 resultClass 属性は ResultSetMetaData に基づき JDBC ResultSet により自動的にマッピングされるクラスを指定できます。どこであれ JavaBean のプロパティと ResultSet のカラムが一致する際、プロパティは、カラムの値が代入されます。これによりクエリーマップされたステートメントを短期間かつ円滑に作成できます。例えば:

```
<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>
```

上記の例では、この Person クラスは次のプロパティを含みます: *id, firstName, lastName, birthDate, weightInKilograms, heightInMeters*。それぞれが SQL の SELECT ステートメント(標準 SQL 機能の "as" キーワードによる)により記述されたカラムのエイリアスに合致します。 カラムのエイリアスはデータベースのカラム名が一致しない場合のみに必要ですがこれは一般的ではありません。実行すると、Person オブジェクトのインスタンスが生成され、プロパティ名とカラム名に基づいて結果セットの値がマップされます。

先に述べたとおり、自動マッピングを resultClass で行うにはいくつかの制限があります。(もし必要な場合)出力される列の型を指定する方法がないこと、自動的に関連するデータ(複合プロパティ)をロードする方法が無いこと、そしてこの手法が ResultSetMetaData へのアクセスを要求することによる若干のパフォーマンスへの影響もあります。これら全ての制限は resultMap を明示的に使用することで解決できます。resultMap については、この文書で後ほど詳細に記述します。

resultMap

この resultMap プロパティはより一般的に使われる、最も重要な属性と考えられます。 resultMap 属性の値は resultMap エレメント(下記参照)により定義された名前です。 この resultMap 属性の使用により、どのようにデータを結果セットから抽出するかとそのプロパティが、どのカラムにマップされるかを制御することが出来ます。 resultClass 属性(上記)を使った自動マッピング手法と違い、この resultMap は列の型、null 値の置き換えと組み合わせたプロパ

ティのマッピングを記述できます。(他の JavaBean、Collection と基本データ型のラップクラスを含む)

resultMap の詳細の構造についてはこの文書の後で説明します、しかし次の例で resultMap がどの様に関連するステートメントを参照するかを示します。

```
<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>
```

上記の例では、SQL クエリーからの ResultSet が resultMap 定義にしたがって Product インスタンスにマップされます。この ResultMap では" id" プロパティが" PRD_ID" 列、そして" description" プロパティが" PRD_DESCRIPTION" 列で代入されることを表します。"select *" がサポートされていることに注目してください。ResultSet の全ての列をマップさせる必要はありません。

cacheModel

cacheModel 属性値は cacheModel エレメント(下記参照)で定義される名前です。cacheModel はクエリーマップがマップされたステートメント と共に使用されるキャッシュを記述するために使われます。個々のクエリーマップがマップされたステートメントは別の cacheModel も同じ cacheModel も利用できます。cacheModel エレメントとその属性の全詳細については後に解説します。次の例はこれが関連するステートメントをどの様に参照するかを表します。

```
<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>
```

上記の例では弱い参照タイプで 24 時間毎または関連する更新ステートメントが実行される毎にフラッシュされる product へのキャッシュを定義します。

xmlResultName

結果を直接 XML 文書にマッピングするとき、xmlResultName の値はこの XML 文書の root エレメントになります。例えば:

```
<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>
```

上記では select 文は次の構造に従った XML オブジェクトを提供します:

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

remapResults

この remapResults 属性は<statement>、 <select> と <procedure> にマップされたステートメントに適用されます。これはオプションの属性で初期値は false です。

クエリーが戻り列の変数セットを持つ時、remapResults 属性を true に設定するべきです。例えば、次のクエリーを考えてみましょう:

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

たとえ、このテーブルが常に同じだとしても、前の例では、カラムのリストは動的です。

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

この例では、このテーブルは異なるものにできます。select 句に*を使っているため、結果の列名も同様に異なるものにできます。また、動的なエレメントは列リストを一つのクエリー実行から次のものへと変化させます。

この resultset metadata が自明でないことを判断/決定するためのオーバーヘッドがあるために、iBATIS はクエリーが最後に実行された時に何が戻されたかを思い出します。上記の例に類似した状況下で問題を起こします。従って、全てのクエリーを実行する際に metadata introspection を行う可能性があります。

従って、この戻り列を変更するのであれば remapResults を true へ、そうでなければ remapResults を false に設定するように変更して metadata introspection のオーバーヘッドを避けることが出来ます。

resultSetType

SQl ステートメントの resultSetType を特定するために、いずれかを指定できます。:

- **FORWARD_ONLY**: カーソルは前方へのみ移動する可能性がある
- **SCROLL_INSENSITIVE**: カーソルはスクロール可能、しかし一般に他からの変更に対して敏感でない
- **SCROLL_SENSITIVE**: カーソルはスクロール可能、かつ一般に他からの変更に対して敏感

resultSetType は一般に必須ではなく、異なる JDBC ドライバは同じ resultSetType 設定を使用しても異なる振る舞いをする場合があることに注意しましょう。(例えば Oracle は **SCROLL_SENSITIVE** をサポートしない).

fetchSize

実行されるSQLステートメントに対してfetchSizeを設定します。これはJDBCドライバ にデータベースサーバへの最小の通信回数でプリフェッチするためのヒントを与えます。

timeout（iBATISバージョン2.2.0以降のみ）

ステートメントに対するJDBCクエリータイムアウトを設定します。ここで指定した値がSQLMapConfig.xmlファイルの“defaultStatementTimeout”で指定された値を上書きします。もしデフォルトのタイムアウトを指定したい、又は個別のステートメントに対してタイムアウトを指定しない場合は0に設定して下さい。この指定された値はステートメントが完了するまでドライバが待機する秒数です。全てのドライバが、この設定をサポートしないことに注意して下さい。

Parameter Maps と インラインParameters

既に見た通り、parameterMap はステートメントのパラメータに JavaBean プロパティをマッピングする責任があります。parameterMap が外部的の形をとる事は稀ですが、これらを理解することはインライン parameter を理解することを助けることでしょう。インライン parameter は、このセクションに続いてすぐに説明します。

```
<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>
```

[括弧] の中はオプションです。parameterMap 自体はステートメントから識別されるための ***id*** 属性だけが必須です。***class***属性はオプションですが強く推奨されています。ステートメントの parameterClass 属性と同様に、*class* 属性はフレームワークが入力されるパラメータを検証することを許可し、エンジンの性能を最適化します。

<parameter> エレメント

parameterMap はステートメントのパラメータへ直接マップするパラメータのマッピングをいくつでも含むことが出来ます。次の数節では *property* エレメントの属性について記述します。

property

parameter map の *property* 属性はマップされたステートメントへ渡されたパラメータオブジェクトの JavaBean プロパティ(get メソッド)の名前です。この名前はそのステートメント中で必要とされる回数に従い複数回使用されることがあります。(例.同じプロパティが 一つの SQL 更新ステートメントの set 句中で更新される場合、又は where 句中のキーとして使われる場合)

jdbcType

jdbcType 属性は、プロパティで設定されるパラメータの database column type を明示的に指定するために使われます。いくつかの JDBC ドライバは列型をドライバに明確に伝えないと一部の操作において列の型を特定することが出来ません。この完全な例は PreparedStatement.setNull(int parameterIndex, int sqlType) メソッドです。このメソッドは特定される型を必要とします。いくつかのドライバは単純に Types.OTHER か Types.NULL を渡すことでその型が暗黙的になることを許可するでしょう。しかし、この振る舞いは一貫性が無い上に、いくつかのドライバは特定される明確な型を必要とします。このような状況の為に、Data Mapper API は parameterMap プロパティエレメントの *jdbcType* 属性を使用しての指定を可能にします。

この属性は、通常カラムが null を許可する場合のみ必要です。しかしながら、この *jdbcType* 属性を使う理由は、date 型を明確に特定するという別の理由があります。Java は唯一の Date 型を持ちますが、多くの SQL データベースは多くの-普通は少なくとも 3 種類の型を持ちます。このため、あなたは、DATE 型のカラムに対して(例えば)DATETIME であることを明確に指定したいことがあるかもしれません。

JdbcType 属性は JDBC 型クラスの定数に適合する全ての文字列の値を設定できます。これらはいずれも設定することが出来ますが、いくつかの型はサポートされません。(例 blob 等)。この文書の後の節でフレームワークによってサポートされる型について解説します。

Note! ほとんどのドライバはnullを許可する列への型の指定だけを必要とします。従って、そのようなドライバではnullを許可する列への型を指定するだけで構いません。

Note! Oracleドライバを使う場合、その型の指定をせずに列にnull値を設定しようとすると“Invalid column type”エラーになることがあります。.

javaType

*javaType*属性は設定されるパラメータのJava property typeを明確に特定する為に使用されます。通常これはJavaBeanのプロパティからリフレクションを通じて導かれます、しかしMapとXMLのマッピングの様ないくつかのマッピングにおいてはフレームワークへ型が提供されません。
もしjavaTypeが未設定かつフレームワークが型を他に決定出来なかった場合、その型はObjectと見なされます。

typeName

typeName 属性は参照型またはユーザー定義型を明確に特定するときに使用されます。

javadocには以下のように述べられています:

typeName属性... “ユーザ定義か参照の出力パラメータに使用されるべきです。ユーザ定義型の例:STRUCT, DISTINCT, JAVA_OBJECT, 名前付けされた配列型。... ユーザ定義パラメータにはパラメータの完全修飾されたSQL型名、さらに参照パラメータは参照型の完全修飾された型名が与えられるべきです。型コードと 型名の情報を必要としないJDBCドライバはこれを無視する可能性があります。移植可能性のため、しかしながら、アプリケーションは常にこれらの値をユーザ定義または参照パラメータへ提供するべきです。これはユーザ定義と参照パラメータを意図したものですが、この属性はいずれかのJDBC型のパラメータを登録するために使われることもあります。もしこのパラメータがユーザ定義または参照の型を持たないとき、*typeName*パラメータは無視されます。“

＊斜体字 の単語はこの文書の文脈において解説を加えるために置き換えられました

nullValue

nullValue属性は任意の有効な(プロパティの型に基づく)値が設定されます。このnullValue属性は出力するnull値の変換を指定するために使われます。これはJavaBeanのプロパティにこの値が見つかった時、NULLがデータベースに書き込まれることを意味します(これは入力されるnull値の変換の反対の振舞いです)。これはあなたのアプリケーション内のnull値をサポートしない型(例.int,double,float 等)に対して、”魔法の”null値を利用可能にします。プロパティのこれらの型が合致するnull値(例.-9999)を含んでいるとき、この値に代わってNULLがデータベースへ書き込まれます。

resultMap

ストアドプロシージャ出力パラメータの値としてjava.sql.ResultSetのインスタンスを期待する場合に *resultMap*エレメントを指定します。これはiBATISがオブジェクトマッピングに正常な結果をセットをするのを可能にするでしょう。

mode

この*mode*属性はストアドプロシージャのモードを指定します。有効な値はIN,OUTまたはINOUTです。

typeHandler

typeHandler 属性はデフォルトの iBATIS 型システムの代わりにこのプロパティに使用されるカスタム型ハンドラを指定するために使われます。もし指定される時、この値は com.ibatis.sqlmap.engine.type.TypeHandler インターフェースまたは com.ibatis.sqlmap.client.extensions.TypeHandlerCallback インターフェースのどちらかを実装したクラスの完全修飾名とする必要があります。この値はこのプロパティに適用されるグローバル型ハンドラを上書きします。カスタム型ハンドラに関するこれ以上の詳細はこの文書で後に記述しています。

numericScale

(numericScale は iBATIS バージョン 2.2.0 以上でのみ有効です)

numericScale 属性は NUMERIC もしくは DECIMAL 型ストアドプロシージャ出力パラメータの桁数(小数点以下の桁数)を特定するために使われます。もし *mode* 属性へ OUT または INOUT を指定し、かつ *jdbcType* が DECIMAL または NUMERIC の時、同様に *numericScale* を指定するべきです。この属性に指定される値は 0 以上でなければいけません。

<parameterMap> の例

parameterMap の完全な構造をした例は次の通りです。

```
<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>
```

上記の例では、JavaBean プロパティの *id* と *description* は Mapped Statement *insertProduct* のパラメータに列挙された順番で値が設定されます。つまり、*id* は最初のパラメータ(?)が設定され、*description* は 2 番目になります。もし順序が逆の場合、XML は次のように見なされます:

```
<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>
```

Note! Parameter Map 名は必ずそれらが定義されている SQL Map XML ファイルに対してローカルです。別の SQL Map XML ファイルの Parameter Map を SQL Map の id(<sqlMap>ルートタグで設定)と Parameter Map の id の前に付けることによって参照することが出来ます。例えば、上記の Parameter Map を別のファイルから参照する時、参照への完全名は "Product.insert-product-param" となります。

インライン Parameter Maps

とても記述的ですが、parameterMap を宣言するための上記のシンタックスは、とても冗長です。インライン Parameter Maps は、冗長さを減らし、より記述的です。明示的に parameterMap を書く代わりに、こちらを使用することを勧めます。インライン Parameter Maps で、SQL に直接パラメータを定義できます。デフォルトでは、マップされたステートメントは、インライン

Parameter Maps を解析するために指定する明示的な parameterMap を持ちません。上記のサンプル(すなわち product)をインライン Parameter Maps で実装すると、このようになります：

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

下記のシンタックスをインライン Parameter Maps に指定することで型を宣言できます。：

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

下記のシンタックスをインライン Parameter Maps に指定することで型と null 値の置き換えを宣言できます。

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

Note! インライン Parameter Maps を使用している時、型を指定せずに null 値の置き換えを指定できません。解析順に指定する必要があります。

Note! null 値を完全に透過にしたいのであれば、このドキュメントで後述しているように Result Map に null 値の置き換えを指定しなければなりません。

Note! 多くの型の記述と null 値を置き換える必要があるのであれば、外部の parameterMap を記述することによりコードをクリーンに保つことができます。.

Inline Parameter Map のシンタックス

iBATIS は、インライン Parameter Maps において、2 つの異なるシンタックスをサポートしています。シンプルなシンタックスと、より複雑で完全なシンタックスです。

シンプルなシンタックスは、下記となります。

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

上記は、シンタックスの例です。propertyName エレメントは、パラメータオブジェクト（もしくは、String, Integer、などのシンプルな値であればパラメータオブジェクト自身の値）の名前です。
jdbc エレメントは、パラメータの JDBC 型を指定するために使用します。値は、java.sql.Types (VARCHAR, INTEGER など)にリストされている型の 1 つでなければなりません。
一般的に、jdbcType エレメントは、値が NULL か、DATE か TIME フィールド（TIMESTAMP フィールドとは対照的に)となりえる可能性があるのであれば必要です。nullValue エレメントは、上記の記述において NULL 値を置き換える値の指定に使用されます。jdbcType の指定なしに、nullValue を指定することができないことに注意してください。

このシンタックスは、正式な parameter map の詳細なオプションに、アクセスする必要がない大抵のシチュエーションにおいて、適切です（例えば、ストアドプロシージャを呼ぶとき)。

より詳細なシンタックスは下記のようになります：

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

“?”は、属性を指定するところです。

詳細なシンタックスは、正式なparameter mapの大部分にアクセスできます。propertyNameエレメントは、必須です。それ以外のエレメントは、全てオプショナルです。値は、propertyNameエレメントが先頭にしなければいけないことを除き、任意の順番で指定できます。異なる属性のために使用できる値は、正式なparameter mapを使用しているときに許可されているものです。このシンタックスについても注意があります。handler属性は、別名が登録済みのTypeHandlerのための別名を使用することです。このシンタックスを使用して、ストアドプロシージャを呼び出す例は、下記です:

```
<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>
```

プリミティブ型 Parameters

パラメータとして使用するために、JavaBeansを書くことは、常に必要でも、便利でもありません。それらの場合、パラメータに直接プリミティブ型ラッパーオブジェクト(String, Integer Dateなど)の使用を推奨します。例えば::

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

PRD_IDが、numeric型と仮定すると、このmapped statementにjava.lang.Integerを渡して呼び出すことができます。#value#パラメータは、Integerインスタンスの値で置き換えられます。**“value”という名前は、単なるプレースホルダーです。名前は何でもかまいません。ResultMap(下記で説明しています)は、結果においてもプリミティブ型をサポートしています。より詳細な情報は、下記のResultMapセクションとプログラミングData Mapper (API)を参照してください。**

プリミティブ型は、より簡潔なコードのために、別名づけられています。例えば、“java.lang.Integer”の場所に、“int”を使用できます。別名は、下記のタイトルのテーブルに記述しています。“サポートしているParameter MapとResult Mapの型”

Map Type Parameters

もし、JavaBeansクラスを書く必要がないか、不都合で単一のプリミティブ型パラメータを使用できない(例えば、複数のパラメータがある)のであれば、パラメータオブジェクトとして、Map(例えば、HashMap,TreeMap)を使用できます。

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

mapped statementの実装において、違いがないことに注意してください！上記の例において、もしステートメントを呼び出すのにMapインスタンスを渡したのであれば、Mapは、”catId”と”code”という名前のキーを含んでいなければなりません。それらのキーにより、適切な型の値が参照されます(上記の例では、IntegerとString)。(下記で説明している)ResultMapも、同様

にMap型をサポートしています。Map型をパラメータで使用する、より詳細な情報は、ResultMapセクションと、**プログラミングData Mapper（API)を参照してください。**

プリミティブ型は、より簡潔なコードのために、別名づけられています。例えば、“java.util.Map”の場所に、“map”を使用することがでいます。別名は、下記のタイトルのテーブルに記述しています。“サポートされているParameter MapとResult Mapの型”

置き換え文字列

iBATISは、常にSQLを実行するためにJDBC prepared statementを使用します。JDBC prepared statementは、”パラメータマーカ”を使用し、パラメータをサポートしています。parameter mapとインラインParameter Mapsともに、指定したパラメータの場所に、パラメータマーカを使用しiBATISにSQLを生成させます。例えば、以下のステートメントであれば：

```
select * from PRODUCT where PRD_ID = #value#
```

iBATISは、このSQL文字列のparepared statementを生成します：:

```
select * from PRODUCT where PRD_ID = ?
```

データベースは、ほとんどの場所にパラメータマーカを使用できます。しかし、SQLステートメントの部分全てではありません：

```
select * from ?
```

データベースは、このステートメントを事前に解析できません。なぜなら、何のテーブルが使用されるか知りえないためです。

```
select * from #tableName#
```

SQLExceptionを受け取ることになるでしょう。

これらの問題のいくつかを克服するために、iBATISは、ステートメントが事前に解析される前にSQLの文字列を置き換えるシンタックスを提供しています。動的なSQLステートメントを生成するために、この機能を使用できます。置き換え文字列のシンタックスの使用例は、下記です。

```
select * from $tableName$
```

このシンタックスで、iBATISはステートメントが解析される前にSQLの中にある“tableName”の値で置き換えます。この機能により、SQLステートメントの、どんな部分も置き換えることができます。.

Important Note 1: このサポートは、文字列で置き換えるだけの機能です。そのため、複雑なデータ型(DateまたはTimestamp)のためには、ふさわしくありません。

Important Note 2: テーブル名かカラムリストを、この機能で作り替えたいのであれば、SQL selectステートメントにおいて、remapResults=”true”を常に指定する必要があります。

Result Maps

Result Mapは、Data Mapperにおいて極めて重要なコンポーネントです。resultMapは、クエリマップステートメントの実行により生成されたResultSetのカラムとJavaBeansのプロパティをマッピングさせる役割を担います。resultMapの構造は、このようになります：

```
<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>
```

[ブラケット]の部分は、オプショナルです。resultMap 自身は、ステートメントが参照に使用するための id 属性を持ちます。resultMap も、クラスの完全な修飾名(すなわち フルパッケージ)か、タイプエイリアスの class 属性を持ちます。このクラスは、インスタンス化され、結果マッピングに基づいて代入されます。extends 属性は、基礎とする別の resultMap の名前を任意でセットできます。これは Java においてクラスを継承することに似ています。親となる resultMap の全てのプロパティは、子の resultMap の部分として含まれます。親となる resultMap のプロパティは、常に子の resultMap よりも前に挿入されます。そして、親となる resultMap は、子の resultMap よりも先に定義しておく必要があります。親/子 resultMap のクラスは、同じである必要はありません。全てにおいて関連する必要もありません。(それぞれ任意のクラスを使用することが可能です)

resultMap エレメントも、groupBy 属性をサポートしています。groupBy 属性は、返された結果セットの行を一意に識別するために使用される resultMap の中のプロパティのリストを指定するために使用されます。指定したプロパティと同じ値の行は、単一の結果として生成されます。(下記の例のための説明しているのを参照してください)

resultMap は、JavaBeans プロパティを ResultSet のカラムにマップする結果マッピングをいくつか含むことができます。それらのプロパティマッピングは、ドキュメントに定義されている順番で適用されます。関連するクラスは、各プロパティ、Map もしくは、XML に適切な get/set メソッドを持っている JavaBeans でなければなりません。

Note! カラムは、ResultMap において指定された順番で明確に読み込まれます(いくつかの貧弱な JDBC ドライバにおいて便利です)

次のいくつかのセクションで、result エレメントの属性を記述します。

property

property 属性は、マップされたステートメントによって返された結果オブジェクトの JavaBeans プロパティ(get メソッド)の名前です。名前は、結果を代入する数に依存し、1 回以上使用されます。

column

column 属性は、ResultSet の中のカラム名です。その値は、プロパティに代入するために使用されます。

columnIndex

オプショナル(最小)のパフォーマンスの拡張として、columnIndex 属性値は、ResultSet から JavaBeans プロパティに値を移しかえるためのカラムのインデックスです。これは、アプリケーションの 99%は、必要としないでしょう。そして、スピードのためにメンテナンス性と可読性を

犠牲にします。いくつかの JDBC ドライバは、理解しないためパフォーマンスのためにならないかもしれません。しかし、他の JDBC ドライバは、劇的にスピードアップするでしょう。

jdbcType

jdbcType 属性は、JavaBean プロパティに値を移しかえるために使用する ResultSet のデータベースカラムの型を明示的に指定するために使用されます。しかしながら、result maps は、null 値の時に、違いがありません。型を指定することは、Date プロパティのような特定のマッピングタイプのために役に立ちます。なぜならば、Java は、1 つの Date 値の型を持ち、SQL データベースは、date を（通常少なくても 3）多くの型を持っているかもしれません。いくつかのケースにおいて date の型を指定することは、dates(または、他の型）に正しくセットすることを確実にするために必要となります。同様に、String 型は、多分、VARCHAR, CHAR, CLOB によって代入されるかもしれません。そのため、型を指定することは、（ドライバ依存）他のケースにおいても必要となるかもしれません。

javaType

javaType 属性は、セットするプロパティの Java プロパティ型を明示的に指定するために使用されます。通常、JavaBeans のプロパティから、リフレクションを通じて渡されます。しかし、Map と XML マッピングのような特定のマッピングは、フレームワークに型を提供できません。もし、javatype がセットされなければフレームワークは型を決めることができないため、型を Object と仮定します。

nullValue

nullValue 属性は、データベースに NULL の場所に使われる値です。もし、ResultSet から NULL が読み込まれたら JavaBeans のプロパティに NULL の代わりに nullValue 属性に指定された値がセットされます。null 属性値は、どのような値もとりえますが、プロパティ、型に適切なものでなければなりません。

もし、データベースが NULLABLE カラムを持っていたら、下記のように result map で nullValue を指定して、アプリケーションで NULL を意味する定数にできます:

```
<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>
```

上記の例において。PRO_SUB_CODE から NULL を読み込むと、subCode プロパティには、-999 の値がセットされます。データベースにおいて NULLABLE カラムを意味するために Java クラスでプリミティブ型を使うことができます。update/insert においても、クエリーのために、これを動作させたいのであれば、(このドキュメントで以前に説明した)parameter map において nullValue を指定しなければいけないことを、覚えておいてください。

select

select 属性は、オブジェクトと複雑なプロパティタイプ（例.ユーザ定義)を自動的にロードする関係を記述するために使われます。ステートメントプロパティの値は、別の mapped statement の名前で、なければなりません。データベースカラム（column 属性）の値は、同じ property エレメントに関連する mapped statement のパラメータとして渡されるために定義されます。
プリミティブ型と複雑なプロパティマッピング/リレーションに関する詳しい情報は、このドキュメントで後述しています。

resultMap

resultMap 属性は、result mapping の中で、再利用できる入れ子の resultMap を記述するために使用します。これは、1 対 1 か 1 対 N のリレーションシップに使用できます。もし、1 対 N リレー

ションシップを期待しているのであれば、関連するプロパティは、コレクション(List, Set, Collection, など) でなければなりません。そして、iBATISが、どのように行をグルーピングすることの印として、resultMapエレメントにgroupBy属性を指定してください。1対1のリレーションシップにおいては、関連するプロパティは、いずれかの型となります。そして、groupBy属性を指定しなくても構いません。1対Nか、いくつかの1対1プロパティの時に、groupBy属性を使用する可能性があります。

typeHandler

typeHandler 属性は、デフォルト iBATIS タイプシステムの代わりに使用するカスタムの typeHandler を指定するために使用されます。
指定するのであれば、この値は、 com.ibatis.sqlmap.engine.type.TypeHandler インタフェースか、com.ibatis.sqlmap.client.extensions.TypeHandlerCallback インタフェースのどちらかを実装したクラスの完全修飾名となります。このドキュメントの後半に、カスタム typeHandler の、さらなる詳細を記述しています。

暗黙的な Result Maps

もし、明示的に定義されたresultMapを再利用する必要がないような、とてもシンプルな要求がある場合、mapped statementのresultClass属性をセットすることによって、暗黙的なresult mapを素早く指定する方法があります。仕組みは、JavaBeanの書き込み可能なプロパティと、result setが返すカラム名（もしくは、ラベル/エイリアス）が、マッチすることを保証しなくてはなりません。例えば、上記で記述したProductクラスを検討するのであれば、下記の暗黙的なresult mapを作成することが可能です。

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

上記のmapped statementは、resultClassを指定し、各カラムとマッチするProductクラスのJavaBeansプロパティのためのエイリアスを宣言します。これで、十分です。result mapは必要ありません。このトレードオフとして、最適なカラム型か、（通常は、必須ではありません）nullValue値の指定が、できなくなります。多くのデータベースは、大文字・小文字を区別しないため、暗黙的なresultmapも、ケースセンシティブではありません。そのため、JavaBeanが、firstNameとfirstnameという2つのプロパティを持っているのであれば、同一とみなされるため、暗黙的なresult mapを使用することはできません（JavaBeanクラスの設計で、潜在的な同一の性の問題がある場合もです）。さらに、resultClassを経由して自動マッピングで、関連付けるパフォーマンスのオーバーヘッドがあります。ResultSetMetaDataにアクセスすることは、いくつかの貧弱なJDBCドライバでは、遅くなることがあります。

プリミティブ Results（すなわち、String, Integer, Boolean）

追加で、JavaBeansに従ったクラスをサポートするために、Result Mapsは、String, Integer, Booleanなどのようなシンプルな Javaタイプラッパーに、都合良く代入できます。プリミティブオブジェクトのコレクションも、下記に記述しているAPI (queryForList()を参照)を使って、取得できます。覚えている1つのことだけで、プリミティブ型は、JavaBeanと、全く同じ方法でマップされます、プリミティブ型は、好みの名前（通常、"value"、または"val"）を付けることができまる1つのプロパティだけを持てます。例えば、もし、Productクラス全体の代わりに、全てのProductクラスの記述(Strings)のリストをロードしたいのであれば、マップは、このようになります：

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

```
</resultMap>
```

よりシンプルなアプローチは、mapped statement において、単に result クラスを使用することです（”as”キーワードを使用して、カラム名の別名”value”を作ります）

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

Map Results

Reulst Maps は、HashMap か、TreeMap のような Map インスタンスにも、都合良く代入することもできます。オブジェクトのコレクション(例. Map のリスト)は、下記で示す API(queryForList()参照)を使って取得することもできます。Map 型は、JavaBean と全く同じ方法でマップされます。JavaBean プロパティ設定の代わりに、Map のキーは、マップされたカラムと連動するためのリファレンスの値がセットされます。例えば、Product の値を素早く Map にロードしたいのであれば、下記のようにできます。

```
<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>
```

上記のサンプルにおいて、HashMap のインスタンスは作成され、Product データが代入されます。プロパティ名の属性(例.”id”)は、HashMap のキーとなります。そして、mapped columns の値は、HashMap において、値となります。

もちろん、Map 型を使用して、暗黙の result map を使うこともできます。例えば：

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

上記は、基本的に、返された ResultSet の意味する Map を返します。

複雑な Properties（すなわち．ユーザ定義クラスのプロパティ）

どのように適切なデータとクラスをロードするか知っている mapped statement と resultMap プロパティを関連させることによって、複雑な型（ユーザ定義のクラス）のプロパティを自動的に代入させることができます。データベースにおいて、一般的に、１対１か、１対多のリレーションシップ経由でデータを表現します。複雑なプロパティを持つクラスは、リレーションシップの”多”とプロパティ自身はリレーションシップの”１”の側を持ちます。下記の例を検討します。

```
<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>
```

上記のサンプルにおいて、Product インスタンスは、Category 型の category と呼ばれるプロパティを持ちます。category は、複雑なユーザ型です（例.ユーザ定義クラス）。JDBC は、category の代入することができません。別の mapped statement のプロパティマッピングに関連付けることにより、SQLMap のエンジンに、category に代入するための適切で、十分な情報を提供できます。上記の getProduct を実行すると、get-product-result ResultMap は、PRD_CAT_ID カラムの値を使って、getCategory を呼び出します。get-category-result ResultMap は、Category をインスタンス化し、代入します。Category インスタンス全体は取得され、Product の category プロパティにセットされます。

N+1 Select(1:1)を回避する

上記の解決した問題は、Product をロードするときに起こります、2 つの SQL ステートメントは、実際のところ、動作します（1 つは、Product のため、もう 1 つは、Category のため）。この問題は、単一の Product をロードするときには、ささいなことに見えます。しかし、10 個の Product をロードするとしたら、各 Product に関連する category をロードするために別のクエリーが実行されます。この結果、11 クエリーがトータルで実行されます。：Product のリストのために 1 つ、そして、各 Product に関連する Category のために 1 つです（N+1 もしくは、このケースでは、10+1=11）

解決策は、join と、分けた select ステートメントの代わりに、ネストした propery mappings を使用することです。これは、上記と同じシチュエーション（Products と Categories）の使用例です。

```
<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 バージョン 2.2.0 以上では、カラムの繰り返しの代わりに、1:1 クエリーにおいて result map の再利用をすることもできます。下記は使用例です。

```
<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>
```

Lazy Loading vs. Joins (1:1)

join を使用するとき注意すべき重要なことは、常に better ではないことです。もし、めったにアクセスすることがない関連するオブジェクトの場合(例.Product クラスの category プロパティ)、実際は、join と全ての category プロパティのロードが必要ない方が早いかもしれません。データベースの設計として、outer joins もしくは、nullable and/or non-indexed カラムを巻き込む場合には、特にそうです。それらのシチュエーションにおいては、lazy-loading と bytecode 拡張オプションを有効(SQL Map Config settings 参照)にした sub-select ソリューションを使用することがベターかもしれません。一般的には、関連するプロパティにアクセスしそうであれば join を使用し、そうではなく、lazy loading が、オプションでないときだけ、lazy loading を使用してください。

もし、どちらの方法にするか決めかねていたとしても心配しないでください。どちらの選択も重大なことでは、ありません。Java コードへの影響なく、いつでも選択を変更できます。上記の 2 つサンプルは、同じオブジェクトグラフの結果となり、そして、それらは、全く同じメソッドを使用してロードしています。唯一、検討しなければいけないことは、キャッシュを有効にするのであれば、分けた select(join ではなく)を使用する解決策は、戻ってきたインスタンスをキャッシュにいれることができます、しかし、大抵、問題の原因とはならないでしょう。(アプリケーションは、インスタンスレベルの同一性、すなわち、"=="に依存するべきではありません)。

複雑な Collection Properties

複雑なオブジェクトリストを表すプロパティをロードすることも可能です。データベースにおいて、データは、M:M リレーションシップ、もしくは、リストを含んでいるクラスが、" one side" 上にあり、" many side" のリストの中のオブジェクトを含む 1:M リレーションシップにより表現されます。オブジェクトのリストをロードするために、ステートメントに変更はありません (上記の例を参照)。違いは、SQL Map framework が、プロパティを List としてロードするビジネスオブジェクトが java.util.List か、java.util.Collection でなければならないことです。例えば、Category が、Product インスタンスの List を持っているのであれば、mapping は、このようになります (Category が、java.util.List 型の" productList" という List を持っていることを想定しています) 。

```
<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 Selects を回避する（1:M と M:N）

上記にある１対１の状況と同じですが、より大きな関心は、潜在的に大量のデータに関係することです。上記の状況における問題は、Category を読み込むとき、二つの SQL ステートメントが実際に走ってしまうことです（一つは Category のため、もう一つは関連する Product のリストのためです）。たったひとつの Category を読込むときであれば、ささいな問題ですが、仮に 10 個の Category を読み込むためのクエリーを走らせるときに関連する Product のリストを読み込むため別個にクエリーを投げたらどうでしょうか。Category のリストと、Category に関連する Product のリストの読み込みのために合計 11 個のクエリーを投げることになります（N+1、この場合１０＋１で１１）。この状況をより悪くするために、潜在的にデータの大きなリストを対処しています。

1:N & M:N ソリューション

iBATIS は N+1 selects solution を十分に解決します。これは例です:

```
<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>
```

呼び出す時...

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

...メインのクエリーが実行され、 結果が com.ibatis.example.Category 型の beans である *myList* 変数に格納されます。List の中の各オブジェクトは、”productList” プロパティを持ちます、”productList” にも、（”productResult” を使って、result map を子要素の beans に代入することを除いて）同じクエリーから、List が代入されます。そのようにして、一つのクエリを実行することで sub-lists を含むリストを取得できます。 そして、1 つのデータベースクエリーのみが実行されます。

重要なのは...

```
groupBy="id"
```

...属性で、そして...

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

...categoryResult result map の property mapping です。他に重要なことは productList プロパティのための result mapping は namespace を意識することです。単純に productResult だけでは動かないでしょう。

このアプローチを使えば、深かったり広かったりするいかなる N+1 問題も解決できます。

Important Notes： queryForPaginatedList() API を用いて groupBy することは未定義であり、期待する結果と違う結果を取得してしまうかもしれません。これら二つの考えを混ぜ合わせることはしないでください。groupBy を使うときは、常に queryForList か queryForObject メソッドを使うようにしてください。

ネストしたプロパティは、java.util.Collection のいずれかの実装にできます。プロパティのアクセッサはシンプルで内部の属性への単純アクセスを提供しているだけであるべきです。iBATIS はプロパティへの get メソッドを繰り返し呼び出しますし、result set の処理においてプロパティの add()メソッドを呼び出します。常に普通の getter や setter であるようにしてください（内部の配列を List にラップしようとするように）……iBATIS が、失敗する原因になりえます。set メソッドを一回呼ぶだけですむオブジェクトへのバッチのようなものといった誤解が iBATIS に対してよくあります。もし、get メソッドが null を返すような場合 iBATIS は set メソッドを呼ぶだけです。iBATIS は、プロパティのデフォルトの実装を作り、結果オブジェクトに作った新しいオブジェクトを設定しようとします。新しく作られたオブジェクトは、常に空になるでしょう。iBATIS は、property プロパティを得るための get メソッドと add メソッドを呼び出します。

Lazy Loading vs. Joins (1:M and M:N)

前の 1:1 と同じように、join を使うのは 常に良いとは限りません。大量のデータのための独立したプロパティのためのコレクションにおいては、いっそう注意が必要です。関連したオブジェクトに、アクセスするのが稀である場合（例：Category クラスの productList プロパティ）、join を使わないで、produst のリストの不必要な読み込みを避けた方が早いはずです。外部結合を実行できるように、デザインされたデータベースあるいは、nullable かつ/あるいはインデックス化されていないカラムの読み込みで、特に顕著です。このような場合には、遅延読み込みを用いた副問い合わせを用いた解決や、bytecode enhancement オプション（SQL Map Config の設定を参照）を使用できるようにした方が良いと思われます。一般的な経験則は以下のとおりです：join は関連したプロパティにアクセスするだろうと思われる場合に使ってください。そうでない場合、lazy loading がオプションで無い場合は、常にそれだけを使ってください。

前述のように、どちらに決めようか迷ったとしても問題ないです。どちらに決めたとしても、Java コードの実装の変更無しに、いつでも変更できます。上記の二つの例題は全く同じオブジェクトのグラフの結果で、全く同じメソッドを使ってロードされました。キャッシュを有効にして個別に select （join ではなく）を発行することでキャッシュされたインスタンスを受け取るかどうかという点だけ考えます。しかし、しばしば、それは問題を引き起こさないでしょう（==のような、アプリケーションは、インスタンスレベルの同一性に依存するべきではありません）。

Composite キー、または Multiple Complex パラメータプロパティ

おそらく気が付かれたと思いますが、上記の例題は resultMap のカラム属性により指定された単一のキーしかありません。これは単一のカラムだけが related mapped statement と関連付けられることを示しているかのようですが、複数のカラムを関連付けることができる別の記述方法もあります。これは複合キーの関連がある場合や、*#value#*.以外のいくつかの名前パラメータを使いたいと思っているだけのときなどに役立ちます。{param1=column1, param2=column2, …, paramN=columnN}のようにもう一つのカラム属性の記法はシンプルです。下記にある Customer ID と Order ID が主キーの PAYMENT テーブルについて考えてみましょう。

```
<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>
```

パラメータと同じ順番であれば、カラム名を指定するだけで定義することもできます。例：

```
        {ORD_ID, ORD_CST_ID}
```

通常、可読性やメンテナンス性への影響に比べて、この記法によるパフォーマンスの向上は微々たるものです。

Important! 現在の SQL Map フレームワークは、循環的なリレーションシップを自動的に解決しません。 親子関係（木構造）を実装する際には、このことに気をつけてください。簡単な回避策として、親オブジェクトをロードできない場合のための、別の result map を定義するか、あるいは、「N+1 を避ける」ソリューションのような join を使うようなことが考えられます。

Note! いくつかの JDBC ドライバ（例：PointBase 組み込みの）は、一つのコネクションごとに複数の ResultSet を同時に扱えません。そのようなドライバでは、SQL Map エンジンが複数 ResultSet コネクションを必要とするため、complex object は動きません。繰り返しになりますが、代わりに join を使ってください。

Note! Result Map の名前はそれらが定義されている SQL Map XML ファイルの中の一部であるようにしてください。SQL Map（<sqlMap> ルートタグ内にある）の名前と一緒に Result Map の名前のプレフィクシングにより別の SQL Map XML ファイルにある Result Map を参照できます。

Microsoft SQL Server 2000 の JDBC ドライバを使うのでれば、マニュアルトランザクションモードで複数ステートメントを実行するために、コネクション URL に、SelectMethod=Cursor を付け加える必要があります(参照 MS Knowledge Base Article 313181: http://support.microsoft.com/default.aspx?scid=kb%3Ben-us%3B313181)。
If you are using the Microsoft SQL Server 2000 Driver for JDBC you may need to add SelectMethod=Cursor to the connection url in order to execute multiple statements while in manual transaction mode (see MS Knowledge Base Article 313181: http://support.microsoft.com/default.aspx?scid=kb%3Ben-us%3B313181).

サポートしている Parameter Maps と Result Maps の型

iBATIS フレームワークのパラメータと result でサポートする Java の型は以下のとおりです：
The Java types supported by the iBATIS framework for parameters and results are as follows:

Java Type	JavaBean/Map Property Mapping	Result Class / Parameter Class***	Type Alias**
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

大文字・小文字を区別する version 2.2.0 の 型のエイリアスのケースに留意ください。「string」、「String」、「StrinG」のようなエイリアスはすべて java.lang.String 型にマップされます。

** java.sql. date 型の仕様はおすすめできません。代わりに java.util.Date 使うのがベストプラクティスです。*

*** .parameter や result の定義の際、完全なクラス名の変わりにエイリアスが使用できます。*

******* iBATIS のデータベースレイヤーが完全なオブジェクト指向であるために int、boolean、float のようなプリミティブ型は直接そのままの型としてサポートされていません。そのためすべてのパラメータや result の最上位レベルは Object 型でなければなりません。JDK1.5 のオートボクシング機能はプリミティブを上手く扱えるでしょう。

Custom Type Handler の作成

TypeHandler や TypeHandlerCallback インターフェスを使うことで iBATIS の型サポートを拡張できます。TypeHandlerCallback は簡単に実装できるので、より複雑な TypeHandler インターフェイスを使うよりも、推奨します。自身の type handler を作るために TypeHandlerCallback の実装クラスを作る必要があります。自作の type handler を使うことでサポートしていない型を扱えるようにしたり、別の方法で型のハンドルをサポートできるように、フレームワークを拡張できます。例えば、（例：Oracle）BLOB プロパティのために自作の type handler を使ったり、よくある 0/1 の代わりに「Y」や「N」を使って boolean を扱えます。

これは「Yes」と「No」を使う boolean handler の簡単な例です：

```
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);
    }
  }
```

iBATIS で、それら型を使うために宣言を、sqlMapConfig.xml に以下のように書いてください：

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

以降、iBATIS は定義された、ある特定の型handler callback を使っての公式の Java 型と jdbc 型間の変換を知ることができます。任意ですが、<result> mapping や explicit やインラインの parameter map に type handler を定義することで、個別のプロパティの type handler を定義す

ることもできます。

Mapped Statement Results のキャッシュ

statement タグ（上記参照のこと）で cacheModel パラメータを定義することで Query Mapped Statement からの結果を、簡単にキャッシュすることが出来ます。キャッシュモデルは SQL map の中で定義されます。次のような cacheModel エレメントを使って、キャッシュモデルは定義されます：

```
<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>
```

上記のキャッシュモデルは、「product-cache」と名づけられた LRU 実装のキャッシュのインスタンスを生成します。type 属性の値は、完全クラス名か実装に含まれるエイリアスの一つです（下記参照）。キャッシュモデルの中で定義されている flush エレメントに基づきこのキャッシュは、毎 24 時間ごとにフラッシュされます。hours, minutes, seconds or milliseconds の単位を使って flush interval element は一つだけ定義できます。さらに、insertProduct, updateProduct, あるいは、 deleteProduct の mapped statement が実行される度に、キャッシュは完全にフラッシュされます。「flush on execute」エレメントは一つのキャッシュにいくつでも定義できます。いくつかのキャッシュ実装では、上記の例にある「cache-size」プロパティのような追加プロパティをが必要かもしれません。LRU キャッシュの場合、size はキャッシュに入れることが出来るエントリの数を決定します。一度、キャッシュモデルを決めると、mapped statement で定義したキャッシュモデルを使うことが出来ます。例：

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

Read-Only vs. Read/Write

フレームワークは、読み込み専用と読み書きの両方をサポートします。読み込み専用キャッシュは、すべてのユーザで共有されるので、大幅なパフォーマンス向上を提供できます。しかしながら、読み込み専用キャッシュから読み込まれるオブジェクトは修正するべきではありません。その代わりに、更新のためには新しいオブジェクトをデータベース（あるいは読み書き両用キャッシュ）から読み出すべきです。あるいは、オブジェクトを検索と更新のために使うという意図がある場合は、読み書き両用キャッシュを推奨します（または、必須です）。読み込み専用キャッシュを使うためには cache model element に readOnly=" true" をセットしてください。読み書き両用キャッシュであれば readOnly=" false" をセットしてください。read-only はデフォルトでは true です。

Serializable Read/Write Caches

上記のようなセッション単位のキャッシュは、アプリケーション全体のパフォーマンスには、それほど利益がないということはご理解いただけるかと思います。serializable read/write cache という、もう 1 つの読み書き両用キャッシュがありますが、これであれば（セッション単位ではどうかわかりませんが）アプリケーション全体でパフォーマンス向上できます。このキャッシュは、各々のセッションでキャッシュから異なるインスタンス（コピーですが）を返します。そのため、各々のセッションで安全に更新できます。キャッシュから同じインスタンスが帰ってくることを期待するのが普通かもしれませんが、この場合、異なるインスタンスを受け取るということの意味の違いに気づいていただきたいです。それと、serializable cache にストアされるので、全てのオブジェクトは serializable でなければいけません。ということは、遅延プロクシはシリアライズできないということから、遅延読み込みと serializable cache を同時に使う

ことは難しいことを意味します。遅延読み込みとテーブルの join を同時に使う方法を試してみることを推奨します。serializable cache を使うためには readOnly=”false”、serialize=”true”にしてください。デフォルトのキャッシュモデルは読み込み専用でシリアライズではありません。読み込み専用キャッシュは、シリアル化できるべきではありません（メリットが無いので）。

Cache Types

キャッシュモデルはいろいろ拡張できます。（上で説明したように）キャッシュの実装は cacheModel エレメントの type 属性で定義されます。クラス名は CacheController インターフェイスを実装しているか、下にあるような４つのエイリアスのうちのひとつである必要があります。詳細な設定パラメータは、cacheModel の本体に含まれる property エレメントを通じて実装を渡すことができます。それは：

“MEMORY” (com.ibatis.db.sqlmap.cache.memory.MemoryCacheController)

MEMORY キャッシュ実装は参照型をキャッシュの振る舞いの管理に使います。このため、ガベージコレクターがなにをキャッシュするかあるいはそうしないかを的確に決定できます。MEMORY キャッシュはオブジェクトの再利用の identifiable pattern を取らない場合や、アプリケーションのあるメモリが不十分である場合によい選択肢です。

MEMORY 実装はこのように定義されます：

```
<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>
```

MEMORY キャッシュ実装はひとつのプロパティだけ認識します。reference-type プロパティには STRONG、 SOFT あるいは WEAK という値を設定する必要があります。これらの値は JVM で使われるメモリ参照型と同じです。

下の表には MEMORY キャッシュで使用できる参照型の相違点が書いてあります。参照型をより理解するために、JDK のドキュメントの java.lang.ref の reachability について参照ください。

WEAK *(default)*	この参照型がほとんどの場合おそらく最も良い選択であり、reference-type を定義しない場合のデフォルトです。これは、一般的な結果のパフォーマンスを向上させますが、他のオブジェクトで利用する場合に完全にメモリを開放します、この場合結果は使われていないわけと仮定します。
SOFT	この参照型は、結果が現在使われていないで他のオブジェクトにメモリが必要な場合にメモリを使い果たす可能性を減らします。しかし、これは最も積極的な参照型ではないのでメモリを割り当て続けて他の重要なオブジェクトに割り当てられないかもしれません。
STRONG	この参照型はキャッシュが（タイムインターバルや実行時のフラッシュなど）はっきりとフラッシュされるまで結果が保持されつづけることを保証します。これは、以下のとおりである結果に理想的です、つまり、１）とても小さく、２）完全に静的で、３）とても頻繁に使用される。この特定のクエリのためのパフォーマンスには利点があります。欠点は、結果が必要である限りメモリが使われつづけることであり、他の（あるいはより重要な）オブジェクトにメモリを割り当てられないことです。

“LRU” (com.ibatis.db.sqlmap.cache.lru.LruCacheController)

LRU キャッシュ実装は、Least Recently Used（最長未使用時間）アルゴリズムを使って、どのようにオブジェクトをキャッシュから除くのかを決めます。キャッシュが満杯を超えた場合、最も長い時間使われていないオブジェクトをキャッシュから除きます。このように、しばしば参照されているオブジェクトがあれば、それは削除される可能性が最小の状態でキャッシュにとどまります。LRU キャッシュは、特定のオブジェクトを一人以上がより長い期間、良く使われるようなアプリケーション（例.ページ付きリストを前後する、人気のサーチキーなど）のために、良い選択です。

LRU 実装は以下のように定義されます：

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

LRU キャッシュ実装は一つのプロパティしか認識しません。これは、size プロパティという名前で、一度にキャッシュに保持するオブジェクトの最大数を表す ingeger の値を設定します。この場合のオブジェクトは、単なる String のインスタンスだったり、あるいは JavaBeans の ArrayList だったりすることに注意してください。あまりキャッシュを大きくしすぎることは out of memory のリスクがあることに気をつけてください。

“FIFO” (com.ibatis.db.sqlmap.cache.fifo.FifoCacheController)

FIFO キャッシュ実装は First In First Out アルゴリズムを使ってキャッシュからどのオブジェクトを自動的に除くかを決定します。キャッシュが溢れた場合、最も古いオブジェクトがキャッシュから除かれます。FIFO は連続して短い期間に参照されるが、少し後には必要が無くなるようなクエリを使う場合に有効です。

FIFO キャッシュ実装は以下のように定義されます：

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

FIFO キャッシュ実装は一つのプロパティのみ認識します。これは、size プロパティという名前で、一度にキャッシュに保持するオブジェクトの最大数を表す ingeger の値を設定します。この場合のオブジェクトは、単なる String のインスタンスだったり、あるいは JavaBeans の ArrayList だったりすることにご注意ください。あまりキャッシュを大きくしすぎることは out of memory のリスクがあることに気をつけてください。

“OSCACHE” (com.ibatis.db.sqlmap.cache.oscache.OSCacheController)

OSCACHE キャッシュ実装は OSCache 2.0 キャッシュエンジンのためのプラグインです。かなり自在に構成したり設定できたりします。

OSCACHE キャッシュ実装は以下のように定義されます:

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

OSCACHE キャッシュ実装は設定にプロパティは使用しません。その代わり、OSCache インスタンスをクラスパスのルートにある *oscache.properties* ファイルを使って設定されるのが一般的です。ファイルには（上で説明したような）アルゴリズムやキャッシュのサイズや永続化の方法（メモリかファイルかあるいは他の何か）やクラスタリングなどを設定できます。

詳しくは OSCache のドキュメントを参照ください。OSCache とそのドキュメントは以下の Open Symphony のサイトにあります。

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

動的な Mapped Statements

JDBC を直接使っていると動的な SQL がいつも問題になります。(パラメータとカラムが、全て含まれている) パラメータの値だけでなく、変化する SQL ステート面とを動作させることは、通常とても困難です。if-else 条件分岐が、めちゃくちゃなステートメントや、恐ろしい文字列の連結などを用いるのが典型的な解決法です。求められる結果は、しばしばクエリーです。例のオブジェクトと似ているオブジェクトを見つけるためにクエリーを構築できます。
Data Mapper API はどんな mapped statement element にも適用することが出来る比較的エレガントな解決方法を適用します。これは、シンプルな例です：

```
<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>
```

上の例は、パラメータ Bean の「id」 プロパティの状態によって作られる二つの可変のステートメントです。パタメータが 0 より大きければ、ステートメントは以下のように作られます。

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

あるいは、パラメータが 0 以下であれば、ステートメントは下記のようになります。

```
select * from ACCOUNT
```

もっと複雑な状況でなければ、ありがたみが分からないかもしれません。例えば、下のような若干、複雑な例をあげてみます：

```
<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>
```

状況により、上記の動的ステートメントによって 16 もの異なった SQL クエリーが作られます。if-else 構文と文字列連結のコードなら、より怠慢に数百行以上のコードを必要とするでしょう

SQL の動的な部分を条件タグで囲むことで、簡単に動的ステートメントを使うことが出来ます。例えば：

```
<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>
```

上のステートメントでは、 <dynamic>エレメントは、動的な SQL の境界を示しています。dynamic エレメント（つまり WHERE)は、任意で、条件を追加しないステートメントに、prepend が含まれているケースにおいて、prepend を管理する方法を提供します。ステートメントセクションでは、ステートメントに含める動的なＳＱＬコードを決定する条件のエレメントを好きなだけ含めることが出来ます。条件エレメントはクエリーに入れられたパラメータオブジェクトの条件に基づき動作します。dynamic element と条件エレメントは「prepend」属性を持ちます。prepend 属性は、必要であれば親のエレメントに追加することでオーバーライドすることが出来るコードの一部です。上記の例では、「where」prepend は初めの true 条件 prepend としてオーバーライドされています。これはＳＱＬステートメントが、適切に構築されるために必要なことです。例えば、初めが true の状態の場合、ＡＮＤは必要なく、ステートメントに、もし、ＡＮＤがあるとするとステートメントが壊れてしまいます。次のセクションでは、バイナリ条件や単項 演算条件やイテレートなどを含む様々な種類のエレメントについて、記述しています。

Dynamic エレメント

dynamic タグは、他の動的な sql エレメントのラップと、全体の前に付与するか、開始するか、終了させ結果として、本体部分に結合する方法を提供するシンプルなタグです。このタグの removeFirstPrepend 属性の機能を使用しているとき、始めの本体が生成した入れ子のタグが持っている前に付与されている内容(prepend) は、削除されます。

Dynamic 属性：

prepend - ステートメントの前に追加する上書き可能な SQL の部分(オプショナル)
open - 全結果の本体部分を開始する文字列（オプショナル）
close - 全結果の本体部分を終了する文字列（オプショナル）

<dynamic>	prepent, open と close 全体を考慮に入れるラッパータグ

バイナリ条件エレメント

バイナリ条件エレメントは、プロパティ値を決まっている値か、別の値かを比較するエレメントです。もし、結果が true であれば SQL クエリーの本体に含まれます。

バイナリ条件属性：

prepend - ステートメントの前に追加する上書き可能な SQL の部分(オプショナル)
property - 比較されるプロパティ（必須）
compareProperty - 比較される他のプロパティ（必須 または compareValue）
compareValue - 比較される値（必須 または compareProperty）
removeFirstPrepend - 初めの入れ子のタグが生成するタグの prepend を削除する(true|false オプショナル)

open – 全結果の本体部分を開始する文字列（オプショナル）
close – 全結果の本体部分を終了する文字列（オプショナル）

<isEqual>	プロパティと値、または別のプロパティと同一をチェックします。
<isNotEqual>	プロパティと値、または別のプロパティと同一ではないことをチェックします。
<isGreaterThan>	プロパティが、値または別のプロパティより大きいことをチェックします。
<isGreaterEqual>	プロパティが、値または別のプロパティ以上かどうかをチェックします。
<isLessThan>	プロパティが、値または別のプロパティより小さいことをチェックします。
<isLessEqual>	プロパティが、値または別のプロパティ以下かどうかをチェックします。使用例： **<isLessEqual prepend=" AND" property=" age" compareValue=" 18" > ADOLESCENT = 'TRUE' </isLessEqual>**

単項条件エレメント

単項条件エレメントは、プロパティの状態が指定の条件かを確認します。

単項条件属性:
prepend – ステートメントの前に追加する上書き可能なSQLの部分(オプショナル)
property – 比較されるプロパティ（必須）
removeFirstPrepend – 初めの入れ子のタグが生成するタグのprependを削除する(true|false オプショナル)
open – 全結果の本体部分を開始する文字列（オプショナル）
close – 全結果の本体部分を終了する文字列（オプショナル）

<isPropertyAvailable>	プロパティが利用可能か確認します(すなわち、パラメータのプロパティであること)
<isNotPropertyAvailable>	プロパティが利用不可能か確認します(例 パラメータのプロパティではないこと)
<isNull>	プロパティがnullかどうか確認します。
<isNotNull>	プロパティがnot nullかどうか確認します。
<isEmpty>	Collection,StringかString.valueOfプロパティの値がnullか空(“”かsize()＜1)を見て確認します。
<isNotEmpty>	Collection,StringかString.valueOfプロパティの値がnot nullでかつ、空ではないこと(“”かsize()＜1)を見て確認します。 使用例： **<isNotEmpty prepend=" AND" property=" firstName" > FIRST_NAME=#firstName# </isNotEmpty>**

他のエレメント

パラメータプレゼント：それらのエレメントは、パラメータオブジェクトが存在するか確認します。

パラメータプレゼント属性：
prepend - ステートメントの前に追加する上書き可能なSQLの部分(オプショナル)
removeFirstPrepend - 初めの入れ子のタグが生成するタグのprependを削除する(true|false オプショナル)
open - 全結果の本体部分を開始する文字列（オプショナル）
close - 全結果の本体部分を終了する文字列（オプショナル）

<isParameterPresent>	パラメータオブジェクトが提供されている(not null)か確認します。
<isNotParameterPresent>	パラメータオブジェクトが提供されていない(null)か確認します。 使用例： **<isNotParameterPresent prepend=" AND" > EMPLOYEE_TYPE = 'DEFAULT' </isNotParameterPresent>**

Iterate: このタグは、コレクションを繰り返し、そしてListにある各アイテムのために本体部分を繰り返します。

Iterate 属性：
prepend - ステートメントの前に追加する上書き可能なSQLの部分(オプショナル)
property - 繰り返されるjava.util.Collectionか java.util.Iteratorか配列のプロパティ(オプショナル - プロパティが指定されていなければ、パラメータオブジェクトはコレクションとみなされます。より詳細は、下記を参照してください。)
open - 全結果の繰り返しブロック部分を開始する文字列、ブラケットのために便利（オプショナル）
close - 全結果の繰り返しブロック部分を終了する文字列、ブラケットのために便利（オプショナル）
conjunction - ANDとORのために便利な各繰り返しの間に、適用される文字列(オプショナル)
removeFirstPrepend - 初めの入れ子のタグが生成するタグのprependを削除する(true|false オプショナル - より詳細な情報は下記を参照)

<iterate>	java.util.Collection か java.util.Iterator か配列のプロパティ分、繰り返します。 使用例： <iterate prepend=" AND" property=" userNameList" open=" (" close=")" conjunction=" OR" > username=#userNameList[]# </iterate> マップされたステートメントに、パラメータとしてコレクションが渡された時に、iterate を使用することもできます。 使用例： <iterate prepend=" AND" open=" (" close=")" conjunction=" OR" > username=#[]# </iterate> Note: iterate エレメントを使用した時に、プロパティ名の最後に[]を含めることは、とても重要です。[]を付与することにより、単にコレクションを String として出力するのではなく、このオブジェクトをコレクションと見分けます。

より進んだ<iterate>タグの使用方法：

１つ目の例であることを注意してください。" userNameList[]" は、リストの中の現在のアイテムを参照する操作になります。このようにリストアイテムからプロパティを選択するために、このオペレータを使用できます。

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

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

</iterate>
```

iBATIS version 2.2.0 において、iterate タグは、ネストして複雑な条件を作成することもできます。これは、使用例です。

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

これは、オブジェクトのリストである“orConditions”プロパティをパラメータオブジェクトが保持していると想定しています。そして、List の中の各オブジェクトは、“orConditions”と呼

ばれるListプロパティを含みます。そのため、パラメータオブジェクトにListの中に、Listを持っています。

“orConditions[].conditions[].condition”というフレーズに注意してください。これは、外側のループの現在のエレメントのconditionsプロパティの内側のリストの現在のエレメントからconditionsプロパティを取得することを意味します。iterateタグに、ネストの深さに対する制限は、ありません。また、“current item”オペレータは、他の動的タグの入力として使用できます。

<iterate>タグで、removeFirstPrependは、他のタグとは、幾分異なります。もし、removeFirstPrependにtrueを指定すると、始めの入れ子のタグが生成した属性は、削除されます。これは、ループ全体で一度だけ発生します。大抵の状況下において、正しい振舞です。

いくつかの状況では、ループの各行の繰り返しごとに、removeFirstPrepend機能を動作させたいかもしれません。このケースにおいては、removeFirstPrependの値にiterateを指定します。この機能は、iBATIS version 2.2.0以降で利用可能です。

シンプルな動的SQLエレメント

Dynamic Mapped Statement APIの全てのパワーについて、上記で説明していますが、時々シンプルで、小さなSQLの部分を動的にする必要があります。このために、SQLステートメントと、ステートメントは、動的なorder by句か動的なカラム、または、SQLステートメントのいずれかの部品のselectの実装を助けるために、シンプルな動的SQLのエレメントを含むことができます。コンセプトは、インラインParameter Mapsの動作と、とても似ています。しかし、シンタックスが、わずかに異なります。下記のサンプルを検討してみましょう。

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

上記のサンプルにおいて、*preferredOrder*動的エレメントは、(parameter mapのように)パラメータオブジェクトの*preferredOrder*の値によって置き換えられます。異なるのは、単にパラメータに値を設定するよりも、よりシリアスにSQLステートメント自身を本質的に変更するところです。シンプルな動的SQLによって、もたらされた勘違いは、セキュリティ、パフォーマンスと安定性にたいするリスクを招きます。シンプルな動的SQLが適切に使用されているかどうか保証するために、十分注意してください。また、データベースの詳細が、ビジネスオブジェクトモデルを侵す可能性があります。設計を意識してください。例えば、order by句の最後を意味するカラム名がビジネスオブジェクトのプロパティとしてか、JSPページのフィールドの値とすることを望まないでください。

シンプルな動的エレメントは、ステートメントの中に含めることが可能です。そして、SQLステートメント自身を、変更する必要がある時に役立ちます。例えば：

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

上記の例において、パラメータオブジェクトは、$operator$トークンを置き換えるために使用されます。もし、operatorプロパティがlikeであって、*descriptionプロパティが、'%dog%'であればSQLステートメントが、このようになります。*

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

Data Mapperによるプログラミング：The API

SqlMapClient APIは、シンプルで最小のはずです。4つの主要な機能を使用する手段をプログラマに提供します。：SQL Mapの設定、更新SQL（insertとdeleteを含む)の実行、単一のオブジェクトのためのクエリの実行、そして、リストオブジェクトのためのクエリの実行

設定

SQL Mapを設定することは、（上記で既に説明したように）SQL Map XML定義ファイルとSQL Map設定ファイルを1度作成するだけのことです。SqlMapClient インスタンスは、SqlMapClientBuilderにより構築されます。 SqlMapClientBuilderクラスは、 SqlMapClientインスタンスを構築して返すbuildSqlMap()というstaticメソッドをオーバーロードしました。buildSqlMap()メソッドは、 Readerか InputStreamを読み取ることができます。そして、設定ファイルのプロパティに、値をセットするために使用することができる Propertiesオブジェクトを任意で受け入れることができます。これは、それらのメソッドのいくつかの使用例です：

```
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);
```

それらのメソッドの差異は、主にキャラクタエンコーディングと国際化の問題に関係しています。より詳細については、国際化のセクションを参照してください。

トランザクション

デフォルトでは、SqlMapClientインスタンスにある、いずれかのメソッドを実行すると（例.queryForObject()、もしくは、insert()）自動コミットか、自動ロールバックが行われます。これは、いずれかの各メソッドを単一の作業として、実行されることを意味します。シンプルな意図ですが、単一の作業として(例.グループとして成功か、失敗)複数のステートメントを実行しなければいけないのであれば、良い考えではありません。トランザクションとして実行しないといけません。

もし、(SQL Map設定ファイルの設定により)グローバルトランザクションを使用しているのであれば、自動コミットを使用することができ、いつでも作業単位をまとめることができます。しかしながら、コネクションプールとデータベース初期化のトラフィックを減らすのと同様に、パフォーマンスとトランザクションの境界について、検討するべきです。

下記のSqlMapClientインタフェースは、トランザクションの境界を宣言するメソッドを持っています。下記のSqlMapClientインタフェースのメソッドを使用して、トランザクションの開始、コミット、もしくは、ロールバックをできます。

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

トランザクションを開始させることにより、コネクションプールからコネクションを取得し、SQLクエリーと更新を受け取るために、コネクションを開きます。

トランザクションの使用例は、下記です:

```
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()はエラーに関わらずコールされることに注意してください。これは、クリーンアップ処理を確実にする重要なステップとなります。規則は次のとおりで、startTransaction()がコールされた場合、確実にendTransaction()がコールされます。（たとえ、コミットされていない場合でも同様です）

Note! トランザクションは、入れ子にできません。同一スレッド内で、commit()やrollback()をコールする前に、一度以上、startTransaction()をコールすると、例外がスローされます。要するに、各スレッド内は、SqlMapClientインスタンスに対して、一つのトランザクションだけ持つことができます。

Note! SqlMapClientのトランザクションは、JavaのThreadLocalで保持しているトランザクションオブジェクトを利用します。これは、スレッド同士でお互い別のstartTransaction()をコールでき、ユニークなコネクションをトランザクションで利用できることを意味しています。*データソースへ、コネクションを戻す唯一の方法（コネクションのクローズなど）は、commitTransaction ()またはendTransactionを呼ぶことです。そうしないと、プールを使い果たしてしまい、コネクションがロックされてしまいます。*

自動的なトランザクション

明示的なトランザクションの使用を、特に推奨していますが、(一般的に、読み込み専用)シンプルな要求のために使用できるシンプルなセマンティックがあります。
もし、startTransaction()、commitTransaction()、およびendTransaction()を使って、明示的にトランザクションの境界を策定しない場合、それらは全て、いつでも上記で示されたトランザクションブロックの外側で、自動的に呼ばれます。

```
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();
    }
}
```

Note！！ 自動トランザクションは魅力的ですが、使用に関しては非常に注意してください。なぜならば、複数のデータベースへのアップデートが必要な場合に問題が起こるからです。上の例で、2回目の sqlMap.update("updateItem", item)の処理に失敗した場合、description には"TX1"というデータで更新されたままになります。(すなわち、これはトランザクショナルな振舞いではありません)

グローバルトランザクション(分散トランザクション)

Data Mapper フレームワークはグローバルトランザクションをサポートしています。
分散トランザクションとして知られているグローバルトランザクションは、複数のデータベース(または、JTA に準拠したリソース)を同じ作業単位で更新できます。
(すなわち、複数のデータソースへの更新を１つのグループとして、成功か、失敗させることができます)。.

外部/プログラム的なグローバルトランザクション

グローバルトランザクションの外部的な管理を、プログラム的(手作業)か、とても一般的な EJB のように、別のフレームワークの実装のどちらかを選択できます。EJB を使用し、EJB デプロイメントデスクプリタの中にトランザクション（の境界ををセットする）の境界を記述できます。どのように完了させるという、さらに進んだ説明は、このドキュメントの範囲外です。外部、またはプログラム的なグローバルトランザクションのサポートを有効にするために、SQLMap 設定ファイル(上記を参照)の中にある<transactionManager>の type 属性に、" EXTERNAL"をセットしなければいけません。コントロールされた外部のトランザクションマネージャを使用する時、SQL Map トランザクションコントロールメソッドは、幾分、冗長です。なぜなら、トランザクションの開始、コミット、ロールバックは、外部のトランザクションマネージャで、コントロールされるためです。しかしながら、それらは依然として SqlMapClient の startTransaction()メソッドを使用して、トランザクションの境界を示して、パフォーマンスを（開始、コミット、ロールバックを自動トランザクションで行うのに対して）
向上させることができます。それらのメソッドを使い続けることで、一貫したプログラムのパラダイムを維持でき、さらにコネクションプールからコネクションに関するリクエストを減らします。それ以上の利点は、いくつかのケースにおいて、グローバルトランザクションがコミットされた時に対して、リソースがクローズされる（commitTransaction() または、endTransaction())順序を変更する必要があるかもしれないことです。異なるアプリケーションサーバとトランザクションマネージャは、（不幸なことに）異なるルールを持っています。それらの単純な検討の他は、グローバルトランザクションを使用するために、SQLMap を書く必要が、実際のところはありません。

管理されたグローバルトランザクション

SQL Map フレームワークは、グローバルトランザクションの管理をすることもできます。グローバルトランザクションの管理を有効にするために、SQL Map 設定ファイルの<transactionManager>の type 属性を“JTA”にしなければなりません。そして、UserTransaction インスタンスを見つけられるように“UserTransaction”プロパティに完全な JNDI 名をセットしなければなりません。<transactionManager>に関する設定詳細については上記で検討していますので見てください。

グローバルトランザクションを用いたプログラミングは、それほど難しくありません。しかしながら、いくつか検討すべきことがあります。下記は、例です。

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

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

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

例において、２つの異なるデータベースを使用する２つの SqlMapClient インスタンスがあるとします。トランザクションを使い始めた１つ目の SqlMapClient(orderSqlMap)は、グローバルトランザクションも開始します。これ以降、他の全ての活動は、同 SqlMapClient(orderSqlMap)が、commitTransaction()と endTransaction()を呼び出すまで、グルーバルトランザクションの部分とみなされます。呼び出された時に、グローバルトランザクションはコミットし、他の全ての作業を完了したとみなします。

Warning! シンプルに見えますが、グローバル（分散された）トランザクションを使いすぎないことは、パフォーマンスと密接な関係があるので、とても重要です。同様にアプリケーションサーバとデータベースドライバにも複雑な設定が必要となります。簡単に見えますが、いくつかの問題に遭遇するかもしれません。EJB は、より多くの工業サポートとサポートするツールを備えています。そして、分散トランザクションを必要とする作業のために、SessionEJB を使用する方が、賢明かもしれません。ibatis.apache.org で、見つけられる JPetStore サンプルは、SQLMap グローバルトランザクションの使用例です。

マルチスレッドプログラミング

iBATIS は、マルチスレッドプログラミングをサポートしています。しかし、知っておくべき検討事項があります。

はじめに、そして、一番重要な点は、トランザクションは、単一のスレッドに全て含められなければなりません。言い換えると、トランザクションは、スレッドの境界をまたがることができません。この理由のために、完全な作業単位全体にスレッドの開始を考えることは良い考えです。作業ごとのスレッドの関係性を保証することができないのであれば、トランザクションを開始し、実行するために、スレッドのプールを持つことは、一般的に良くない考えです。

もう１つの重要なことは、各スレッドに１つのアクティブなトランザクションだけ存在できることです。１つのスレッドで、１つ以上のトランザクションを実行するコードを書くことができます。しかし、トランザクションは一度に開始させるのではなく、同時にトランザクションを開始

できないため順番でなければなりません。これは、スレッドにおける複数のトランザクションの例です。

```
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();
}
```

重要なことは、トランザクションで一度にアクティブとなれるトランザクションが1つであるということです。もちろん、自動的なトランザクションの各ステートメントは、異なるトランザクションです。

iBATIS クラスローディング

(このセクションの情報は、iBATIS バージョン 2.2.0 以降で扱うことができます)

iBATIS は、*com.ibatis.common.resources.Resources クラスを、クラスをロードするために使用します。このクラスのクラスロードを検討するにあたり、最も重要なメソッドは、classForName(String)メソッドです。このメソッドは、iBATIS において全てのクラスローディングの源です。デフォルトでは、このメソッドは、下記の通り動作します。*

1. 現在のスレッドのコンテキストローダから、クラスをロードすることを試行します。
2. もし何らかのエラーが発生した時は、*Class.forName(String)でクラスのロードを試行します。*

このメソッドは、多くの環境で良く動作します。あなたの環境において、何らかの理由で、このメソッドが動作しないのであれば、Resources.setDefaultClassLoader(ClassLoader)static メソッドを呼び出して、全てのオペレーションに使用するためのクラスローダを指定できます。クラスローダを提供するのであれば、iBATIS は、指定されたクラスローダから全てのクラスをロードすることを試行します。(エラー時には、Class.forName(String)を呼び出します) もし、カスタムのクラスローダを提供するのであれば、iBATIS からのオペレーションを実行する前にクラスローダを指定しなければなりません。

バッチ

もし、クエリーではない(insert/update/delete)ステートメントを大量に実行したい時、ネットワークトラフィックを最小化し、JDBC のパフォーマンスを最適化(例.圧縮)するバッチとして実行したいでしょう。SQL Map API で、バッチをシンプルに使用できます。シンプルなメソッドで、バッチの境界を宣言できます。

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

上記の executeBatch()を呼ぶ、全てのバッチステートメントは、JDBC ドライバを通じて実行されます。executeBatch()の呼出しは任意です。なぜならば、コミット操作は、オープンしているバッチがあれば自動的に実行されるからです。そのため、影響をうける行数を知りたいときに、executeBatch()を呼び出します。または、行数を知ることをスキップして commitTransaction()だけを呼び出すことができます。

大量のオペレーションをバッチ処理する時に、バッチの途中でコミットをしたいことがあるかもしれません。例えば、1000 行も挿入するような場合、作成している大きなトランザクションから 100 行ごとにコミットしたいとします。そして、定期的にコミットを終了したいのであれば各コミットの後に、startBatch()を呼ぶことを知っておくことは重要です。なぜならば、バッチの最後にコミットが呼び出されるからです。下記は、例です。

```
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();
}
```

バッチに関する重要な点:

1. バッチは、常に明確なトランザクションの内側で使用すべきです。もし、明確なトランザクションを使用に失敗すれば、バッチを開始していないものとして iBATIS は、各ステートメントを実行するでしょう。
2. バッチの中でマップされたステートメントを実行したいことがあるかもしれません。もし、異なるマップされたステートメント（例 inserts、それから updates)を実行すれば、iBATIS は、最後に実行したステートメントの生成した SQL ベース上のサブバッチに分けます。例えば、下記のコードを考えてみてください。

```
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();
}
```

iBATISは、3つのサブバッチで、このバッチを実行します。1つ目は、最初の4つのinsertステートメント。もう1つは、次の2つのupdateステートメント。そして、最後は、3つのinsertステートメント。注意することは、最後の3つのinsertステートメントは、最初の4つのinsertステートメントと同じだとしても、iBATISは、異なるサブバッチのままで実行します。これは、updateステートメントが間にあるためです。

3. executeBatch()メソッドは、intを返します。バッチで更新されたレコードの総数です。もし、サブバッチがあればiBATISは、各サブバッチで更新された行数を総数に追加します。executeBatch()メソッドがレコードを更新しても0を返す場合においては、バッチの中で更新されたレコードの数を返すことをJDBCドライバが失敗することが正当であることに注意してください。Oracleドライバは、この振舞いをする好例です。
4. iBATISバージョン2.2.0以上であれば、バッチを実行するための異なるメソッド(*executeBatchDetailed*)を使用できます。このメソッドの機能は、executeBatchメソッド（明確なトランザクションが必要、サブバッチの使用など）と同じです。しかし、行数についてのより詳細な情報を返します。
executeBatchDetailedメソッドは、(サブバッチごとの)BatchResultオブジェクトのListを返します。各BatchResultオブジェクトは、サブバッチと関連するステートメントだけではなく、サブバッチが実行された時にJDBCドライバから返ってきたint[]に関連する情報も含みます。もし、 java.sql.BatchUpdateExceptionが発生した時には、このメソッドは例外が発生したステートメントだけではなく、いずれかの前に成功したサブバッチからのBatchResultオブジェクトのListに関する情報を含んでいるBatchExceptionをthrowします。

SqlMapClient API経由でのステートメントの実行

SqlMapClientは、全てのマップされたステートメントを実行するAPIを提供します。それらのメソッドは下記です。

```
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)
```

それぞれのケースにおいて、Mapped Statement はの名前は、1つ目のパラメータに渡します。この名前は、上記に記述したステートメントのエレメント(<insert>, <update>, <select>など)のname属性と連動しています。また、パラメータオブジェクトを渡すことは、常に任意です。パラメータが期待されていなければnullパラメータオブジェクト渡せます。それ以外は、パラメータオブジェクトが必須です。iBATIS 2.2.0現在、多くのメソッドもパラメータ期待されていない場合、パラメータオブジェクトなしのオーバーロードを持ちます。ほとんど似ていますが、残っている違いについてを下記のアウトラインに記します。

insert(), update(), delete():これらのメソッドは、更新ステートメント （ non-query とも呼ばれる） のために特に用意されているメソッドです。とは言うものの、下記のいずれかの query メソッドを使って更新ステートメントを実行することは不可能ではありません。
ただし、これは普通ではないセマンティックであり、明らかにドライバに依存します。
executeUpdate() の場合、更新ステートメントは単純に実行され、影響があった行の数が返却されます。

queryForObject(): executeQueryForObject()は２つのバージョンがあります。1つは、新しく割り当てられたオブジェクトを返します。もう1つは、パラメータに渡された(事前に割り当て済みの)オブジェクトを使用します。後者は、1つ以上のステートメントに代入されるオブジェクトのために便利です。

queryForList(): queryForList()の４つのバージョンがあります。１つ目は、クエリーを実行してクエリーの結果を全て返します。2つ目は、１つ目と似ています。しかし、パラメータオブジェクトを受け取りません。３つ目は、指定された結果数分スキップし、レコードの最大数分返します。レコード全体が大きすぎる時に、役に立ちます。４つ目は、３つ目に似ています。しかしパラメータオブジェクトを受け取りません。

queryWithRowHandler():このメソッドは、結果オブジェクトを通常通り行と列を使うのではなく、結果オブジェクトを使って、行ごとに結果セットを処理できるようにします。 メソッドに、標準名とパラメータオブジェクトが渡されます。しかし、それも RowHandler は、取得します。row handler は、 RowHandler インタフェースを実装したクラスです。RowHandler インタフェースは、下記のメソッドだけを持っています。

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

このメソッドは、データベースから行が返される度に RowHandler 上で呼び出されます。メソッドに渡される valueObject は、現在行のために解決済みの Java オブジェクトです。クエリーの結果を処理する方法するための、とてもきれいでシンプルかつ、拡張的な方法です。このメソッドによる iBATIS が全体をリストとして返すよりも、クエリーを固有のオブジェクトとして返すことができます。これは、とても大きな結果セットに対処する効果的な方法になるかもしれませんし、結果的にメモリを節約できます。

RowHandler の使用例は、下記の例となるセクションをみてください。

queryForPaginatedList():これは、前と後ろにナビゲートすることができるデータのサブセットを管理できるリストを返す便利なメソッドです。これは、クエリーから返された利用可能なレコード全てのサブセットを表示するだけのユーザインタフェースの実装に共通的に使用されます。サーチエンジンで 10,000 ヒットしたときに、一度に 100 個ずつ表示するような動作と似ています。PaginatedList インタフェースは、ページを通じたナビゲート(nextPage(), previousPage(), gotoPage())とページのステータスをチェックする(isFirstPage(), isMiddlePage(), isLastPage(), isNextPageAvailable(), isPreviousPageAvailable(), getPageIndex(), getPageSize())ためのメソッドを含んでいます。利用可能なレコードの総数に PaginatedList インタフェースからアクセスできないにも関わらず、期待する結果をカウントする２つ目のステートメントを単に実行することで容易に達成できるべきです。さもなければ、多すぎるオーバヘッドは、 PaginatedList と関連します。

queryForMap():このメソッドは、結果の集合をリストに入れ、ロードするための代替手段を提供するメソッドです。ロードする代わりに keyProperty に渡されたパラメータによってマップキーに結果を入れます。例えば、Employee オブジェクトの集合をロードするのであれば employeeNumber によって番号付けされたマップに、それらをロードします。
マップの値は、employee オブジェクト全体か、任意の valueProperty と呼ばれる２つ目のパラメータで指定された employee オブジェクトのプロパティとできます。例えば、単に社員番号のキーと社員名をマップしたいかもしれません。
このメソッドと結果オブジェクトとして Map 型を使う概念を混同しないでください。このメソッドは、結果オブジェクトが JavaBean か Map(かプリミティブラッパー、しかし役に立たない可能性があります)かどうか関係なく使用できます。

flushDataCache():これらのメソッドは、 データキャッシュをフラッシュするプログラム的な方法を提供します。引数がないメソッドは全てのデータキャッシュをクリアします。引数にキャッシュ ID を受け取るメソッドは、名前つきのデータキャッシュのみをフラッシュします。後者の使いかたには、注意点があります。ネームスペースを使用してキャッシュ ID を指定する必要があります。(**useStatementNamespaces を false にしていたとしてもです。**)

例 1 : 更新の実行(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();
```

例 2 : オブジェクトのためのクエリーの実行 (select)

```
sqlMap.startTransaction();

Integer key = new Integer (1);

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

sqlMap.commitTransaction();
```

例 3 : 事前に割り当て済みの結果オブジェクトを用いたオブジェクトのためのクエリーの実行 (select)

```
sqlMap.startTransaction();

Customer customer = new Customer();

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

sqlMap.commitTransaction();
```

例 4 : List のためのクエリーの実行 (select)

```
sqlMap.startTransaction();

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

sqlMap.commitTransaction();
```

例５：自動コミット

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

例６：結果件数指定を用いたList のためのクエリーの実行（select）

```
sqlMap.startTransaction();

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

sqlMap.commitTransaction();
```

例７：RowHandler を用いたクエリーの実行（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();
```

例８：ページ化したリストのためのクエリーの実行（select）

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

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

例９：Map のためのクエリーの実行

```
sqlMap.startTransaction();
Map map = sqlMap.queryForMap ( “getProductList”, null, “productCode” );
sqlMap.commitTransaction();
Product p = (Product) map.get( “EST-93” );
```

SqlMap アクティビティのロギング

SqlMap フレームワークは、内部のログファクトリを使用してログ情報を提供します。内部のログファクトリは、下記のログ実行の１つにログ情報を委譲します。:

1. Jakarta Commons Logging（JCL - *ジョブ操作言語ではありません！*）
2. Log4J
3. JDK logging（JRE 1.4 以上が必須）

ロギングソリューションは、内部の iBATIS ログファクトリによって実行時のイントロスペクションをベースに選択されます。iBATIS ログファクトリは、始めに見つけたログ実装を使用します(実装された検出順は、上記の記述順です)。もし、iBATIS が上記の実装を見つけられないのであればロギングは無効となります。

多くの環境では、JCL は、アプリケーションサーバの classpath に含まれています(良い例は、Tomcat, WebSphere)。環境を知ることは、重要です。iBATIS は、ログ実装として JCL を使用するでしょう。 WebSphere の環境では、Log4J 設定は無視されることを意味します。なぜならば WebSphere は、独自の JCL の実装を提供しているためです。これは、とても厄介です。なぜならば、あなたの Log4J の設定を無視してしまうからです（実際に iBATIS は、Log4J 設定を無視します。iBATIS は、そのような環境で JCL を使用するためです）。

もし、アプリケーションサーバの classpath に JCL が含まれている環境で、他のロギングの実装を使いたいのであれば、下記のメソッドの１つを呼ぶことで異なるログの実装を選択できます(iBATIS 2.2.0 以降で利用可能です)。):

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

もし、それらのメソッドの１つを呼び出すのであれば、他の iBATIS メソッドよりも前に呼び出さなくてはいけません。また、それらのメソッドは、もし実行時の classpath で利用可能なログ実装にのみ切り替えることができます。例えば Log4J が利用可能でない時に、Log4J ロギングを選択しても、iBATIS は、Log4J を使用する要求を無視して、通常のアルゴリズムでログの実装を見つけるでしょう。

Jakarta Commons Logging, Log4J と JDK 1.4 Logging API の仕様は、このドキュメントの範囲外です。しかしながら、設定例は下記で取得できます。それらのフレームワークについて、もっと知りたいのであれば下記の場所より情報を得ることができます。

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

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

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

ログ設定

iBATIS は、iBATIS の中にないログクラスを使用しているほとんどの活動を記録します。iBATIS のロギングステートメントを見るために、java.sql パッケージの下記で指定しているクラスのロギングを有効にするべきです。

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

また、この設定を実行する方法はロギングの実装に依存しています。ここでは、Log4Jにおける設定例を示します。

ロギングサービスを設定することは、単に１つかそれ以上の特別な設定ファイル(例 log4j.properties)と、たまに新しいJarファイル(例 log4j.jar)を含めることです。下記の設定例は、プロパティとしてLog4Jを使用して完全なログサービスを設定しています。２つのステップがあります。

ステップ１：Log4J JARファイルの追加

Log4Jを使用するので、アプリケーションで利用可能なようにJARファイルを確保する必要があります。Log4Jを使用するために、アプリケーションのclasspathにJARファイルを追加する必要があります。上記のURLよりLog4Jをダウンロードできます。Webもしくは、エンタープライズアプリケーションでは、WEB-INF/libディレクトリにlog4j.jarを追加できます。スタンダードアプリケーションであれば、単に起動パラメータの-classpathにlog4j.jarファイルを追加してください。.

ステップ２：Log4Jの設定

Log4Jの設定は、シンプルです。下記のようなlog4j.propertiesと呼ばれるファイルを作成してください:

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
```

上記のファイルは、発生したロギングをレポートするだけの最小の設定です。ファイルの２行目は、stdout appenderにエラーをレポートするだけのLog4Jの設定を示しています。appenderとは、出力(例.コンソール、ファイル、データベースなど)を集めたコンポーネントのことです。レポートのレベルを最大にするために２行目を下記に変更します。

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

２行目を上記に変更することで、Log4Jは、'stdout' appender(コンソール)に全てのイベントをレポートするでしょう。もし、ロギングのレベルを調整したいのであれば、(５行目から14行目までコメントアウトされている)上記ファイルの'SqlMap logging configuration' を使用してシステムに記録する各クラスを設定できます。そのため、もし PreparedStatementの活動(SQLステートメント)をDEBUGレベルで、コンソールにログを出力したいのであれば13行目を単に以下のように変更します（コメントアウトされていないことに注意してください）。

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

log4j.properties ファイルに残っている設定は、appender の設定です。設定については、このドキュメントの範囲外です。しかしながら、Log4J のウェブサイト(URL は、前述)で、より多くの情報を見つけることができます。もしくは、他の設定の効果を単に見るために設定を変えてみることもできます。

1ページの JavaBeans コース

Data Mapper フレームワークは、JavaBeans のしっかりとした理解を必要とします。幸運なことに Data Mapper に関する限り JavaBeans API はそれほど多くありません。これは、JavaBeans API に接したことがない方のための JavaBeans のクイックイントロダクションです。

JavaBean は何でしょうか？JavaBean は、アクセスまたはクラスの状態を変化させるメソッドの命名規則に則ったクラスです。言い方を変えると、getter/setter の規約に則っているクラスです。JavaBean のプロパティは、フィールドではなくメソッドにより定義されます。set で始まるメソッドは、プロパティを変更可能とするメソッドです（例.setEngine）。get で始まるメソッドは、プロパティを読み込めるメソッドです(例.getEngine)。boolean プロパティを読み込めるメソッドは、is で始めることもできます（例.isEngine)。Set メソッドは、返り値を定義すべきではありません(void にすべき)そして、プロパティのためにふさわしい単一のパラメータのみ受け入れるべきです。Get メソッドは、ふさわしい値（例.String）を返して、パラメータを受け入れないべきです。通常、強制ではないが、Set メソッドのパラメータの型と Get メソッドのパラメータの型は同一にすべきです。JavaBeans は、Selializable インタフェースも実装すべきです。JavaBeans は、（イベントなど）他の機能もサポートします。しかし、それらは Data Mapper の範疇においては重要ではありません。そして、通常は、web アプリケーションの範疇においても重要ではありません。

これは、JavaBean のサンプルです。

```
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;
  }
}
```

Note! 与えられたプロパティのために get と set プロパティの型を混ぜないでください。例えば数値の”account”プロパティの getter と setter で同じ数値型が使用されていることを確認してください。

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

同じ int 型を使用していることに注意してください。例えば get メソッドだけ long 型を返すことは問題を発生させるでしょう。

Note! 同様に getXxxx()と setXxxx()と名前をつけたメソッドが１つだけであることを確認してください。多様性に思慮深くなってください。同じ名前のメソッドに特別な名前をつけることでより良い状態になります。

Note! boolean型プロパティには、getterシンタックスの代用があります。そのgetメソッドは、多分isXxxx()の書式で命名されています。２つではなく、isメソッドかgetメソッドのどちらかだけ持っていることを確認してください。両方ではありません！

素晴らしい！コースをパスしました！

Okay、２ページ目

Side Bar: Object Graph Navigation (JavaBeans Properties, Maps, Lists)

このドキュメントを通さなくてもStrutsとか何か他のJavaBeans互換性があるフレームワークを通じて特別なシンタックスでオブジェクトにアクセスできるのをみているかもしれません。Data Mapperフレームワークは、JavaBeansプロパティ、Map(キー値)、そしてListをナビゲートするオブジェクトグラフを使用することが可能です。下記のナビゲーションを見てみましょう。(ListとMapとJavaBeansが含まれています)

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

employeeオブジェクトのプロパティは、SqlMapClientにおいて（上記のemployeeオブジェクトに対して）は下記のようにナビゲートできます：

```
“departmentList[3].address.city”
```

Important: このシンタックスは、動的SQLエレメントのためのiBATISサポートを使用してプロパティをを使用するときだけに適用できます。<result>もしくは、<parameter>マッピングにおいては動作しないでしょう。

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

Resourcesクラスは、classpathからリソースをロードするためのメソッドを提供します。ClassLoadersとやりとりすることは、特にアプリケーションサーバ/コンテナにおいて、退屈な作業を簡単にやりとりできるようにします。

リソースファイルの共通の使用方法：

- claspathからSQL Map設定ファイル（例. sqlMap-config.xml)の読み込み
- classpathからさまざまな*.propertiesファイルの読み込み
- Etc.

それらは、リソースファイルをロードする異なる方法を含んでいます。

- Reader:シンプルな読み込み専用テキストデータ向け
- InputStream:シンプルな読み込み専用バイナリデータ向け
- File:読み書きできるバイナリもしくはテキストファイル向け
- Propertiesファイル：読み込み専用設定propertiesファイル向け

上記のスキーマのいずれか1つを使ってリソースを読み込むResoucesクラスのメソッドは下記です。（上記の順番と同じ順番で記載しています。）

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

それらのケースにおいて、リソースをロードするために使用するClassLoaderは、Resoucesクラスをロードしたものと同じです。ロードが失敗したときは、システムクラスローダーが使用されます。（例えばアプリケーションサーバの中など）ClassLoaderがやっかいな環境においては、使用するClassLoaderを指定できます(例.独自アプリケーションクラスからClassLoaderを使用する)。上記のメソッドでは、それぞれ一つめの引数にClassLoaderを指定できるシグネチャのメソッドがあります。それらは：

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

resoucerパラメータによって、命名されたリソースはパッケージと全てのファイル/リソース名となります。例えば、classpathに‘com.domain.mypackage.MyPropertiesFile.properties’というリソースを持っているとすれば、Resoucesクラスを使ってプロパティファイルをロードするコードは下記となります。(リソースが”/”で始まらないことに注意してください)

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

同様にReaderでclasspathからSqlMap設定ファイルをロードできます。classpathに、propertiesパッケージ(properties.sqlMap-config.xml)がある場合は、こうなります。

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

リソースの国際化

Note: このセクションの情報は、現在 iBATIS バージョン 2.3 以降となります。

iBATIS において、国際化に対する主要に関心をもつメインエリアは、XML 設定ファイルです。ファイルがエンコーディングを持たないか XML ファイルのコーディングとシステムデフォルトエンコーディングがマッチしないのであれば、ときどきエラーが発生します。iBATIS は、この問題に対する２つの異なる解決策を提供します。

キャラクタリーダによる国際化

Reader を使っているとき、iBATIS は、ファイルのエンコードをするために Java クラス InputStream を使います。デフォルトでは、このクラスはシステムのデフォルトエンコーディングを使用します。いくつかの環境においては、システムのデフォルトエンコーディングは、XML によって支持されている unicode エンコードとうまくいきません。もし、入力として Reader により iBATIS XML ファイルを解析しているときにエンコーディング問題に遭遇したら、デフォルトエンコーディングを XML ファイルのエンコーディングに変更できます。例えば：

```
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”メソッドは、将来の全て“getResourceAsReader”呼び出しに対するエンコーディングを変更します。もし、システムのデフォルトに戻したいのであれば、単に“setCharset(null)”を呼び出します。

Byte Input Stream による国際化

もし、XML 設定ファイルを読み出すために byte InputStream を使用するのであれば、多くの場合パーサーは、ファイルエンコーディングを自動的に決定することができる。多くの場合二つのメソッドの使用に対してベストチョイスです。このメソッドの使用例は下記です。

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

このメソッドは、キャラクタエンコーディングのパーサーのネイティブサポートに頼ります。もし、このメソッドでエラーが発生すのであれば、どのようにデフォルトスキーマを上書きするのかパーサーのドキュメントをみてください。

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

SimpleDataSourceクラスは、JDBC 2.0 に対応したDataSourceのシンプルな実装です。それは、コネクションプールの機能と完全な同期のセットをサポートします。SimpleDataSourceは、とても軽量で手軽なコネクションプーリングのソリューションです。SimpleDataSourceは、他のJDBC標準拡張APIの一部として文書化されているJDBC DataSourceの実装と同様に使用されます。文書はここで見つけることができます：

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

Note!: JDBC 2.0 APIは、現在J2SE 1.4.xの一部となっています。

Note!: SimpleDataSourceは、とても便利で効率的で有能です。しかし、規模が大きいエンタープライズもしくは、ミッションクリティカルなアプリケーションのためには、エンタープライズレベルのデータソースの実装を使用することを推奨します。(アプリケーションサーバ付属、もしくは、広く利用されているO/Rマッピングツール)

SimpleDataSourceのコンストラクタは、設定プロパティの値が格納されているパラメータを必要利益。下記のテーブルに名前とプロパティについて記述しています。”JDBC.”プロパティだけは必須です。

プロパティ名	必須	デフォルト	説明
JDBC.Driver	Yes	n/a	JDBCドライバクラス名
JDBC.ConnectionURL	Yes	n/a	JDBC接続URL
JDBC.Username	Yes	n/a	データベースにログインするためのユーザ名
JDBC.Password	Yes	n/a	データベースにログインするためのパスワード
JDBC.DefaultAutoCommit	No	ドライバ依存	プールで作成された全てのコネクションに対するautocommitのデフォルト設定
Pool.MaximumActiveConnections	No	10	一時に、オープンすることができるコネクションの最大数
Pool.MaximumIdleConnections	No	5	プールに格納されたアイドル接続の数
Pool.MaximumCheckoutTime	No	20000	接続が強制されたコネクション候補になる前に、”checkout”となることができる時間（ミリ秒）の長さ
Pool.TimeToWait	No	20000	(それらが全て使用されているので)クライアントがスレッドがコネクションを取得しようとする前に強制的に待つ時間（ミリ秒）の長さ。この時間内にコネクションがプールに返されてスレッドに通知される十分な可能性があります。それゆえ、スレッドは、指定した時間まで待つ必要がないかもしれません（単に最大待ち時間です）
Pool.PingQuery	No	n/a	pingクエリーは、データベースに対してテスト接続を行います。コネクションが信頼できない環境においては、ping queryを使用してプールから常に良いコネクションを返すことを保証するために役に立ちます。しかしながら、これはパフォーマンスに重大な影響をもらたします。pingクエリーを設定するときは、気を付けてください。そして多くのテストで確認してください。

SimpleDataSource (continued…)

Pool.PingEnabled	No	false	pingクエリーの有効か無効を設定します。大抵のアプリケーションには設定する必要はありません。
Pool.PingConnectionsOlderThan	No	0	この値（ミリ秒）より古い接続は、pingクエリーによってテストされます。もし、あなたのデータベースの環境で決まった時間（例.12時間）でコネクションを落としてしまうのであれば役に立ちます。
Pool.PingConnectionsNotUsedFor	No	0	コネクションが、pingクエリーを使用してテストされた値よりも長くインアクティブになっていたらpingクエリーによってテストされます。もし、あなたのデータベースの環境で決まった時間（例.12時間）でコネクションを落としてしまうのであれば役に立ちます。
*Driver.**	No	n/a	多くのJDBCドライバは、拡張プロパティで追加機能を設定することをサポートしています。ドライバにプロパティを送るために、”Driver.”を先頭に付けることにより送ることができます。例えば、“compressionEnabled”プロパティを持っているのであれば“Driver.compressionEnabled=true”とセットできます。 Note: それらのプロパティは、sqlMap-config.xmlの中でも動作します。

SimpleDataSource使用例：

```
// 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();
```