MyBatis 3

Manual de Usuario

Cuidado al copiar código de este manual

No, esta no es una advertencia legal. Es por tu salud mental. Los procesadores de texto modernos hacen un gran trabajo al formatear texto de forma que este sea legible y estético. Sin embargo, suelen estropear los ejemplos de código porque insertan caracteres especiales que en ocasiones aparentan ser iguales que los que debería haber. Las "Comillas" y los guiones son un ejemplo perfecto –las comillas y el guión que ves a la izquierda no servirían como comillas en un IDE o un editor de texto, al menos no deberían hacerlo.

Por lo tanto, lee este documento, disfrútalo y esperamos que te sea de utilidad. En cuanto a los ejemplos de código, usa los de las descargas (incluidas la pruebas unitarias…), o ejemplos de la página web o de la lista de distribución.

Colabora en mejorar esta documentación…

Si ves que hay alguna carencia en esta documentación, o que falta alguna característica por documentar, te animamos a que lo investigues y la documentes tu mismo!

Puedes enviar tus contribuciones a la documentación a nuestra Wiki::

http://opensource.atlassian.com/confluence/oss/display/IBATIS/Contribute+Documentation

Eres el mejor candidato para documentar porque los lectores de esta documentación son gente como tú!

Contents

Qué es MyBatis? 5

Primeros Pasos 5

Cómo crear un SqlSessionFactory a partir de XML 5

Cómo crear un SqlSessionFactory sin XML 6

Cómo obtener un SqlSession a partir del SqlSessionFactory 6

Cómo funcionan los *Mapped SQL Statements* 7

Nota sobre *namespaces* 8

Ámbito y ciclo de vida 9

Configuración XML 10

properties 11

settings 12

typeAliases 13

typeHandlers 14

objectFactory 16

plugins 16

environments 18

transactionManager 19

dataSource 20

mappers 22

SQL Map XML Files 22

select 23

insert, update, delete 25

sql 27

Parameters 27

resultMap 29
Mapeo de resultados avanzado 32
id, result 34
Tipo JDBC soportados 34
constructor 34
association 36
collection 39
discriminator 41
cache 43
Usando una caché personalizada 44
cache-ref 45
SQL dinámico 45
if 46
choose, when, otherwise 47
trim, where, set 47
foreach 49
Java API 50
Estructura de directorios 50
SqlSessions 51
SqlSessionFactoryBuilder 51
SqlSessionFactory 53
SqlSession 55
SelectBuilder 63
SqlBuilder 67
Logging 68

Qué es MyBatis?

MyBatis es un *framework* de persistencia que soporta SQL, procedimientos almacenados y mapeos avanzados. MyBatis elimina casi todo el código JDBC, el establecimiento manual de los parámetros y la obtención de resultados. MyBatis puede configurarse con XML o anotaciones y permite mapear mapas y *POJOs (Plain Old Java Objects)* con registros de base de datos.

Primeros Pasos

Una aplicación que usa MyBatis deberá utilizar una instancia de SqlSessionFactory. La instancia de SqlSessionFactory se puede obtener mediante el SqlSessionFactoryBuilder. Un SqlSessionFactoryBuilder puede construir una instancia de SqlSessionFactory a partir de un fichero de configuración XML o de una instancia personalizada de la clase Configuration.

Cómo crear un SqlSessionFactory a partir de XML

Crear una instancia SqlSessionFactory desde un fichero xml es muy sencillo. Se recomienda usar un *classpath resource*, pero es posible usar cualquier *Reader*, incluso creado con un *path* de fichero o una URL de tipo file://. MyBatis proporciona una clase de utilidad, llamada Resources, que contiene métodos que simplifican la carga de recursos desde el *classpath* u otras ubicaciones.

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

El fichero de configuración XML contiene la configuración del *core* de MyBatis, incluyendo el DataSource para obtener instancias de conexión a la base de datos y también un TransactionManager para determinar cómo deben controlarse las transacciones. Los detalles completos de la configuración XML se describen más adelante en este documento, a continuación se muestra un ejemplo:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
    PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-config.dtd">
<configuration>
  <environments default="development">
      <environment id="development">
      <transactionManager type="JDBC"/>
      <dataSource type="POOLED">
        <property name="driver" value="${driver}"/>
        <property name="url" value="${url}"/>
        <property name="username" value="${username}"/>
        <property name="password" value="${password}"/>
      </dataSource>
    </environment>
  </environments>
  <mappers>
    <mapper resource="org/mybatis/example/BlogMapper.xml"/>
```

```
  </mappers>
</configuration>
```

Aunque hay mucha más información sobre el fichero de configuración XML, el ejemplo anterior contiene las partes más importantes. Observa que hay una cabecera XML, requerida para validar el fichero XML. El cuerpo del elemento *environment* contiene la configuración de la gestión de transacciones correspondiente al entorno. El elemento *mappers* contiene la lista de *mappers* – Los ficheros XML que contienen las sentencias SQL y las definiciones de mapeo.

Cómo crear un SqlSessionFactory sin XML

Si lo prefieres puedes crear la configuración directamente desde Java, en lugar de desde XML, o crear tu propio *builder*. MyBatis dispone una clase *Configuration* que proporciona las mismas opciones de configuración que el fichero XML.

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

Puedes observar que en este caso la configuración está añadiendo una clase mapper. Las clases mapper son clases Java que contienen anotaciones de mapeo SQL que permiten evitar el uso de XML. Sin embargo el XML sigue siendo necesario en ocasiones, debido a ciertas limitaciones le las anotaciones Java y la complejidad que pueden alcanzar los mapeos (ej. mapeos anidados de Joins). Por esto, MyBatis siempre busca si existe un fichero XML asociado a la clase mapper (en este caso, se buscará un fichero con nombre BlogMapper.xml cuyo nombre deriva del classpath y nombre de BlogMapper.class). Hablaremos más sobre esto más adelante.

Cómo obtener un SqlSession a partir del SqlSessionFactory

Ahora que dispones de un SqlSessionFactory, tal y como su nombre indica, puedes adquirir una instancia de SqlSession. SqlSession contiene todos los métodos necesarios para ejecutar sentencias SQL contra la base de datos. Puedes ejecutar mapped statements con la instancia de SqlSession de la siguiente forma:

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

Aunque esta forma de trabajar con la SqlSession funciona correctamente y les será familiar a aquellos que han usado las versiones anteriores de MyBatis, actualmente existe una opción más recomendada. Usar un interface (ej. BlogMapper.class) que describe tanto el parámetro de entrada como el de retorno

para una sentencia. De esta forma tendrás un código más sencillo y *type safe*, sin castings ni literales de tipo String que son fuente frecuente de errores.

Por ejemplo:

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

Vemos con detalle cómo funciona esto.

Cómo funcionan los *Mapped SQL Statements*

Te estarás preguntando qué se está ejecutando en SqlSession o en la clase Mapper. Los *Mapped Sql Statements* son una materia muy densa, y será el tema que domina la mayor parte de esta documentación. Pero, para que te hagas una idea de qué se está ejecutando realmente proporcionaremos un par de ejemplos.

En cualquiera de los ejemplos a continuación podrían haberse usado indistintamente XML o anotaciones. Veamos primero el XML. Todas las opciones de configuración de MyBatis pueden obtenerse mediante el lenguaje de mapeo XML que ha popularizado a MyBatis durante años. Si ya has usado MyBatis antes el concepto te será familiar, pero verás que hay numerosas mejoras en los ficheros de mapeo XML que iremos explicando más adelante. Por ejemplo este mapped statement en XML haría funcionar correctamente la llamada al SqlSession que hemos visto previamente.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
    PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="org.mybatis.example.BlogMapper">
  <select id="selectBlog" parameterType="int" resultType="Blog">
    select * from Blog where id = #{id}
  </select>
</mapper>
```

Aunque pudiera parecer que hay excesivo XML para un ejemplo tan simple, en realidad no hay tanto. Puedes definir tantos mapped statements en un solo fichero XML como quieras así que rentabilizarás las líneas XML extra que corresponden a la cabecera XML y a la declaración de doctype. El resto del fichero se explica por sí mismo. Define un nombre para el mapped statement “selectBlog”, en un namespace (espacio de nombres) “org.mybatis.example.BlogMapper”, que permite realizar una llamada especificando el nombre completo (*fully qualified*) “org.mybatis.example.BlogMapper.selectBlog” tal y como muestra el código a continuación:

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

Observa el gran parecido que hay entre esta llamada y la llamada a una clase java y hay una razón para eso. Este literal puede mapearse con una clase que tenga el mismo nombre que el namespace, con un método que coincida con el nombre del statement, y con parámetros de entrada y retorno iguales que los del statement. Esto permite que puedas hacer la misma llamada contra una interfaz Mapper tal y como se muestra a continuación:

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

Este segunda forma de llamada tiene muchas ventajas. Primeramente, no se usan literales de tipo String lo cual es mucho más seguro dado que los errores se detectan en tiempo de compilación. Segundo, si tu IDE dispone de autocompletado de código podrás aprovecharlo. Y tercero, no es necesario el casting del tipo de salida dado que el interfaz tiene definidos los tipos de salida (y los parámetros de entrada).

Nota sobre *namespaces*

➔ **Los Namespaces** eran opcionales en versiones anteriores de MyBatis, lo cual creaba confusión y era de poca ayuda. Los *namespaces* son ahora obligatorios y tienen un propósito más allá de la clasificación de *statements*.

Los *namespaces* permiten realizar el enlace con los interfaces como se ha visto anteriormente, e incluso si crees que no los vas a usar a corto plazo, es recomendable que sigas estas prácticas de organización de código por si en un futuro decides hacer lo contrario. Usar un *namespace* y colocarlo en el paquete java que corresponde con el namespace hará tu código más legible y mejorará la usabilidad de MyBatis a largo plazo.

➔ **Resolución de nombres**: Para reducir la cantidad de texto a escribir MyBatis usa las siguientes normas de resolución de nombres para todos los elementos de configuración, incluidos statements, result maps, cachés, etc.

- Primeramente se buscan directamente los nombres completamente cualificados *(fully qualified names)* (ej. "com.mypackage.MyMapper.selectAllThings").
- Pueden usarse los nombres cortos (ej. "selectAllThings") siempre que no haya ambigüedad. Sin embargo, si hubieran dos o más elementos (ej. "com.foo.selectAllThings and com.bar.selectAllThings"), entonces obtendrás un error indicando que el nombre es ambiguo y que debe ser *"fully qualified"*.

Hay otro aspecto importante sobre las clases Mapper como BlogMapper. Sus *mapped statements* pueden no estar en ningún fichero XML. En su lugar, pueden usarse anotaciones. Por ejemplo, el XML puede ser eliminado y reemplazarse por:

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

Las anotaciones son mucho más claras para sentencias sencillas, sin embargo, las anotaciones java son limitadas y más complicadas de usar para sentencias complejas. Por lo tanto, si tienes que hacer algo complejo, es mejor que uses los ficheros XML.

Es decisión tuya y de tu proyecto cuál de los dos métodos usar y cómo de importante es que los *mapped statements* estén definidos de forma consistente. Dicho esto, no estás limitado a usar un solo método, puedes migrar fácilmente de los mapped statements basados en anotaciones a XML y viceversa.

Ámbito y ciclo de vida

Es muy importante entender los distintos ámbitos y ciclos de vida de las clases de las que hemos hablado hasta ahora. Usarlas de forma incorrecta puede traer serias complicaciones.

SqlSessionFactoryBuilder

Esta clase puede instanciarse, usarse y desecharse. No es necesario mantenerla una vez que ya has creado la SqlSessionFactory. Por lo tanto el mejor ámbito para el SqlSessionFactoryBuilder es el método (ej. una variable local de método). Puedes reusar el SqlSessionFactoryBuilder para construir más de una instancia de SqlSessionFactory, pero aun así es recomendable no conservar el objeto para asegurarse de que todos los recursos utilizados para el parseo de XML se han liberado correctamente y están disponibles para temas más importantes.

SqlSessionFactory

Una vez creado, el SqlSessionFactory debería existir durante toda la ejecución de tu aplicación. No debería haber ningún o casi ningún motivo para eliminarlo o recrearlo. Es una buena práctica el no recrear el SqlSessionFactory en tu aplicación más de una vez. Y lo contrario debería considerarse código sospechoso. Por tanto el mejor ámbito para el SqlSessionFactory es el ámbito de aplicación. Esto puede conseguirse de muchas formas. Lo más sencillo es usar el patrón *Singleton* o el *Static Singleton.* Sin embargo ninguno de estos patrones está considerado una buena práctica. Es más recomendable que utilices un contendor de IoC como Google Guice o Spring. Estos *frameworks* se encargarán de gestionar el ciclo de vida del SqlSessionFactory por ti.

SqlSession

Cada *thread* (hilo de ejecución) debería tener su propia instancia de SqlSession. Las instancias de SqlSession no son *thread safe* y no deben ser compartidas. Por tanto el ámbito adecuado es el de petición (*request)* o bien el método. No guardes nunca instancias de SqlSession en un campo estático o incluso en una propiedad de instancia de una clase. Nunca guardes referencias a una SqlSession en ningún tipo de ámbito gestionado como la HttpSession. Si estás usando un framework web considera que el SqlSession debería tener un ámbito similar al HttpRequest. Es decir, cuando recibas una petición http puedes abrir una SqlSession y cerrarla cuando devuelvas la respuesta. Cerrar la SqlSession es muy importante. Deberías asegurarte de que se cierra con un bloque finally. A continuación se muestra el patrón estándar para asegurarse de que las sesiones se cierran correctamente.

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

Usando este patrón en todo el código se asegura que los recursos de base de datos se liberarán correctamente (asumiendo que no has pasado tu propia conexión a base de datos, con lo cual MyBatis entenderá que serás tú quien gestione dicha conexión)

Instancias de Mapper

Los mappers son interfaces que creas como enlace con los mapped statements. Las instancias de mappers se obtienen de una SqlSession. Y por tanto, técnicamente el mayor ámbito de una instancia de Mapper es el mismo que el de la SqlSession de la que fueron creados. Sin embargo el ámbito más recomendable para una instancia de mapper es el ámbito de método. Es decir, deberían ser obtenidos en el método que vaya a usarlos y posteriormente descartarlos. No es necesario que sean explícitamente cerrados. Aunque no es un problema propagar estos objetos por varias clases dentro de una misma llamada, debes tener cuidado porque puede que la situación se te vaya de las manos. Hazlo fácil (*keep it simple*) y mantén los mappers en el ámbito de método. Este ejemplo muestra esta práctica:

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

Configuración XML

El fichero de configuración XML contiene parámetros y configuraciones que tienen un efecto crucial en cómo se comporta MyBatis. A alto nivel contiene:

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

properties

Contiene propiedades externalizables y sustituibles que se pueden configurar en un típico *properties* de Java o bien puede definirse su contenido directamente mediante subelementos *property*. Por ejemplo:

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

Las propiedades pueden usarse a lo largo del fichero de configuración para sustituir valores que deben configurarse dinámicamente. Por ejemplo:

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

El usuario y password de este ejemplo se reemplazarán por los valores de los elementos de tipo property. El driver y la url se reemplazarán por los valores contenidos en el fichero config.properties. Esto aumenta mucho las posibilidades de configuración.

Las propiedades también pueden pasarse como parámetro al método SqlSessionBuilder.build(). Por ejemplo:

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

// ... or ...

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

Si una propiedad existe en más de un lugar, MyBatis la carga en este orden:

- Primero se leen las propiedades especificadas en el elemento XML properties,
- Posteriormente se cargan las propiedades de recursos de tipo classpath o url del elementos properties, si hubiera una propiedad repetida sería sobrescrita,
- Y finalmente se leen las propiedades pasadas como parámetro, que en caso de duplicidad sobrescriben las propiedades que se hayan cargado del elemento properties o de recursos/url.

Por tanto las properties más prioritarias son las pasadas como parámetro, seguidas de los atributos tipo classpath/url y finalmente las propiedades especificadas en el elemento properties..

settings

Son muy importantes para definir cómo se comporta MyBatis en runtime. La siguiente tabla describe las configuraciones (*settings*), sus significados y sus valores por defecto.

Parámetro	Descripción	Valores	Defecto
cacheEnabled	Habilita o inhabilita globalmente todas las cachés definidas en el mapper	true | false	true
lazyLoadingEnabled	Habilita o inhabilita globalmente la carga diferida (*lazy loading*). Cuando está inhabilitada todas las asociaciones se cargan de forma inmediata (*eagerly*)	true | false	true
aggressiveLazyLoading	Cuando está habilitada, todos los atributos de un objeto con propiedades *lazy loaded* se cargarán cuando se solicite cualquiera de ellos. En caso contrario, cada propiedad es cargada cuando es solicitada.	true | false	true
multipleResultSetsEnabled	Habilita o inhabilita la obtención de múltiples ResultSets con una sola sentencia (se requiere un driver compatible)	true | false	true
useColumnLabel	Utiliza la etiqueta de columna (*label*) en lugar del nombre de conlumna. Algunos drivers se comportan distinto en lo que a esto respecta. Consulta la documentación del driver o prueba ambos modos para descubrir cómo funciona tu driver	true | false	true
useGeneratedKeys	Habilita el uso del soporte JDBC para claves autogeneradas. Se requiere un driver compatible. Este parámetro fuerza el uso de las claves autogeneradas si está habilitado. Algunos drivers indican que no son compatibles aunque funcionan correctamente (ej. Derby)	true | false	False
autoMappingBehavior	Especifica cómo deben mapearse las columnas a los campos/propiedades. PARTIAL solo mapea automáticamente los resultados simples, no anidados. FULL	NONE, PARTIAL, FULL	PARTIAL

	mapea los mapeos de cualquier complejidad (anidados o no).		
defaultExecutorType	Configura el ejecutor (*executor*) por defecto. SIMPLE no hace nada especial. REUSE reúsa prepared statements. BATCH reúsa *statements* y ejecuta actualizaciones en batch.	SIMPLE REUSE BATCH	SIMPLE
defaultStatementTimeout	Establece el *timeout* que determina cuando tiempo debe esperar el driver la respuesta de la base de datos.	Cualquier entero positivo	Sin valor (null)
safeRowBoundsEnabled	Habilita el uso de RowBounds en statements anidados.	true | false	true
mapUnderscoreToCamelCase	Mapea automáticamente los nombres clásicos de columnas de base de datos A_COLUMN a nombres clásicos de propiedades Java aColumn.	true | false	false

A continuación se muestra un ejemplo del elemento settings al completo:

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

typeAliases

Un *type alias* es simplemente un alias (un nombre más corto) para un tipo Java. Solo es importante para la configuración XML y existe para reducir la cantidad de texto al teclear nombres de clase cualificados (*fully qualified*). Por ejemplo:

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

Con esta configuración, puede usarse “Blog” en lugar de “domain.blog.Blog”.

Hay muchos type aliases pre construidos. No son sensibles a mayúsculas/minúsculas. Observa los nombres especiales de los tipos primitivos dadas las colisiones de nombres.

Alias	Mapped Type
_byte	byte
_long	long
_short	short
_int	int
_integer	int
_double	double
_float	float
_boolean	boolean
string	String
byte	Byte
long	Long
short	Short
int	Integer
integer	Integer
double	Double
float	Float
boolean	Boolean
date	Date
decimal	BigDecimal
bigdecimal	BigDecimal
object	Object
map	Map
hashmap	HashMap
list	List
arraylist	ArrayList
collection	Collection
iterator	Iterator

typeHandlers

Cuando MyBatis establece el valor de un parámetro de un *PreparedStatement* u obtiene un valor de un *ResultSet*, se utiliza un *TypeHandler* para convertir el valor al tipo Java apropiado. La siguiente tabla recoge los TypeHandlers predefinidos.

Type Handler	Tipo Java	Tipo JDBC
BooleanTypeHandler	Boolean, boolean	Cualquiera compatible con BOOLEAN
ByteTypeHandler	Byte, byte	Cualquiera compatible con NUMERIC o BYTE
ShortTypeHandler	Short, short	Cualquiera compatible con NUMERIC o SHORT INTEGER
IntegerTypeHandler	Integer, int	Cualquiera compatible con NUMERIC o INTEGER
LongTypeHandler	Long, long	Cualquiera compatible con NUMERIC o LONG INTEGER
FloatTypeHandler	Float, float	Cualquiera compatible con NUMERIC o FLOAT
DoubleTypeHandler	Double, double	Cualquiera compatible con NUMERIC o DOUBLE
BigDecimalTypeHandler	BigDecimal	Cualquiera compatible con NUMERIC o

		DECIMAL
StringTypeHandler	String	CHAR, VARCHAR
ClobTypeHandler	String	CLOB, LONGVARCHAR
NStringTypeHandler	String	NVARCHAR, NCHAR
NClobTypeHandler	String	NCLOB
ByteArrayTypeHandler	byte[]	Cualquiera compatible con byte stream
BlobTypeHandler	byte[]	BLOB, LONGVARBINARY
DateTypeHandler	Date (java.util)	TIMESTAMP
DateOnlyTypeHandler	Date (java.util)	DATE
TimeOnlyTypeHandler	Date (java.util)	TIME
SqlTimestampTypeHandler	Timestamp (java.sql)	TIMESTAMP
SqlDateTypeHadler	Date (java.sql)	DATE
SqlTimeTypeHandler	Time (java.sql)	TIME
ObjectTypeHandler	Any	OTHER, o no especificado
EnumTypeHandler	Enumeration Type	VARCHAR – Cualquiera compatible con string porque se guarda el código (no el índice).

Es posible sobrescribir los *TypeHanders* o crear *TypeHanders* personalizados para tratar tipos no soportados o no estándares. Para conseguirlo, debes simplemente implementar la interfaz *TypeHandler* (org.mybatis.type) y mapear tu nuevo *TypeHandler* a un tipo Java y opcionalmente a un tipo JDBC. Por ejemplo:

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

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

Al usar este *TypeHandler* se sobrescribe el *TypeHandler* existente para los tipos String y los parámetros y resultados VARCHAR. Observa que MyBatis no introspecciona la base de datos para conocer el tipo así

que debes especificar que se trata de un VARCHAR en los mapeos de parámetros y resultados para que se use el *TypeHandler* adecuado. Esto se debe a que MyBatis no conoce nada sobre los tipos de datos hasta que la sentencia ha sido ejecutada.

objectFactory

Cada vez que MyBatis crea una nueva instancia de un objeto de retorno usa una instancia de *ObjectFactory* para hacerlo. El *ObjectFactory* por defecto no hace mucho más que instanciar la clase destino usando su constructor por defecto, o el constructor que se ha parametrizado en su caso. Es posible sobrescribir el comportamiento por defecto creando tu propio *ObjectFactory*. Por ejemplo:

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

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

La interfaz *ObjectFactory* es muy sencilla. Contiene solo dos métodos de creación, uno para el constructor por defecto y otro para el constructor parametrizado. Adicionalmente el método setProperties sirve para configurar el *ObjectFactory*. Las propiedades definidas en el cuerpo del elemento *objectFactory* se pasan al método *setProperties* después de que el *ObjectFactory* haya sido inicializado.

plugins

MyBatis permite interceptar las llamadas en ciertos puntos de la ejecución de un *mapped statement*. Por defecto, MyBatis permite incluir plugins que intercepten las llamadas de:

- Executor
  (update, query, flushStatements, commit, rollback, getTransaction, close, isClosed)
- ParameterHandler
  (getParameterObject, setParameters)

- ResultSetHandler
  (handleResultSets, handleOutputParameters)

- StatementHandler
  (prepare, parameterize, batch, update, query)

Los detalles de estos métodos se pueden conocer observando sus signaturas y el código fuente de los mismos que está disponible en el sitio de MyBatis. Es recomendable que comprendas el funcionamiento del método que estas sobrescribiendo siempre que vayas a hacer algo más complejo que monitorizar llamadas. Ten en cuenta que si modificas el comportamiento de alguno de estos métodos existe la posibilidad de que rompas el funcionamiento de MyBatis. Estas clases son de bajo nivel y por tanto debes usar los *plugins* con cuidado.

Utilizar un *plugin* es muy sencillo para la potencia que ofrecen. Simplemente implementa el interfaz Interceptor y asegúrate de especificar las signaturas que quieres interceptar.

```
// ExamplePlugin.java
@Intercepts({@Signature(
      type= Executor.class,
      method = "update",
      args = {MappedStatement.class,Object.class})})
public class ExamplePlugin implements Interceptor {
  public Object intercept(Invocation invocation) throws Throwable {
    return invocation.proceed();
  }
  public Object plugin(Object target) {
    return Plugin.wrap(target, this);
  }
  public void setProperties(Properties properties) {
  }
}

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

El plugin anterior interceptará cualquier llamada al método “update” en la instancia de Executor, que es un objeto interno que se encarga de la ejecución a bajo nivel de los *mapped statements*.

Acerca de sobre escribir la clase Configuration

Además de modificar el comportamiento de MyBatis mediante los plugins, también es posible sobrescribir la clase Configuración por completo. Extiende la clase, sobrescribe sus métodos y pásala como parámetro en la llamada al método sqlSessionFactoryBuilder.build(myConfig). Nuevamente, ten cuenta que esto puede afectar seriamente al funcionamiento de MyBatis así que úsalo con cuidado.

environments

En MyBatis pueden configurarse varios entornos. De esta forma puedes usar tus SQL Maps en distintas bases de datos por muchos motivos. Por ejemplo puede que tengas una configuración distinta para tus entornos de desarrollo, pruebas y producción. O quizá tengas varias bases de datos en producción que comparten el esquema y quieres usar los mismos SQL maps sobre todas ellas. Como ves, hay muchos casos.

Debes recordar un asunto importante. Cuando configures varios entornos, solo será posible usar UNO por cada instancia de SqlSessionFactory.

Por lo tanto, si quieres conectar a dos bases de datos, deberás crear dos instancias de SqlSessionFactory, una para cada cual. Para el caso de tres bases de datos necesitarás tres instancias y así sucesivamente. Es fácil de recordar:

- ⇨ **Una instancia de SqlSessionFactory por base de datos**

Para indicar qué entorno debe utilizarse, debes informar el parámetro opcional correspondiente en la llamada al SqlSessionFactoryBuilder. Existen dos signaturas que aceptan el entorno:

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

Si se omite el entorno se usará el entorno por defecto:

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

El elemento *environments* contiene la configuración del entorno:

```
<environments default="development">
  <environment id="development">
    <transactionManager type="JDBC">
      <property name="..." value="..."/>
    </transactionManager>
    <dataSource type="POOLED">
      <property name="driver" value="${driver}"/>
      <property name="url" value="${url}"/>
      <property name="username" value="${username}"/>
      <property name="password" value="${password}"/>
    </dataSource>
  </environment>
</environments>
```

Observa que las secciones importantes son:

- El ID del entorno por defecto (ej. default=”development”).

- El ID de de cada entorno definido (ej. id="development").
- La configuración del TransactionManager (ej. type="JDBC")
- La configuración del DataSource (ej. type="POOLED")

El ID del entorno por defecto y de los entornos existentes son auto-explicativos. Puedes nombrarlos como más te guste, tan sólo asegúrate de que el valor por defecto coincide con un entorno existente.

transactionManager

MyBatis incluye dos tipos de TransactionManager (ej. type="[JDBC|MANAGED]"):

- **JDBC** – Este TransactionManager simplemente hace uso del las capacidades de commit y rollback de JDBC. Utiliza la conexión obtenida del DataSource para gestionar la transacción.
- **MANAGED** – Este TransactionManager no hace nada. No hace commit ni rollback sobre la conexión. En su lugar, permite que el contenedor gestione el ciclo de vida completo de la transacción (ej. Spring o un servidor de aplicaciones JEE). Por defecto cierra la conexión. Sin embargo, algunos contenedores no esperan que la conexión se cierre y por tanto, si necesitas cambiar este comportamiento, informa la propiedad closeConnection a false. Por ejemplo:

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

Ninguno de estos TransactionManagers necesita ninguna propiedad. Sin embargo ambos son Type Aliases, es decir, en lugar de usarlos puedes informar el nombre totalmente cualificado o el Type Alias de tu propia implementación del interfaz TransactionFactory:

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

Todas las propiedades que configures en el XML se pasarán al método setProperties() tras la instanciación de la clase. Tu implementación debe crear una implementación de Transaction, que a su vez es también un interfaz muy sencillo:

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

Con estos dos interfaces puedes personalizar por completo la forma en la que MyBatis gestiona las transacciones.

dataSource

El elemento dataSource sirve para configurar la forma de obtener conexiones JDBC mediante la interfaz DataSource JDBC estándar.

⇨ La mayoría de las aplicaciones que usen MyBatis configurarán el dataSource como se muestra en el ejemplo. Sin embargo, esta configuración no es obligatoria. Ten en cuenta, aun así, que el dataSource es necesario para utilizar Lazy Loading.

Hay tres tipos de dataSources pre-construidos (ej. type="????"):

UNPOOLED – Esta implementación de DataSource abre y cierra una conexión JDBC cada vez que se solcita una conexión. Aunque es un poco lento, es una buena elección para aplicaciones que no necesitan la velocidad de tener conexiones abiertas de forma inmediata. Las bases de datos tienen un rendimiento distinto en cuanto al rendimiento que aportan con este tipo de DataSource, para algunas de ellas no es muy importante tener un *pool* y por tanto esta configuración es apropiada. El DataSource UNPOOLED tiene cinco opciones de configuración:

- **driver** – El nombre completamente cualificado de la clase java del driver JDBC (NO de la clase DataSource en el caso de que tu driver incluya una).
- **url** – La URL de la instancia de base de datos.
- **username** – El usuario de conexión
- **password** - La password de conexión.
- **defaultTransactionIsolationLevel** – El nivel de aislamiento por defecto con el que se crearán las conexiones.

Opcionalmente, puedes también pasar propiedades al driver de la base de datos. Para ello prefija las propiedades con "driver.", por ejemplo:

- **driver.encoding=UTF8**

Esto pasaría la propiedad "encoding" con el valor "UTF8" al driver de base datos mediante el método DriverManager.getConnection(url, driverProperties).

POOLED – Esta implementación de DataSource hace usa un pool de conexiones para evitar el tiempo necesario en realizar la conexión y autenticación cada vez que se solicita una nueva instancia de conexión. Este es un enfoque habitual en aplicaciones Web concurrentes para obtener el mejor tiempo de respuesta posible.

Además de las propiedades de (UNPOOLED) hay otras muchas propiedades que se pueden usar para configurar el DataSource POOLED:

- **poolMaximumActiveConnections** – Número máximo de conexiónes activas que pueden existir de forma simultánea. Por defecto: 10
- **poolMaximumIdleConnections** – Número máximo de conexiones libres que pueden existir de forma simultánea.
- **poolMaximumCheckoutTime** – Tiempo máximo que puede permanecer una conexión fuera del pool antes de que sea forzosamente devuelta. Por defecto: 20000ms (20 segundos)
- **poolTimeToWait** – Este es un parámetro de bajo nivel que permite escribir un log y reintentar la adquisición de una conexión en caso de que no se haya conseguido la conexión transcurrido un tiempo razonable (esto evita que se produzcan fallos constantes y silenciosos si el pool está mal configurado). Por defecto: 20000ms (20 segundos)
- **poolPingQuery** – La query de ping (sondeo) que se envía a la base de datos para verificar que la conexión funciona correctamente y que está lista para aceptar nuevas peticiones de conexión. El valor por defecto es "NO PING QUERY SET", que hará que la mayoría de los drivers de base de datos devuelvan un error con un mensaje de error decente.
- **poolPingEnabled** – Habilita o inhabilita la query de ping. Si está habilitada deberías informar también la propiedad poolPingQuery con una sentencia SQL (preferentemente una rápida). Por defecto: false.
- **poolPingConnectionsNotUsedFor** – Configura la frecuencia con la que se ejecutará la sentencia poolPingQuery. Normalmente se iguala al timeout de la conexión de base de datos para evitar pings innecesarios. Por defecto: 0 (todas las conexiones se testean continuamente – solo si se ha habilitado poolPingEnabled).

JNDI – Esta implementación de DataSource está pensada para ser usada en contenedores como Spring o los servidores de aplicaciones JEE en los que es posible configurar un DataSource de forma externa y alojarlo en el contexto JNDI. Esta configuración de DataSource requiere solo dos propiedades:

- **initial_context –** Propiedad que se usa para realizar el lookup en el InitialContext (initialContext.lookup(initial_context)). Esta propiedad es opcional, si no se informa, se buscará directamente la propiedad data_source.
- **data_source –** Es el contexto donde se debe buscar el DataSource. El DataSource se buscará en el contexto resultado de buscar data_source en el InitialContext o si no se ha informado la propiedad se buscará directamente sobre InitialContext.

Al igual que en las otras configuraciones de DataSource. Es posible enviar propiedades directamente al InitialContext prefijando las propiedades con “env.”, por ejemplo:

- env.encoding=UTF8

Enviará la propiedad “encoding” y el valor “UTF-8” al constructor del InitialContext durante su instanciación.

mappers

Ahora que se ha configurado el comportamiento de MyBatis con todos los elementos de configuración comentados estamos listos para definir los SQL mapped statements (sentencias SQL mapeadas). Primeramente necesitaremos indicarle a MyBatis dónde encontrarlos. Java no ofrece muchas posibilidades de auto-descubrimiento así que la mejor forma es simplemente decirle a MyBatis donde encontrar los ficheros de mapeo. Puedes utilizar referencias tipo classpath, o tipo path o referencias url completamente cualificadas (incluyendo file:///) . Por ejemplo:

```
// Using classpath relative resources
<mappers>
    <mapper resource="org/mybatis/builder/AuthorMapper.xml"/>
    <mapper resource="org/mybatis/builder/BlogMapper.xml"/>
    <mapper resource="org/mybatis/builder/PostMapper.xml"/>
</mappers>

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

Esta configuración solo indica a MyBatis cuáles son los ficheros de mapeo. El resto de la configuración se encuentra dentro de estos ficheros, y eso es de lo que hablaremos en el siguiente apartado.

SQL Map XML Files

La potencia de MyBatis reside en los Mapped Statements. Aquí es donde está la magia. Para lo potentes que son, los ficheros XML de mapeo son relativamente simples. Sin duda, si los comparas con el código JDBC equivalente comprobarás que ahorras el 95% del código.

Los ficheros XML de mapeos SQL solo tienen unos pocos elementos de alto nivel (en el orden en el que deberían definirse):

- **cache** – Configuración de la caché para un namespace.
- **cache-ref** – Referencia a la caché de otro namespace.
- **resultMap** – El elemento más complejo y potente que describe como cargar tus objetos a partir de los ResultSets.

- ~~**parameterMap –** Deprecada! Antigua forma de mapear parámetros. Se recomienda el uso de parámetros en línea. Este elemento puede ser eliminado en futuras versiones. No se ha documentado en este manual.~~
- **sql** – Un trozo de SQL reusable que puede utilizarse en otras sentencias.
- **insert** – Una sentencia INSERT.
- **update** – Una sentencia UPDATE.
- **delete** – Una sentencia DELETE.
- **select** – Una sentencia SELECT.

Las siguientes secciones describen estos elementos en detalle, comenzando con los propios elementos.

select

El *select statement* es uno de los elementos que más utilizarás en MyBatis. No es demasiado útil almacenar datos en la base de datos si no puedes leerlos, de hecho las aplicaciones suelen leer bastantes más datos de los que modifican. Por cada *insert*, *update* o *delete* posiblemente haya varias *selects*. Este es uno de los principios básicos de MyBatis y la razón por la que se ha puesto tanto esfuerzo en las consultas y el mapeo de resultados. El *select statement* es bastante simple para los casos simples. Por ejemplo:

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

Esta sentencia se llama "selectPerson", recibe un parámetro de tipo in (o Integer), y devuelve una HashMap usando los nombres de columna como clave y los valores del registro como valores.

Observa la notación utilizada para los parámetros:

```
#{id}
```

Esto le indica a MyBatis que cree un parámetro de PreparedStatement. Con JDBC, ese parámetro iría identificado con una "?" en la select que se le pasa al PreparedStatement, algo así:

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

JDBC requiere mucho más código para extraer los resultados y mapearlos a una instancia de un objetos, que es precisamente lo que MyBatis evita que tengas que hacer. Aun queda mucho por conocer sobre los parámetros y el mapeo de resultados. Todos sus detalles merecen su propio capítulo, y serán tratados más adelante.

El *select statement* tiene más atributos que te permiten configurar como debe comportarse cada *select statement.*

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

Atributo	**Descripción**
id	Un identificador único dentro del namespace que se utiliza para identificar el statement.
parameterType	El nombre completamente cualificado de la clase o el alias del parámetro que se pasará al statement.
~~parameterMap~~	~~Este es un atributo obsoleto que permite referenciar a un elemento parameterMap externo. Se recomienda utilizar mapeos en línea (*in line*) y el atributo parameterType.~~
resultType	El nombre completamente cualificado o el alias del tipo de retorno de este statement. Ten en cuenta que en el caso de las colecciones el parámetro debe ser el tipo contenido en la colección, no el propio tipo de la colección. Puedes utilizar resultType o resultMap, pero no ambos.
resultMap	Una referencia una un resultMap externo. Los resultMaps son la característica más potente de MyBatis, con un conocimiento detallado de los mismos, se pueden resolver muchos casos complejos de mapeos. Puedes utilizar resultMap O resultType, pero no ambos.
flushCache	Informar esta propiedad a true hará que la caché se vacíe cada vez que se llame a este statement. Por defecto es false para select statements.
useCache	Informar esta propiedad a true hará que los resultados de la ejecución de este statement se cacheen. Por defecto es true para las select statements.
timeout	Establece el tiempo máximo que el driver esperará a que la base de datos le devuelva una respuesta antes de lanzar una excepción. Por defecto: no informado (depende del driver de base de datos).
fetchSize	Este es un atributo que "sugiere" al driver que devuelva los resultados en bloques de filas en el número indicado por el parámetro. Por defecto: no informado (depende del driver de base de datos).
statementType	Puede valer STATEMENT, PREPARED o CALLABLE. Hace que MyBatis use Statement, PreparedStatement o CallableStatement respectivamente. Por defecto: PREPARED.
resultSetType	Puede valer FORWARD_ONLY|SCROLL_SENSITIVE|SCROLL_INSENSITIVE. Por defecto: no informado (depende del driver de base de datos).

insert, update, delete

Los insert, update y delete statements son muy similares en su implementación:

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

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

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

Atributo	Descripción
Id	Un identificador único dentro del namespace que se utiliza para identificar el statement.
parameterType	El nombre completamente cualificado de la clase o el alias del parámetro que se pasará al statement.
~~parameterMap~~	~~Método deprecado de referirse a un parameterMap externo. Usa mapeos inline y el atributo paramterType.~~
flushCache	Informar esta propiedad a true hará que la caché se vacíe cada vez que se llame a este statement. Por defecto es false para select statements.
timeout	Establece el tiempo máximo que el driver esperará a que la base de datos le devuelva una respuesta antes de lanzar una excepción. Por defecto: no informado (depende del driver de base de datos).
statementType	Puede valer STATEMENT, PREPARED o CALLABLE. Hace que MyBatis use Statement, PreparedStatement o CallableStatement respectivamente. Por defecto: PREPARED.
useGeneratedKeys	(solo en insert) Indica a MyBatis que utilice el método getGeneratedKeys de JDBC para recuperar las claves autogeneras automáticamente por la base de datos. (ej. campos autoincrementales en SGBD como MySQL o SQL Server). Por defecto: false
keyProperty	(solo en insert) Indica la propiedad a la que MyBatis debe asignar la clave autogenerada devuelva por getGeneratedKeys o por un elemento hijo de tipo selectKey. Por defecto: no informado.

A continuación se muestran unos ejemplos de insert, update y delete.

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

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

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

Tal y como se ha indicado, insert es algo más complejo dado que dispone de algunos atributos extra para gestionar la generación de claves de varias formas distintas.

Primeramente, si tu base de datos soporta la auto-generación de claves (ej. MySQL y SQL Server), entonces puedes simplemente informar el atributo useGeneratedKeys="true" e informar también en keyProperty el nombre del la propiedad donde guardar el valor y ya has terminado.
Por ejemplo, si la columna id de la tabla *Author* del ejemplo siguiente fuera autogenerada el insert statement se escribiría de la siguiente forma:

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

MyBatis puede tratar las claves autogeneradas de otra forma para el caso de las bases de datos que no soportan columnas autogeneradas, o porque su driver JDBC no haya incluido aun dicho soporte.

A continuación se muestra un ejemplo muy simple que genera un id aleatorio (algo que posiblemente nunca harás pero que demuestra la flexibilidad de MyBatis y cómo MyBatis ignora la forma en la que se consigue la clave):

```
<insert id="insertAuthor" parameterType="domain.blog.Author">
  <selectKey keyProperty="id" resultType="int" order="BEFORE">
    select CAST(RANDOM()*1000000 as INTEGER) a from SYSIBM.SYSDUMMY1
  </selectKey>
  insert into Author
      (id, username, password, email,bio, favourite_section)
  values
      (#{id}, #{username}, #{password}, #{email}, #{bio},
      #{favouriteSection,jdbcType=VARCHAR}
  )
</insert>
```

En el ejemplo anterior, el selectKey statement se ejecuta primero, la propiedad id de Author se informará y posteriormente se invocará al insert statement. Esto proporciona un comportamiento similar a la generación de claves en base de datos sin complicar el código Java.

El elemento selectKey tiene el siguiente aspecto:

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

Atributo	Descripción
keyProperty	La propiedad destino con la que debe informarse el resultado del selectKey statement.
resultType	El tipo de retorno. MyBatis puede adivinarlo pero no está de más añadirlo para asegurarse. MyBatis permite usar cualquier tipo simple, incluyendo Strings.
order	Puede contener BEFORE o AFTER. Si se informa a BEFORE, entonces la obtención de la clave se realizará primero, se informará el campo indicado en keyProperty y se ejecutará la insert. Si se informa a AFTER se ejecuta primero la insert y después la selectKey – Esto es habitual en bases de datos como Oracle que soportan llamadas embebidas a secuencias dentro de una sentencia insert.
statementType	Al igual que antes, MyBatis soporta sentencias de tipo STATEMENT, PREPARED and CALLABLE que corresponden Statement, PreparedStatement y CallableStatement respectivamente.

sql

Este elemento se utiliza para definir un fragmento reusable de código SQL que puede ser incluido en otras sentencias. Por ejemplo:

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

Este fragmento de SQL puede ser incluido en otra sentencia, por ejemplo:

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

Parameters

En todos los statements anteriores se han mostrado ejemplos de parámetros simples. Los parámetros son elementos muy potentes en MyBatis. En los casos simples, probablemente el 90% de los casos, no hay mucho que decir sobre ellos, por ejemplo:

```
<select id="selectUsers" parameterType="int" resultType="User">
```

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

El ejemplo anterior demuestra un mapeo muy simple de parámetro con nombre. El atributo parameterType se ha informado a “int”, por lo tanto el nombre del parámetro puede ser cualquiera. Los tipos primitivos y los tipos de datos simples como Integer o String no tienen propiedades relevantes y por tanto el parámetro será reemplazado por su valor. Sin embargo, si pasas un objeto complejo, entonces el comportamiento es distinto. Por ejemplo:

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

Si se pasa un objeto de tipo User como parámetro en este statement, se buscarán en él las propiedades *id, username* y *password* y sus valores se pasarán como parámetros de un PreparedStatement.

Esta es una Buena forma de pasar parámetros a statements. Pero los parameter maps (mapas de parámetros) tienen otras muchas características.

Primeramente, es posible especificar un tipo de dato concreto.

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

Como en otros casos, el tipo de Java (javaType) puede casi siempre obtenerse del objeto recibido como parámetro, salvo si el objeto es un HashMap. En ese caso debe indicarse el javaType para asegurar que se usa el TypeHandler correcto.

➔ **Nota:** El tipo JDBC es obligatorio para todas las columnas que admiten null cuando se pasa un null como valor. Puedes investigar este tema por tu cuenta leyendo los JavaDocs del método PreparedStatement.setNull().

Si quieres customizar aun más el tratamiento de tipos de datos, puedes indicar un TypeHandler específico (o un alias), por ejemplo:

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

Comienza a parecer demasiado verboso, pero lo cierto es que rara vez necesitaras nada de esto.

Para los tipos numéricos hay un atributo numericScale que permite especificar cuantas posiciones decimales son relevantes.

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

Finalmente, el atributo *mode* te permite especificar parámetros IN, OUT o INOUT. Si un parámetro es OUT o INOUT, el valor actual de las propiedades del objeto pasado como parámetro será modificado. Si el mode=OUT (o INOUT) y el jdbcType=CURSOR (ej. Oracle REFCURSOR), debes especificar un resultMap para mapear el RestultSet al tipo del parámetro. Ten en cuenta que el atributo javaType es opcional en

este caso, dado que se establecerá automáticamente al valor ResultSet en caso de no haberse especificado si el jdbcType es CURSOR.

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

MyBatis también soporta tipos de datos avanzados como los *structs*, pero en este caso debes indicar in el statement el jdbcTypeName en la declaración del parámetro de tipo OUT. Por ejemplo:

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

A pesar de estas potentes opciones, la mayoría de las veces simplemente debes especificar el nombre de la propiedad y MyBatis adivinará lo demás. A lo sumo, deberás especificar los jdbcTypes para las columnas que admiten nulos.

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

Sustitución de Strings

Por defecto, usar la sintaxis #{} hace que MyBatis genere propiedades de PreparedStatement y que asigne los valores a parámetros de PreparedStatement de forma segura (ej. ?). Aunque esto es más seguro, más rápido y casi siempre la opción adecuada, en algunos casos sólo quieres inyectar un trozo de texto sin modificaciones dentro de la sentencia SQL. Por ejemplo, para el caso de ORDER BY, podrías utilizar algo así:

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

En este caso MyBatis no alterará el contenido del texto.

➔ **IMPORTANTE:** No es seguro recoger un texto introducido por el usuario e inyectarlo en una sentencia SQL. Esto permite ataques de inyección de SQL y por tanto debes impedir que estos campos se informen con la entrada del usuario, o realizar tus propias comprobaciones o escapes.

resultMap

El elemento resultMap es el elemento más importante y potente de MyBatis. Te permite eliminar el 90% del código que requiere el JDBC para obtener datos de ResultSets, y en algunos casos incluso te permite hacer cosas que no están siquiera soportadas en JDBC. En realidad, escribir un código equivalente para realizar algo similar a un mapeo para un statement complejo podría requerir cientos de líneas de código. El diseño de los ResultMaps es tal, que los statemets simples no requieren un ResultMap explícito, y los statements más complejos requieren sólo la información imprescindible para describir relaciones.

Ya has visto algunos ejemplos de un statement sencillo que no requiere un ResultMap explícito. Por ejemplo:

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

Este statement simplemente obtiene como resultado una HashMap que contiene como claves todas las columnas, tal y como se ha especificado en el atributo resultType. Aunque es muy útil en muchos casos, una HashMap no contribuye a un buen modelo de dominio. Es más probable que tu aplicación use JavaBeans o POJOs (Plain Old Java Objects) para el modelo de dominio. MyBatis soporta ambos. Dado el siguiente JavaBean:

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

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

Basándose en la especificación de JavaBeans, la clase anterior tiene 3 propiedades: ip, username y hashedPassword. Todas ellas coinciden exactamente con los nombres de columna en la sentencia select.

Este JavaBean puede mapearse desde un ResultSet de forma casi tan sencilla como la HashMap.

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

Y recuerda que los TypeAliases son tus amigos. Úsalos y de esa forma no tendrás que escribir constantemente el nombre totalmente cualificado (fully qualified). Por ejemplo:

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

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

En estos casos MyBatis crea automáticamente un RestulMap entre bastidores para mapear las columnas a las propiedades del JavaBean en base a sus nombres. Si los nombres de las columnas no coinciden exactamente, puedes emplear alias en los nombres de columnas de la sentencia SQL (una característica estándar del SQL) para hacer que coincidan. Por ejemplo:

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

Lo mejor de los ResultMaps se que ya has aprendido mucho sobre ellos y ni siquiera los has visto! Los casos sencillos no requieren nada más que lo que ya has visto. Solo como ejemplo, veamos qué aspecto tendría el último ejemplo utilizando un ResultMap externo, lo cual es otra forma de solucionar las divergencias entre los nombres de columnas y de propiedades.

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

Y el statement que las referencia utiliza para ello el atributo resultMap (fíjate que hemos eliminado el atributo resultType). Por ejemplo:

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

Ojala todo fuera tan sencillo.

Mapeo de resultados avanzado

MyBatis fue creado con una idea en mente: las bases de datos no siempre son como a ti te gustaría que fueran. Nos encantaría que todas las bases de datos estuvieran en 3ª forma normal o BCNF, pero no lo están. Sería genial que una base de datos encajara perfectamente con todas las aplicaciones que la usan pero no es así. Los ResultMaps son la respuesta de MyBatis a este problema.

Por ejemplo, como mapearías este statement?

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

Posiblemente te gustaría mapearlo a un modelo de objetos formado por un Blog que ha sido escrito por un Autor, y tiene varios Posts, cada uno de ellos puede tener cero o varios comentarios y tags. A continuación puede observarse un ResultMap complejo (asumimos que Author, Blog, Post, Comments y Tags son typeAliases). Échale un vistazo, pero no te preocupes, iremos paso a paso. Aunque parece enorme, es en realidad, bastante sencillo.

```
<!-- Very Complex Result Map -->
<resultMap id="detailedBlogResultMap" type="Blog">
  <constructor>
    <idArg column="blog_id" javaType="int"/>
  </constructor>
  <result property="title" column="blog_title"/>
  <association property="author" column="blog_author_id" javaType=" Author">
    <id property="id" column="author_id"/>
    <result property="username" column="author_username"/>
```

```
    <result property="password" column="author_password"/>
    <result property="email" column="author_email"/>
    <result property="bio" column="author_bio"/>
    <result property="favouriteSection" column="author_favourite_section"/>
  </association>
  <collection property="posts" ofType="Post">
    <id property="id" column="post_id"/>
    <result property="subject" column="post_subject"/>
    <association property="author" column="post_author_id" javaType="Author"/>
    <collection property="comments" column="post_id" ofType=" Comment">
      <id property="id" column="comment_id"/>
    </collection>
    <collection property="tags" column="post_id" ofType=" Tag" >
      <id property="id" column="tag_id"/>
    </collection>
    <discriminator javaType="int" column="draft">
      <case value="1" resultType="DraftPost"/>
    </discriminator>
  </collection>
</resultMap>
```

El elemento resultMap tiene varios sub-elementos y una estructura que merece la pena comentar. A continuación se muestra una vista conceptual del elemento resultMap.

resultMap

- **constructor** – usado para inyectar resultados en el constructor de la clase durante la instanciación
- **idArg** – argumento ID; marcar el argumento ID mejora el rendimiento
- **arg** – un resultado normal inyectado en el constructor

- **id** – result ID; marcar los results con ID mejora el rendimiento
- **result** – un resultado normal inyectado en un campo o una propiedad de un JavaBean
- **association** – una asociación con un objeto complejo; muchos resultados acabarán siendo de este tipo:
  - *result mapping anidado* – las asociaciones son resultMaps en sí mismas o pueden apuntar a otro resultMap.
- **collection** – a una colección de tipos complejos
  - *result mapping anidado* – las asociaciones son resultMaps en sí mismas o pueden
- **discriminator** – utiliza un valor del resultado para determinar qué resultMap utilizar
  - **case** – un resultMap basado en un valor concreto
    - *result mapping anidado* – un case es un resultMap en sí mismo y por tanto puede contener a su vez elementos propios de un resultMap o bien apuntar a un resultMap externo.

➔ **Buena práctica:** Construye los ResultMaps de forma incremental. Las pruebas unitarias son de gran ayuda en ellos. Si intentas construir un ResultMap gigantesco como el que se ha visto anteriormente, es muy probable que lo hagas mal y será difícil trabajar con él. Comienza con una versión sencilla y evolucionarla paso a paso. Y haz pruebas unitarias! La parte negativa de utilizar frameworks es que a veces son una caja negra (sean opensource o no). Lo mejor que puedes hacer para asegurar que estás consiguiendo el comportamiento que pretendes es escribir pruebas unitarias. También son de utilidad para enviar bugs.

Las próximas secciones harán un recorrido por cada uno de los elementos en detalle.

id, result

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

Estos son los ResultMaps más sencillos. Ambos *id,* y *result* mapean una columna con una propiedad o campo de un tipo de dato simple (String, int, double, Date, etc.).

La única diferencia entre ambos es que *id* marca que dicho resultado es un identificador y dicha propiedad se utilizará en las comparaciones entre instancias de objetos. Esto mejora el rendimiento global y especialmente el rendimiento de la cache y los mapeos anidados (ej. mapeo de joins).

Cada uno tiene los siguientes atributos:

Atributo	Descripción
property	El campo o propiedad al que se va a mapear al valor resultado. Si existe una propiedad tipo JavaBean para el nombre dado, se utilizará. En caso contrario MyBatis buscará un campo con el mismo nombre. En ambos casos puedes utilizar navegación compleja usando la notación habitual con puntos. Por ejemplo, puedes mapear a algo sencillo como: "username", o a algo más complejo como: "address.street.number".
column	El nombre de la columna de la base de datos, o el alias de columna. Es el mismo string que se pasaría al método resultSet.getString(columnName).
javaType	Un nombre de clase Java totalmente cualificado, o un typeAlias (más adelante se indican los typeAlias predefinidos). Normalmente MyBatis puede adivinar el tipo de datos de una propiedad de un JavaBean. Sin embargo si usas una HashMap deberás especificar el javaType para obtener el comportamiento deseado.
jdbcType	Un tipo JDBC de los tipos soportados que se muestran a continuación. **El tipo JDBC solo se requiere para columnas que admiten nulos en insert, update o delete.** Esto es un requerimiento de JDBC no de MyBatis. Incluso si usas JDBC directamente debes especificar el tipo – pero solo para los valores que pueden ser nulos.
typeHandler	Ya hemos hablado sobre typeHandlers anteriormente. Usando esta propiedad se puede sobre escribir el typeHandler por defecto. El valor admite un nombre totalmente cualificado o un alias.

Tipo JDBC soportados

Para referencia futura, MyBatis soporta los siguientes tipos JDBC por medio de la enumeración JdbcType.

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

constructor

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

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

Aunque las propiedades funcionan bien en clases tipo Data Transfer Object (DTO), y posiblemente en la mayor parte de tu modelo de dominio, hay algunos casos en los que puedes querer clases inmutables. En ocasiones, las tablas que contienen información que nunca o raramente cambia son apropiadas para las clases inmutables. La inyección en el constructor te permite informar valores durante la instanciación de la clase, sin necesidad de exponer métodos públicos. MyBatis tambien soporta propiedades privadas para conseguir esto mismo pero habrá quien prefiera utilizar la inyección de Constructor. El elemento *constructor* permite hacer esto.

Dado el siguiente constructor:

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

Para poder inyectar los resultados en el constructor, MyBatis necesita identificar el constructor por el tipo de sus parámetros. Java no tiene forma de obtener los nombres de los parámetros de un constructor por introspección. Por tanto al crear un elemento *constructor*, debes asegurar que los argumentos están correctamente ordenados y que has informado los tipos de datos.

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

El resto de atributos son los mismos que los de los elementos *id* y *result.*

Atributo	Descripción
column	El nombre de la columna de la base de datos, o el alias de columna. Es el mismo string que se pasaría al método resultSet.getString(columnName).
javaType	Un nombre de clase Java totalmente cualificado, o un typeAlias (más adelante se indican los typeAlias predefinidos). Normalmente MyBatis puede adivinar el tipo de datos de una propiedad de un JavaBean. Sin embargo si usas una HashMap deberás especificar el javaType para obtener el comportamiento deseado.
jdbcType	Un tipo JDBC de los tipos soportados que se muestran a continuación. **El tipo JDBC solo se requiere para columnas que admiten nulos en insert, update o delete.** Esto es un requerimiento de JDBC no de MyBatis. Incluso si usas JDBC directamente debes especificar el tipo – pero solo para los valores que pueden ser nulos.
typeHandler	Ya hemos hablado sobre typeHandlers anteriormente. Usando esta propiedad se puede sobre escribir el typeHandler por defecto. El valor admite un nombre totalmente cualificado o un alias.
select	El id de otro mapped statement que cargará el tipo complejo asociado a este argumento. Los valores obtenidos de las columnas especificadas en el atributo *column* se pasarán como parámetros al *select* statement referenciado. Ver el elemento *association* para más información.

resultMap	El id de un resultmap que puede mapear los resultados anidados de este argumento al grafo de objetos (object graph) apropiado. Es una alternativa a llamar a otro select statement. Permite hacer join de varias tablas en un solo ResultSet. Un ResultSet de este tipo puede contener bloques repetidos de datos que deben ser descompuestos y mapeados apropiadamente a un árbol de objetos (object graph). MyBatis te permite encadenar RestultMaps para tratar resultados anidados. Ver el elemento *association* para más información.

association

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

El elemento *association* trata las relaciones de tipo "tiene-un". Por ejemplo, en nuestro ejemplo, un Blog tiene un Autor. Un mapeo *association* funciona casi como cualquier otro *result.* Debes especificar la propiedad destino, la columna de la que obtener el valor, el javaType de la propiedad (que normalmente MyBatis puede adivinar), el jdbcType si fuera necesario y un typeHandler si quieres sobre escribir el tratamiento de los valores de retorno.

Donde la *association* es distinta es en que debes indicar a MyBatis como cargar la asociación. MyBatis puede hacerlo de dos formas distintas:

- **Nested Select:** Ejecutando otra select que devuelve el tipo complejo deseado.
- **Nested Results:** Usando un ResultMap anidado que trata con los datos repetidos de resultsets provenientes de joins.

Primeramente, examinemos la propiedades del elemento. Como veras, es distinto de un ResultMap normal solo por los atributos *select* y *resultMap*.

Atributo	Descripción
property	El campo o propiedad a la que se debe mapear la columna. Si existe una propiedad JavaBean igual al nombre dado, se usará. En caso contrario, MyBatis buscará un campo con el nombre indicado. En ambos casos puedes usar navegación compleja usando la notación habitual con puntos. Por ejemplo puedes mapear algo simple como: "username", o algo más complejo como: "address.street.number".
column	El nombre de la columna de la base de datos, o el alias de columna. Es el mismo string que se pasaría al método resultSet.getString(columnName). **Nota: para tratar con claves compuestas, puedes especificar varios nombres usando esta sintaxis *column="{prop1=col1,prop2=col2}"*. Esto hará que se informen las propiedades prop1 y prop2 del objeto parámetro del select statement destino**
javaType	Un nombre de clase Java totalmente cualificado, o un typeAlias (más adelante se indican los typeAlias predefinidos). Normalmente MyBatis puede adivinar el tipo de datos de una propiedad de un JavaBean. Sin embargo si usas una HashMap deberás especificar el javaType para obtener el comportamiento deseado.
jdbcType	Un tipo JDBC de los tipos soportados que se muestran a continuación. **El tipo JDBC solo**

	se requiere para columnas que admiten nulos en insert, update o delete. Esto es un requerimiento de JDBC no de MyBatis. Incluso si usas JDBC directamente debes especificar el tipo – pero solo para los valores que pueden ser nulos.
typeHandler	Ya hemos hablado sobre typeHandlers anteriormente. Usando esta propiedad se puede sobre escribir el typeHandler por defecto. El valor admite un nombre totalmente cualificado o un alias.

Select anidada en A*ssociation*

Select	El id de otro mapped statement que cargará el tipo complejo asociado a esta propiedad. Los valores obtenidos de las columnas especificadas en el atributo *column* se pasarán como parámetros al *select* statement referenciado. A continuación se muestra un ejemplo detallado. **Nota: para tratar con claves compuestas, puedes especificar varios nombres usando esta sintaxis *column="{prop1=col1,prop2=col2}"*. Esto hará que se informen las propiedades prop1 y prop2 del objeto parámetro del select statement destino**

Por ejemplo:

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

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

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

Tenemos dos statements: uno para cargar el Blog, el otro para cargar el Autor, y el RestulMap de Blog describe que la sentencia "selectAuthor" debe utilizarse para cargar su propiedad author.

Todas las demás propiedades se cargarán automáticamente asumiendo que los nombres de propiedad y de columna coinciden.

Aunque este enfoque es simple, puede no tener un buen rendimiento con gran cantidad de datos. Este problema es conocido como "El problema de las N+1 Selects". En resumidas cuentas, el problema de N+1 selects está causado por esto:

- Tu ejecutas una sentencia SQL para obtener una lista de registros (el "+1").
- Para cada registro obtenido ejecutas una select para obtener sus detalles (el "N").

Este problema puede provocar la ejecución de cientos o miles de sentencias SQL. Lo cual no es demasiado recomendable.

MyBatis puede cargar esas consultas de forma diferida (lazy load), por lo tanto se evita el coste de lanzar todas esas consultas a la vez. Sin embargo, si cargas la lista e inmediatamente iteras por ella para

acceder a los datos anidados, acabarás cargando todos los registros y por lo tanto el rendimiento puede llegar a ser muy malo.

Así que, hay otra forma de hacerlo.

ResultMap anidado en Association

resultMap	El id de un resultmap que puede mapear los resultados anidados de esta asociación al grafo de objetos apropiado. Es una alternativa a llamar a otro select statement. Permite hacer join de varias tablas en un solo ResultSet. Un ResultSet de este tipo puede contener bloques repetidos de datos que deben ser descompuestos y mapeados apropiadamente a un árbol de objetos (object graph). MyBatis te permite encadenar RestultMaps para tratar resultados anidados. A continuación se muestra un ejemplo detallado.

Previamente has visto un ejemplo muy complejo de asociaciones anidadas. Lo que se muestra a continuación es un ejemplo más simple que demuestra cómo funciona esta característica. En lugar de ejecutar un statement separado, vamos a hacer una JOIN de las tablas Blog y Author de la siguiente forma:

```
<select id="selectBlog" parameterType="int" resultMap="blogResult">
    select
           B.id            as blog_id,
           B.title         as blog_title,
           B.author_id     as blog_author_id,
           A.id            as author_id,
           A.username      as author_username,
           A.password      as author_password,
           A.email         as author_email,
           A.bio           as author_bio
    from Blog B left outer join Author A on B.author_id = A.id
    where B.id = #{id}
  </select>
```

Fíjate en la join, y el especial cuidado que se ha dedicado a que todos los resultados tengan un alias que les de un nombre único y claro. Esto hace el mapeo mucho más sencillo. Ahora podemos mapear los resultados:

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

<resultMap id="authorResult" type="Author">
  <id property="id" column="author_id"/>
  <result property="username" column="author_username"/>
  <result property="password" column="author_password"/>
  <result property="email" column="author_email"/>
  <result property="bio" column="author_bio"/>
</resultMap>
```

En el ejemplo anterior puedes ver que la asociación “author” de Blog delega la carga de las instancias de Author en el ResultMap “authorResult”.

Muy importante: El elemento *id* tiene un papel muy importante en el mapeo de resultados anidados. Debes especificar siempre una o más propiedades que se puedan usar para identificar unívocamente los resultados. Lo cierto es que MyBatis también va a funcionar si no lo haces pero a costa de una importante penalización en rendimiento. Elige el número mínimo de propiedades que pueda identificar unívocamente un resultado. La clave primaria es una elección obvia (incluso si es compuesta).
En el ejemplo anterior se usa un resultMap externo para mapear la asociación. Esto hace que el resultMap del Autor sea reusable. Sin embargo, si no hay necesidad de reusarla o simplemente prefieres colocar todos los mapeos en un solo ResultMap, puedes anidarlo en la propia asociación. A continuación se muestra un ejemplo de este enfoque:

```
<resultMap id="blogResult" type="Blog">
  <id property=”blog_id” column="id" />
  <result property="title" column="blog_title"/>
  <association property="author" column="blog_author_id" javaType="Author">
       <id property="id" column="author_id"/>
       <result property="username" column="author_username"/>
       <result property="password" column="author_password"/>
       <result property="email" column="author_email"/>
       <result property="bio" column="author_bio"/>
  </association>
</resultMap>
```

Has visto como se utiliza la asociación “Tiene Un”. Pero ¿qué hay que el “Tiene Muchos”? Ese es el contenido de la siguiente sección.

collection

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

El elemento *collection* funciona de forma casi idéntica al *association*. En realidad, es tan similar, que documentar todas las similitudes sería redundante. Así que enfoquémonos en las diferencias.

Para continuar con nuestro ejemplo anterior, un Blog solo tiene un Autor. Pero un Blog tiene muchos Posts. En la clase Blog esto se representaría con algo como:

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

Para mapear un conjunto de resultados anidados a una Lista como esta, debemos usar el elemento *collection.* Al igual que el elemento *association,* podemos usar una select anidada, o bien resultados anidados cargados desde una join.

Select anidada en *Collection*

Primeramente, echemos un vistazo al uso de una select anidada para cargar los Posts de un Blog.

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

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

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

Hay unas cuantas diferencias que habrás visto de forma inmediata, pero la mayor parte tiene el mismo aspecto que el elemento *association* que vimos anteriormente. Primeramente, verás que estamos usando el elemento *collection.* Verás también que hay un nuevo atributo ***“ofType”***. Este atributo es necesario para distinguir el tipo de la propiedad del JavaBean (o del campo) y el tipo contenido por la colección. Por tanto podrías leer el siguiente mapeo de esta forma:

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

➔ Leído como: “Una colección de posts en un ArrayList de tipos Post.”

El javaTypes es casi siempre innecesario, porque MyBatis lo adivinará en la mayoría de los casos. Así que podrías acortarlo de esta forma:

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

ResultMaps anidados en Collection

A estas alturas, posiblemente ya imaginarás cómo funcionan los ResultMaps anidados en una colección porque funcionan exactamente igual que en una asociación, salvo por que se añade igualmente el atributo ***“ofType”.***

Primero, echemos un vistazo al SQL:

```
<select id="selectBlog" parameterType="int" resultMap="blogResult">
    select
           B.id as blog_id,
           B.title as blog_title,
           B.author_id as blog_author_id,
           P.id as post_id,
           P.subject as post_subject,
           P.body as post_body,
    from Blog B
           left outer join Post P on B.id = P.blog_id
    where B.id = #{id}
  </select>
```

Nuevamente hemos hecho una JOIN de las tablas Blog y Post, y hemos tenido cuidado de asegurarnos que las columnas obtenidas tienen un alias adecuado. Ahora, mapear un Blog y colección de Post es tan simple como:

```
<resultMap id="blogResult" type="Blog">
  <id property=”id” column="blog_id" />
```

```
  <result property="title" column="blog_title"/>
  <collection property="posts" ofType="Post">
    <id property="id" column="post_id"/>
    <result property="subject" column="post_subject"/>
    <result property="body" column="post_body"/>
  </collection>
</resultMap>
```

Nuevamente, recuerda la importancia del elemento *id,* o lee la sección de asociación si no lo has hecho aun.

Además, si prefieres el formato más largo que aporta más reusabilidad a tus ResultMaps, puedes utilizar esta forma alternativa de mapeo:

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

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

➔ **Nota:** No hay límite en profundidad, amplitud o combinaciones de las asociaciones y colecciones que mapees. Debes tener en cuenta el rendimiento cuando crees los mapeos. Las pruebas unitarias y de rendimiento de tu aplicación son de gran utilidad para conocer cuales el mejor enfoque para tu aplicación. La parte positiva es que MyBatis te permite cambiar de opinión más tarde, con muy poco (o ningún) cambio en tu código.

El mapeo de asociaciones y colecciones es un tema denso. La documentación solo puede llevarte hasta aquí. Con un poco de práctica, todo se irá aclarando rápidamente.

discriminator

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

En ocasiones una base de datos puede devolver resultados de muchos y distintos (y esperamos que relacionados) tipos de datos. El elemento *discriminator* fue diseñado para tratar esta situación, y otras como la jerarquías de herencia de clases. El discriminador es bastante fácil de comprender, dado que funciona muy parecido la sentencia *switch* de Java.

Una definición de *discriminator* especifica los atributos *column* y *javaType.* Column indica de dónde debe MyBatis obtener el valor con el que comparar. El javaType es necesario para asegurar que se utiliza el tipo de comparación adecuada (aunque la comparación de Strings posiblemente funcione casi en todos los casos). Por ejemplo:

```
<resultMap id="vehicleResult" type="Vehicle">
      <id property=”id” column="id" />
      <result property="vin" column="vin"/>
```

```
        <result property="year" column="year"/>
        <result property="make" column="make"/>
        <result property="model" column="model"/>
        <result property="color" column="color"/>
        <discriminator javaType="int" column="vehicle_type">
          <case value="1" resultMap="carResult"/>
          <case value="2" resultMap="truckResult"/>
          <case value="3" resultMap="vanResult"/>
          <case value="4" resultMap="suvResult"/>
        </discriminator>
</resultMap>
```

En este ejemplo, MyBatis obtendrá cada registro del ResultSet y comparará su valor vehicle_type. Si coincide con alguno de los casos del discriminador entonces usará el ResultMap especificado en cada caso. Esto se hace de forma exclusiva, es decir, el resto del ResultMap se ignora (a no ser que se extienda, de lo que hablaremos en un Segundo). Si no coincide ninguno de los casos MyBatis utilizará el resultmap definido fuera del bloque *discriminator.* Por tanto si carResult ha sido declarado de la siguiente forma:

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

Entonces solo la propiedad doorCount se cargará. Esto se hace así para permitir grupos de discriminadores completamente independientes, incluso que no tengan ninguna relación con el ResultMap padre. En este caso sabemos que hay relación entre coches y vehículos, dado que un coche es-un vehículo. Por tanto queremos que el resto de propiedades se carguen también, así que con un simple cambio en el ResultMap habremos terminado.

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

Ahora, se cargarán todas las propiedades tanto de vehicleResult como de carResult.

Nuevamente, hay quien puede pensar que la definición externa es tediosa. Por tanto hay una sintaxis alternativa para los que prefieran un estilo más conciso. Por ejemplo:

```
<resultMap id="vehicleResult" type="Vehicle">
        <id property=”id” column="id" />
        <result property="vin" column="vin"/>
        <result property="year" column="year"/>
        <result property="make" column="make"/>
        <result property="model" column="model"/>
        <result property="color" column="color"/>
        <discriminator javaType="int" column="vehicle_type">
          <case value="1" resultType="carResult">
            <result property=”doorCount” column="door_count" />
          </case>
          <case value="2" resultType="truckResult">
            <result property=”boxSize” column="box_size" />
            <result property=”extendedCab” column="extended_cab" />
          </case>
          <case value="3" resultType="vanResult">
            <result property=”powerSlidingDoor” column="power_sliding_door" />
```

```
          </case>
          <case value="4" resultType="suvResult">
            <result property=”allWheelDrive” column="all_wheel_drive" />
          </case>
        </discriminator>
</resultMap>
```

➔ **Recuerda** que todos estos son ResultMaps, y que si no indicas ningún result en ellos, MyBatis mapeará automáticamente las columnas a las propiedades por ti. Así que en muchos casos estos ejemplos son más verbosos de lo que realmente debieran ser. Dicho esto, la mayoría de las bases de datos son bastante complejas y muchas veces no podemos depender de ello para todos los casos.

cache

MyBatis incluye una funcionalidad de caché muy potente que es ampliamente configurable y personalizable. Se han realizado muchos cambios en la caché de MyBatis 3 para hacer la a la vez más potente y más sencilla de configurar.

Por defecto la caché no está activa, excepto la caché local de sesión, que mejora el rendimiento y que es necesaria para resolver dependencias circulares. Para habilitar un segundo nivel de caché simplemente necesitas añadir una línea en tu fichero de mapping:

```
<cache/>
```

Eso es todo literalmente. El efecto de esta sencilla línea es el siguiente:

- Todos los resultados de las sentencias **select** en el mapped statement se cachearán.
- Todas las sentencias **insert, update y delete** del mapped statement vaciarán la caché.
- La caché usarán un algoritmo de reemplazo tipo **Least Recently Used (LRU)**.
- La caché no se vaciará por tiempo (ej. **no Flush Interval**).
- La caché guardará **1024 referencias** a listas u objetos (según lo que devuelva el statement).
- La caché puede tratarse como una cache de tipo **lectura/escritura**, lo cual significa que los objetos obtenidos no se comparten y pueden modificarse con seguridad por el llamante sin interferir en otras potenciales modificaciones realizadas por otros llamantes o hilos.

Todas estas propiedades son modificables mediante atributos del elemento *cache.* Por ejemplo:

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

Esta configuración más avanzada de caché crea una cache de tipo FIFO que se vacía cada 60 segundos, guarda hasta 512 referencias a objetos o listas, y los objetos devueltos son considerados de solo lectura, esto es, que modificarlos puede crear problemas en llamantes de otros hilos.

Las políticas de reemplazo son las siguientes:

- **LRU** – Least Recently Used: Borra los objetos que llevan más tiempo sin ser usados.
- **FIFO** – First In First Out: Borra los objetos en el mismo orden en el que entraron en la caché.
- **SOFT** – Soft Reference: Borra los objetos en base a las referencias Soft del *Garbage Collector*.
- **WEAK** – Weak Reference: Es más agresivo y borra objetos basándose en el estado del *Garbage Collector* y las referencias débiles.

LRU es la política por defecto.

El atributo **flushInterval** acepta un entero positivo y debería representar un lapso de tiempo razonable en milisegundos. No está activo por defecto, por tanto no hay intervalo de vaciado y la caché solo se vacía mediante llamadas a otros statements.

El atributo **size** acepta un entero positivo, ten en cuenta el tamaño de los objetos que quieres cachear y la cantidad de memoria de la que dispone. Por defecto es 1024.

El atributo readOnly puede informarse con *true* o *false.* Una caché de solo lectura devuelve la misma instancia de objeto a todos los llamantes. Por lo tanto estos objetos no deben modificarse. Por otro lado esto proporciona una mejora en el rendimiento. Una caché de tipo lectura-escritura devuelve una copia (vía serialización) del objeto cacheado. Esto es más lento, pero más seguro, y por ello el valor por defecto es *false*.

Usando una caché personalizada

Además de poder personalizar la caché de las formas indicadas, puedes sustituir el sistema de caché por completo y proporcionar tu propia caché, o crear un adaptador para cachés de terceros.

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

Este ejemplo demuestra cómo usar una caché personalizada. La clase especificada el atributo type debe implementar el interfaz org.mybatis.cache.Cache. Este es uno de los interfaces más complejos de MyBatis pero su funcionalidad es simple.

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

```
  void clear();
  ReadWriteLock getReadWriteLock();
}
```

Para configurar tu caché añade simplemente propiedades tipo JavaBean a tu implementación, y pasa las propiedades usando el elemento *cache*, por ejemplo el siguiente ejemplo llamará a un método "setCacheFile(String file)" en tu implementación de caché:

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

Puedes utilizar propiedades JavaBean de cualquier tipo simple y MyBatis hará la conversión.

Es importante recordar que la configuración de la caches y la instancia de caché está asociadas al namespace del fichero SQL Map. Y por tanto, a todas las sentencias del mismo namespace dado que la cache está asociada a él. Los statements pueden modificar cómo interactúan con la caché, o excluirse a sí mismos completamente utilizando dos atributos simples. Por defecto los statements están configurados así:

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

Dado que estos son los valores por defecto, no deberías nunca configurar un statement de esa forma. En cambio, utiliza los atributos flushCache y useCache si quieres modificar el valor por defecto. Por ejemplo en algunos casos quieres excluir los resultados de un caché particular de la caché, o quizá quieras que un statement de tipo select vacíe la caché. O de forma similar, puede que quieras que algunas update statements no la vacíen.

cache-ref

Recuerda que en la sección anterior se indicó que la caché de un namespace sería utilizada por statements del mismo namespace. Es posible que en alguna ocasión quieras compartir la misma configuración de caché e instancia entre statements de distintos namespaces. En estos casos puedes hacer referencia a otra caché usando el elemento *cache-ref*.

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

SQL dinámico

Una de las características más potentes de MyBatis ha sido siempre sus capacidades de SQL dinámico. I tienes experiencia con JDBC o algún framework similar, entenderás que doloroso es concatenar strings

de SQL, asegurándose de que no olvidas espacios u omitir una coma al final de la lista de columnas. El SQL dinámico puede ser realmente doloroso de usar.

Aunque trabajar con SQL dinámico no va a ser nunca una fiesta, MyBatis ciertamente mejora la situación con un lenguaje de SQL dinámico potente que puede usarse en cualquier mapped statement.

Los elementos de SQL dinámico deberían ser familiares a aquel que haya usado JSTL o algún procesador de texto basado en XML. En versiones anteriores de MyBatis había un montón de elementos que conocer y comprender. MyBatis 3 mejora esto y ahora hay algo menos de la mitad de esos elementos con los que trabajar. MyBatis emplea potentes expresiones OGNL para eliminar la necesidad del resto de los elementos.

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

if

La tarea más frecuente en SQL dinámico es incluir un trozo de la clausula where condicionalmente. Por ejemplo:

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

Este statement proporciona una funcionalidad de búsqueda de texto. Si no se pasa ningún título, entonces se retornan todos los Blogs activos. Pero si se pasa un título se buscará un título como el pasado (para los perspicaces, sí, en este caso tu parámetro debe incluir el carácter de comodín)

¿Y cómo hacemos si debemos buscar opcionalmente por título o autor? Primeramente, yo cambiaría el nombre del statement para que tenga algo más de sentido. Y luego añadir otra condición.

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

choose, when, otherwise

En ocasiones no queremos usar una condición sino elegir una de entre varias opciones. De forma similar al *switch* de Java, MyBatis ofrece el elemento *choose.*

Usemos el ejemplo anterior, pero ahora vamos a buscar solamente por título si se ha proporcionado un título y por autor si se ha proporcionado un autor. Si no se proporciona ninguno devolvemos una lista de Blogs destacados (quizá una lista seleccionada por los administradores en lugar de una gran lista de blogs sin sentido).

```
<select id="findActiveBlogLike"
      parameterType="Blog" resultType="Blog">
  SELECT * FROM BLOG WHERE state = 'ACTIVE'
  <choose>
    <when test="title != null">
      AND title like #{title}
    </when>
    <when test="author != null and author.name != null">
      AND author_name like #{author.name}
    </when>
    <otherwise>
      AND featured = 1
    </otherwise>
  </choose>
</select>
```

trim, where, set

En los ejemplos anteriores se ha sorteado intencionadamente un notorio problema del SQL dinámico. Imagina lo que sucedería si volvemos a nuestro ejemplo del *"if"*, pero esta vez, hacemos que "ACTIVE = 1" sea también una condición dinámica.

```
<select id="findActiveBlogLike"
      parameterType="Blog" resultType="Blog">
  SELECT * FROM BLOG
  WHERE
  <if test="state != null">
    state = #{state}
  </if>
  <if test="title != null">
    AND title like #{title}
  </if>
  <if test="author != null and author.name != null">
    AND author_name like #{author.name}
  </if>
</select>
```

¿Qué sucede si no se cumple ninguna condición? Acabarías con una sentencia SQL con este aspecto:

```
SELECT * FROM BLOG
WHERE
```

Y eso fallará. ¿Y qué sucede si se cumple la segunda condición? Acabarías con una sentencia SQL con este aspecto:

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

Y eso también fallará. Este problema no se resuelve fácil con condicionales, y si alguna vez tienes que hacerlo, posiblemente no quieras repetirlo nunca más.

MyBatis tiene una respuesta sencilla que funcionará en el 90% de los casos. Y en los casos en los que no funciona puedes personalizarlo para hacerlo funcionar. Con un cambio simple, todo funciona correctamente:

```
<select id="findActiveBlogLike"
      parameterType="Blog" resultType="Blog">
  SELECT * FROM BLOG
  <where>
      <if test="state != null">
            state = #{state}
      </if>
      <if test="title != null">
            AND title like #{title}
      </if>
      <if test="author != null and author.name != null">
            AND author_name like #{author.name}
      </if>
  </where>
</select>
```

El elemento *where* sabe que debe insertar la "WHERE" solo si los tags internos devuelven algún contenido. Más aun, si el contenido comienza con "AND" o "OR", sabe cómo eliminarlo.

Si el elemento *where* no se comporta exactamente como te gustaría, lo puedes personalizar definiendo tu propio elemento *trim.* Por ejemplo, el *trim* equivalente al elemento *where* es:

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

El atributo *overrides* acepta una lista de textos delimitados pro el carácter "| " donde el espacio en blanco es relevante. El resultado es que se elimina cualquier cosa que se haya especificado en el atributo *overrides,* y que se inserta todo lo incluido en el atributo *with.*

Hay una solución similar para updates dinámicos llamada *set.* El elemento *set* se pude usar para incluir dinámicamente columnas para modificar y dejar fuera las demás. Por ejemplo:

```
<update id="updateAuthorIfNecessary"
          parameterType="domain.blog.Author">
    update Author
      <set>
        <if test="username != null">username=#{username},</if>
        <if test="password != null">password=#{password},</if>
        <if test="email != null">email=#{email},</if>
        <if test="bio != null">bio=#{bio}</if>
```

```
    </set>
  where id=#{id}
</update>
```

En este caso, el elemento *set* prefijará dinámicamente el valor SET y además eliminará todas las comas sobrantes que pudieran quedar tras las asignaciones de valor después de que se hayan aplicado las condiciones.

Si tienes curiosidad de qué aspecto tendría el elemento *trim* equivalente, aquí lo tienes:

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

Fíjate que en este caso estamos sobrescribiendo un sufijo y añadiendo un prefijo.

foreach

Otra necesidad común del SQL dinámico es iterar sobre una colección, habitualmente para construir una condición IN. Por ejemplo:

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

El elemento *foreach* es muy potente, permite especificar una colección y declarar variables elemento e índice que pueden usarse dentro del cuerpo del elemento. Permite también abrir y cerrar strings y añadir un separador entre las iteraciones. Este elemento es inteligente en tanto en cuanto no añade separadores extra accidentalmente.

- ⇨ Nota: Puedes pasar como objeto de parámetro un List o un Array a MyBatis. Cuando haces esto, MyBatis lo envuelve automáticamente en un Map usando su nombre como clave. Las instancias de lista usarán “list” como clave y las instancias array usarán “array”.

Esto finaliza la discusión sobre la configuración XML y los ficheros de mapeo XML. En la sección siguiente hablaremos del API Java en detalle, de forma que puedas obtener el máximo rendimiento de los mapeos que has creado.

Java API

Ahora que ya conoces cómo configurar MyBatis y crear mapeos estás listo para lo mejor. El API Java es donde obtendrás los mejores frutos de tus esfuerzos. Como verás, comparado con JDBC, MyBatis simplifica enormemente tu código y lo mantiene limpio, de fácil comprensión y mantenimiento. MyBatis 3 incorpora muchas mejoras significativas para hacer que el trabajo con SQL Maps sea aun mejor.

Estructura de directorios

Antes de zambullirnos en el propio API Java , es importante comprender las mejores prácticas relativas a la estructura de directorios. MyBatis es muy flexible, y puedes hacer casi cualquier cosa con tus ficheros. Pero como en cualquier otro framework, hay una forma recomendable.

Veamos una estructura de directorios típica:

```
/my_application
      /bin
      /devlib
      /lib                                    ← Los ficheros .jar de
      /src                                      MyBatis van aqui.
            /org/myapp/
                  /action
                  /data                       ← Los artefactos de
                        /SqlMapConfig.xml       MyBatis van aqui, lo que
                        /BlogMapper.java        incluye, mappers,
                        /BlogMapper.xml         configuración XML, y
                  /model                        ficheros de mapeo XML.
                  /service
                  /view
            /properties                       ← Las Properties incluidas
      /test                                     en tu configuración XML van
            /org/myapp/                         aqui.
                  /action
                  /data
                  /model
                  /service
                  /view
            /properties                       Recuerda, estas son
      /web                                    recomendaciones, no
            /WEB-INF                          requisitos, pero otros te
                  /web.xml                    agradecerán que utilices
                                              una estructura de
                                              directorio común.
```

Los ejemplos restantes en esta sección asumen que estás utilizando esta estructura de directorios.

SqlSessions

El interfaz principal para trabajar con MyBatis es el SqlSession. A través de este interfaz puedes ejecutar comandos, obtener mappers y gestionar transacciones. Hablaremos más sobre el propio SqlSession en breve, pero primero veamos cómo obtener una instancia de SqlSession. Las SqlSessions se crean por una instancia de SqlSessionFactory. La SqlSessionFactory contiene métodos para crear instancias de SqlSessions de distintas formas. La SqlSessionFactory en si misma se crea por la SqlSessionFactoryBuilder que puede crear una SqlSessionFactory a partir de XML, anotaciones o un objeto Configuration creado por código.

SqlSessionFactoryBuilder

El SqlSessionFactoryBuilder tiene cinco métodos build(), cada cual permite construir una SqlSession desde un origen distinto.

```
SqlSessionFactory build(InputStream is)
SqlSessionFactory build(InputStream is, String environment)
SqlSessionFactory build(InputStream is, Properties properties)
SqlSessionFactory build(InputStream is, String env, Properties props)
SqlSessionFactory build(Configuration config)
```

Los primeros cuatro métodos son los más comunes, dado que reciben una instancia de InputStream que referencia a un documento XML, o más específicamente, al fichero SqlMapConfig.xml comentado anteriormente. Los parámetros opcionales son *environment* y *properties.* Environment determina qué entorno cargar, incluyendo el datasource y el gestor de transacciones. Por ejemplo:

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

Si llamas al método build que recibe el parámetro *environment,* entonces MyBatis usará la configuración de dicho entorno. Por supuesto, si especificas un entorno inválido, recibirás un error. Si llamas a alguno de los métodos que no reciben el parámetro *environment*, entonces se utilizará el entorno por defecto (que es el especificado como default=”development” en el ejemplo anterior).

Si llamas a un método que recibe una instancia de properties, MyBatis cargará dichas properties y las hará accesibles desde tu configuración. Estas propiedades pueden usarse en lugar de la gran mayoría de los valores utilizando al sintaxis: ${propName}

Recuerda que las propiedades pueden también referenciarse desde el fichero SqlMapConfig.xml, o especificarse directamente en él. Por lo tanto es importante conocer las prioridades. Lo mencionamos anteriormente en este documento, pero lo volvemos a mencionar para facilitar la referencia.

Si una propiedad existe en más de un lugar, MyBatis la carga en el siguiente orden:

- Las propiedades especificadas en el cuerpo del elemento properties se cargan al principio.
- Las propiedades cargadas desde los atributos resource/url del elemento properties se leen a continuación, y sobrescriben cualquier propiedad duplicada que hubiera sido cargada anteriormente.
- Las propiedades pasadas como argumento se leen al final, y sobrescriben cualquier propiedad duplicada que hubiera sido cargada anteriormente.

Por lo tanto la prioridad mayor es la de las propiedades pasadas como parámetro, seguidas por las especificadas en el atributo resource/url y finalmente las propiedades especificadas en el cuerpo del elemento properties.

Por tanto, para resumir, los primeros cuatro métodos son casi iguales pero te permiten opcionalmente especificar el environment y/o las propiedades. Aquí hay un ejemplo de cómo se construye un SqlSessionFactory desde un fichero SqlMapConfig.xml.

```
String resource = "org/mybatis/builder/MapperConfig.xml";
InputStream is = Resources.getResourceAsStream(resource);
SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
SqlSessionFactory factory = builder.build(is);
```

Observa que estamos usando la clase de utilidad *Resources*, que está ubicada en el paquete org.mybatis.io. La clase Resources, tal y como su nombre indica, te ayuda a cargar recursos desde el classpath, el sistema de ficheros o desde una web o URL. Con un vistazo rápido al código fuente en tu IDE descubrirás un conjunto bastante obvio de métodos. Rápidamente:

```
URL getResourceURL(String resource)
URL getResourceURL(ClassLoader loader, String resource)
InputStream getResourceAsStream(String resource)
InputStream getResourceAsStream(ClassLoader loader, String resource)
Properties getResourceAsProperties(String resource)
Properties getResourceAsProperties(ClassLoader loader, String resource)
```

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

El último método build() recibe una instancia de Configuration. La clase Configuration contiene todo lo que posiblemente necesites conocer de la instancia de SqlSessionFactory. La clase Configuración es útil para investigar la configuración, incluido añadir o modificar SQL maps (no es recomendable una vez la aplicación ha comenzado a aceptar peticiones). La clase Configuration tiene todas las opciones de configuración que hemos visto ya pero expuestas como una API Java. A continuación se muestra un ejemplo simple de cómo instanciar manualmente un objeto Configuration y pasarlo al método build() para crear un SqlSessionFactory.

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

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

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

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

Ahora tienes un SqlSessionFactory, que puede utilizarse para crear interfaces SqlSession.

SqlSessionFactory

SqlSessionFactory tiene seis métodos que se usan para crear instancias de SqlSession. En general, las decisiones que deberás tomar cuando tengas qué elegir de entre alguno de estos métodos son:

- **Transacción:** ¿Quieres usar un ámbito transaccional para esta sesión o utilizar auto-commit (lo cual equivale a no usar transacción en la mayoría de las bases de datos y/o JDBC drivers)?
- **Conexión:** ¿Quieres que MyBatis obtenga una conexión de un datasource o quieres proporcionar tu propia conexión?
- **Execution:** ¿Quieres que MyBatis reúse PreparedStatements y/o haga batch updates (incluyendo inserts y deletes)?

El conjunto de métodos sobrecargados openSession te permiten seleccionar una combinación de estas opciones que tenga sentido.

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

El método openSession() por defecto que recibe parámetros crea una SqlSession con las siguientes características.

- Se arranca una transacción (NO auto-commit)
- Se obtiene una conexión de una instancia de DataSource configurada en el environment activo.
- El nivel de aislamiento transaccional será el que la base de datos tenga establecido por defecto.
- No se reusaran PreparedStatements y no se realizarán actualizaciones batch.

La mayoría de los métodos son auto explicativos. Para habilitar el auto-commit, pasa el valor “true” al parámetro opcional *autoCommit.* Para proporcionar tu propia conexión pasa una instancia de conexión al parámetro conexión. Ten en cuenta que no hay método para proporcionar tanto la conexión como el auto-commit porque MyBatis utilizará las opciones que esté usando actualmente la conexión suministrada. MyBatis usa una enumeration para indicar los niveles de aislamiento denominado TransactionIsolationLevel, pero funcionan como se espera de ellos y tiene los 5 niveles soportados por JDBC (NONE, READ_UNCOMMITTED, READ_COMMITTED, REPEATABLE_READ, SERIALIZABLE).

El único parámetro que puede ser nuevo para ti es el ExecutorType. Esta enumeración define tres valores:

ExecutorType.SIMPLE
Este tipo de executor no hace nada en especial. Crea un PreparedStatement para cada sentencia a ejecutar.

ExecutorType.REUSE

Este tipo de executor reusará PreparedStatements.

ExecutorType.BATCH

Este executor hará batch de todos las sentencias de actualización.

➔ Nota: Hay un método más del SqlSessionFactory que no hemos mencionado y es *getConfiguration()*. Este método devuelve una instancia de Configuration que puedes usar para introspeccionar la configuración de MyBatis en tiempo de ejecución.

➔ Nota: Si has usado una versión anterior de MyBatis recordarás que las sesiones, transacciones y batches eran cosas separadas. Esto ya no es así, todas ellas están contenidas en el interfaz SqlSession. No tienes que gestionar las transacciones o los batches de forma separada para obtener todo su potencial.

SqlSession

Como hemos comentado anteriormente, la instancia de SqlSession es la clase más potente de MyBatis. Es donde encontrarás todos los métodos para ejecutar sentencias, hacer commit o rollback de transacciones y obtener mappers.

Hay más de veinte métodos en la clase SqlSession, así que vayamos dividiéndolos en grupo fáciles de digerir.

Métodos de ejecución de sentencias

Estos métodos se usan para ejecutar las sentencias SELECT, INSERT, UPDATE y DELETE que se hayan definido en los ficheros xml de mapeo SQL. Son bastante auto explicativos, cada uno recibe el ID del statement y el objeto de parámetro, que puede ser una primitiva, un JavaBean, un POJO o un Map.

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

La diferencia entre selectOne y selectList es que selectOne debe devolver sólo un objeto. Si hay más de uno se lanzará una excepción. Si no hay ninguno se devolverá null. Si no sabes cuantos objetos esperas recibir, usa selectList. Si quieres comprobar la existencia de un objeto sería mejor que devuelvas un count(). SelectMap es un caso especial diseñado para convertir una lista de resultados en un Map basado en las propiedades de los objetos recibidos. Como no todas las sentencias requieren un parámetro, estos métodos han sido sobrecargados de forma que se proporcionan versiones que no reciben el parámetro objeto.

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

Finalmente hay tres versiones avanzadas de los métodos select que te permiten restringir el rango de filas devueltas, o proporcionar lógica de tratamiento de resultados personalizada, normalmente para grandes cantidades de datos.

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

El parámetro RowBounds hace que MyBatis salte los registros especificados y que limite los resultados devueltos a cierto número. La clase RowBounds tiene un constructor que recibe ambos el offset y el limit, y es inmutable.

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

El rendimiento de algunos drivers puede variar mucho en este aspecto. Para un rendimiento optimo, usa tipos de ResultSet SCROLL_SENSITIVE o SCROLL_INSENSITIVE (es decir, no FORWARD_ONLY).

El parámetro ResultHandler te permite manejar cada fila como tú quieras. Puedes añadirla a una lista, crear un Map, un Set, o descartar cada resultado y guardar solo cálculos. Puedes hacer casi todo lo que quieras con un ResultHandler, de hecho, es lo que MyBatis usa internamente para construir listas de ResultSets.

La interfaz es muy sencilla:

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

El parámetro ResultContext te da acceso al objeto resultado en sí mismo, un contador del número de objetos creados y un método booleano stop() que te permite indicar a MyBatis que pare la carga de datos.

Métodos de control de Transacción

Hay cuatro métodos para controlar el ámbito transaccional. Por supuesto, no tienen efecto alguno si has elegido usar auto-commit o si estás usando un gestor de transacciones externo. Sin embargo, si has elegido el JDBCTransactionManager entonces los cuatro métodos que te resultarán útiles son:

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

Por defecto MyBatis no hace un commit a no ser que haya detectado que la base de datos ha sido modificada por una insert, update o delete. Si has realizado cambios sin llamar a estos métodos, entonces puedes pasar true en al método de commit() y rollback() para asegurar que se realiza el commit (ten en cuenta que aun así no puedes forzar el commit() en modo auto-commit o cuando se usa un gestor de transacciones externo). La mayoría de las veces no tendrás que llamar a rollback() dado que MyBatis lo hará por ti en caso de que no hayas llamado a commit(). Sin embargo, si necesitas un control más fino sobre la sesión, donde puede que haya varios commits, tienes esta opción para hacerlo posible.

Borrar la caché de sesión

```
void clearCache()
```

Cada instancia de SqlSession tiene una caché local que se borra en cada update, commit, rollback y close. Para limpiarla explícitamente (quizá porque tienes la intención de hacer más cosas), puedes llamar a clearCache().

Asegurarse de que la SqlSession se cierra

```
void close()
```

El punto más importante del que debes asegurarte es que cierras todas las sesiones que abres. La mejor forma de asegurarse es usar el patrón mostrado a continuación:

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

➔ Nota: Al igual que con SqlSessionFactory, puedes obtener la instancia de Configuration que está usando al SqlSession llamando al método getConfiguration().

```
Configuration getConfiguration()
```

Uso de mappers

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

Aunque los métodos insert, update, delete y select son potentes, también son muy verbosos, no hay seguridad de tipos (type safety) y no son tan apropiados para tu IDE o tus pruebas unitarias como pudieran ser. Ya hemos visto un ejemplo de uso de mappers en la sección de primeros pasos.

Por lo tanto, una forma más común de ejecutar mapped statements es utilizar clases Mapper. Un mapper es simplemente una interfaz con definiciones de métodos que se hacen encajar con métodos de SqlSession. El ejemplo siguiente demuestra algunas firmas de método y como se asignan a una SqlSession.

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


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

En resumen, cada firma de método de mapper se asigna al método de la SqlSession al que está asociado pero sin parámetro ID. En su lugar el nombre del método debe ser el mismo que el ID del mapped statement.

Además, el tipo devuelto debe ser igual que el result type del mapped statement. Todos los tipos habituales se soportan, incluyendo primitivas, mapas, POJOs y JavaBeans.

➔ Los mappers no necesitan implementar ninguna interfaz o extender ninguna clase. Sólo es necesario que la firma de método pueda usarse para identificar unívocamente el mapped statement correspondiente.

➔ Los mappers pueden extender otras interfaces. Asegúrate que tienes tus statements en los namespaces adecuados en tu fichero XML. Además, la única limitación es que no puedes tener el mismo método, con la misma firma, en dos interfaces de la jerarquía (una mala idea en cualquier caso).

Puedes pasar más de un parámetro a un método de un mapper. Si lo haces, se usará como nombre su posición en la lista de parámetros, por ejemplo: #{1}, #{2} etc. Si quieres cambiar su nombre (solo en caso de parámetros múltiples) puedes usar la notación @Param("paramName").

También puedes pasar una instancia de RowBounds al método para limitar los resultados.

Anotaciones de Mappers

Desde sus comienzos, MyBatis ha sido siempre un framework XML. La configuración se basa en XML y los mapped statements se definen en XML. Con MyBatis 3, hay más opciones. MyBatis 3 se ha desarrollado sobre una exhaustiva y potente API de configuración Java. Este API es el fundamento de la configuración basada en XML y también de la nueva configuración basada en anotaciones. Las anotaciones simplemente ofrecen una forma más sencilla de implementar los mapped statements sin introducir un montón de sobrecarga.

➔ Nota: Las anotaciones Java son desafortunadamente muy limitadas en su flexibilidad y expresividad. A pesar de haber dedicado mucho tiempo a la investigación, diseño y pruebas, los mapeos más potentes de MyBatis simplemente no es posible construirlos con anotaciones. Los atributos C# (por ejemplo) no sufren de las mismas limitaciones y por tanto MyBatis.NET podrá construir una alternativa mucho más rica al XML. Dicho esto, la configuración basada en anotaciones Java también tiene sus ventajas.

Las anotaciones son las siguientes:

Anotación	Target	XML Equivalente	Descripción
@CacheNamespace	Class	<cache>	Configura la cache para un namespace (una clase). Atributos: **implementation**, **eviction**, **flushInterval**, **size** and **readWrite**.
@CacheNamespaceRef	Class	<cacheRef>	Referencia una cache de otro namespace. Atributos: **value**, que debe ser el nombre del namespace (ej. un nombre de clase completamente cualificado).
@ConstructorArgs	Method	<constructor>	Agrupa un conjunto de resultados que serán pasados al constructor de un objeto de resultado. Atributos: **value**, que es un array de Args.
@Arg	Method	<arg> <idArg>	Un argumento que es parte de un ConstructorArgs. Atributos: **id**, **column**, **javaType**, **jdbcType**, **typeHandler, select and resultMap**. El atributo id es un valor booleano que identifica la propiedad que será usada en las comparaciones, parecido al elemento XML <idArg>.
@TypeDiscriminator	Method	<discriminator>	Un grupo de clases que se pueden usar para determinar que mapeo de resultados realizar. Atributos: **column, javaType, jdbcType, typeHandler, cases.** El atributo cases es un array de Cases.

Anotación	Target	XML Equivalente	Descripción
@Case	Method	<case>	Un caso concreto y su mapeo correspondiente. Atributos: **value, type, results.** El atributo results es un array de Results, por tanto esta anotación Case es similar a un ResultMap, que se especifica mediante la anotación Results a continuación.
@Results	Method	<resultMap>	Una lista de Result mapping que contiene los detalles de cómo una columna particular se mapea a una propiedad o campo. Atributos: **value**, que es un array de anotaciones Result.
@Result	Method	<result> <id>	Un result mapping entre una columna y una propiedad o campo. Atributos: : **id, column, property, javaType, jdbcType, typeHandler, one, many.** El atributo id es un valor booleano que indica que la propiedad debe usarse en comparaciones (similar al <id> de los mapeos XML). El atributo one sirve para asociaciones de simples, similar al <association>, y el atributo many es para colecciones, similar al <collection>. Sus denominaciones son tales para evitar conflictos con nombres de clases.
@One	Method	<association>	Un mapeo a una propiedad que contiene un tipo complejo. Atributos: **select**, que contiene el nombre completamente cualificado de un mapped statement (o un método de mapper) que puede cargar la instancia del tipo indicado. ***Nota:*** *Habrás visto que el mapeo de tipo join no se soporta mediante el API de anotaciones. Esto es debido a las limitaciones de las anotaciones en Java que no permiten referencias circulares.*
@Many	Method	<collection>	Un mapeo a una propiedad que contiene una colección de tipos complejos. Atributos: **select**, que contiene el nombre completamente cualificado de un mapped statement (o un método de mapper) que puede cargar la instancia del tipo indicado. ***Nota:*** *Habrás visto que el mapeo de tipo join no se soporta mediante el API de anotaciones. Esto es debido a las limitaciones de las anotaciones en Java que no permiten referencias circulares.*

Anotación	Target	XML Equivalente	Descripción
@MapKey	Method		Se usa en métodos cuyo tipo de retorno es Map. Se usa para convertir una Lista de objetos de resultado a un Map basándose en una propiedad de dichos objetos.
@Options	Method	Atributos de mapped statements.	Esta anotación proporciona acceso a un gran conjunto de opciones de configuración que normalmente aparecen como atributos en los mapped statements. En lugar de complicar cada anotación existente la anotación Options proporciona una forma sencilla y concisa de acceder a estas opciones. Atributos: **useCache=true, flushCache=false, resultSetType=FORWARD_ONLY, statementType=PREPARED, fetchSize=-1, timeout=-1, useGeneratedKeys=false, keyProperty="id", keyColumn="".** Es importante comprender que las anotaciones en Java no permiten indicar un valor nulo. Por lo tanto, cuando usas la anotación Options el statement usará todos los valores por defecto. Presta atención a estos valores pro defecto para evitar comportamientos inesperados. La keyColumn solo se requiere para algunas bases de datos (como PostgreSQL) cuando la columna no es la primera columna de la tabla.
@Insert **@Update** **@Delete** **@Select**	Method	<insert> <update> <delete> <select>	Cada una de estas anotaciones representa el SQL que debe ejecutarse. Cada una de ellas recibe un array de strings (o un solo string). Si se pasa un array de strings, todos ellos se concatenarán introduciendo un espacio en blanco entre ellos. Esto ayuda a evitar el problema "falta un espacio en blanco" al construir SQL en código Java. Sin embargo, también puedes concatenarlos en un solo string si lo prefieres. Atributos: **value**, que es el array de String para formar una única sentencia SQL.

Anotación	Target	XML Equivalente	Descripción
@InsertProvider **@UpdateProvider** **@DeleteProvider** **@SelectProvider**	Method	<insert> <update> <delete> <select> Permite la creación de SQL dinámico.	Estas anotaciones SQL alternativas te permiten especificar un nombre de clases y un método que devolverán la SQL que debe ejecutarse. Cuando se ejecute el método MyBatis instanciará la clase y ejecutará el método especificados en el provider. El método puede opcionalmente recibir el objeto parámetro como su único parámetro o no recibir ningún parámetro. Atributos: **type, method**. El atributo type es el nombre completamente cualificado de una clase. El method es el nombre un método de dicha clase. ***Nota***: A continuación hay una sección sobre la clase SelectBuilder, que puede ayudar a construir SQL dinámico de una forma más clara y sencilla de leer.
@Param	Parameter	N/A	Si tu mapper recibe parámetros múltiples, esta anotación puede aplicarse a cada parámetro para proporcionarle un nombre. En caso contrario, los parámetros múltiples se nombrarán según su posición ordinal (sin incluir el parámetro RowBounds). Por ejemplo: #{1}, #{2} etc. es el defecto. Con @Param("persona"), el parámetro se llamará #{persona}.
@SelectKey	Method	<selectKey>	Esta anotación es igual que la <selectKey> para métodos anotados con @Insert o @InsertProvider. Se ignora en otros métodos. Si especificas la anotación @SelectKey, entonces MyBatis ignorará todas las propiedades de generación de claves proporcionadas por la anotación @Options, o mediante propiedades de configuración. Atributos: **statement** un array de strings que contienen la SQL a ejecutar, **keyProperty** que es la propiedad del objeto parámetro que se actualizará con el Nuevo valor, **before** que debe valer true o false para indicar si la sentencia SQL debe ejecutarse antes o después de la insert, **resultType** que es el tipo de la propiedad keyProperty, y **statementType**=PREPARED.

Anotación	Target	XML Equivalente	Descripción
@ResultMap	Method	N/A	Esta anotación se usa para proporcionar el id de un elemento <resultMap> en un mapper XML a una anotación @Select o @SelectProvider. Esto permite a las selects anotadas reusar resultmaps definidas en XML. Esta anotación sobrescribe las anotaciones @Result o @ConstructorArgs en caso de que se hayan especificado en la select anotada.

Ejemplos de mappers anotados

Este ejemplo muestra como se usa la anotación @SelectKey para obtener un valor de una secuencia antes de de una insert:

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

Este ejemplo muestra como se usa la anotación @SelectKey para obtener el valor de una identity después de una insert:

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

SelectBuilder

Una de las cosas más tediosas que un programador Java puede llegar a tener que hacer es incluir código SQL en código Java. Normalmente esto se hace cuando es necesario generar dinámicamente el SQL – de otra forma podrías externalizar el código en un fichero o un procedimiento almacenado. Como ya has visto, MyBatis tiene una respuesta potente a la generación dinámica de SQL mediante las capacidades del mapeo XML. Sin embargo, en ocasiones se hace necesario construir una sentencia SQL dentro del código Java. En este caso, MyBatis tiene una funcionalidad más para ayudarte en ello, antes de que comiences con el típico lío de signos de suma, comillas, líneas nuevas, problemas de formato y condicionales anidados para tratar con las comas extra y las conjunciones AND… Realmente, generar código dinámico en java, puede ser una verdadera pesadilla.

MyBatis 3 introduce un concepto un tanto distinto para tratar con el problema. Podríamos haber creado simplemente una clase que te permitiera invocar a métodos para crear una sentencia SQL paso a paso. En su lugar hemos intentado proporcionar algo distinto. El resultado está más próximo a un Domain Specific Languaje de lo que java jamás estará en su estado actual….

Los secretos de SelectBuilder

La clase SelectBuilder no es mágica, y tampoco pasa nada si no sabes cómo funciona internamente. Así que veamos sin más preámbulo qué es lo que hace. SelectBuilder usa una combinación de imports estáticos y una variable ThreadLocal para permitir una sintaxis más limpia que permite entrelazar condicionales y se encarga del formateo de SQL por ti. Te permite crear métodos como este:

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

Este es un ejemplo bastante sencillo que posiblemente habrías preferido construir en estático. Aquí hay uno más complicado:

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

Construir el SQL de arriba concatenando Strings sería complicado. Por ejemplo:

```
"SELECT P.ID, P.USERNAME, P.PASSWORD, P.FULL_NAME, "
"P.LAST_NAME,P.CREATED_ON, P.UPDATED_ON " +
"FROM PERSON P, ACCOUNT A " +
"INNER JOIN DEPARTMENT D on D.ID = P.DEPARTMENT_ID " +
"INNER JOIN COMPANY C on D.COMPANY_ID = C.ID " +
"WHERE (P.ID = A.ID AND P.FIRST_NAME like ?) " +
"OR (P.LAST_NAME like ?) " +
"GROUP BY P.ID " +
"HAVING (P.LAST_NAME like ?) " +
"OR (P.FIRST_NAME like ?) " +
"ORDER BY P.ID, P.FULL_NAME";
```

Si prefieres esa sintaxis, no hay problema en que la uses. Sin embargo, es bastante propensa a errores. Fíjate la cuidadosa adición de un espacio en blanco al final de cada línea. Aun así, incluso si prefieres esta sintaxis, el siguiente ejemplo es decididamente más sencillo que la concatenación de Strings:

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

¿Qué hay de especial en este ejemplo? Bien, si lo miras detenidamente, verás que no hay que preocuparse de duplicar “AND”s, o elegir entre “WHERE” o “AND”, o ninguno de ambos! La sentencia de arriba crea una “query by example” (query a partir de un ejemplo) para todos los registros de PERSON, algunos con un like por el parámetro ID, otros por el parámetro firstName o por lastName - o cualquier combinación de ellos. La SelectBuilder se encarga de saber cuándo es necesario añadir un “WHERE”, cuando debes añadirse un “AND” y también de la concatenación de Strings. Y lo mejor de todo es que lo hace casi de forma independiente de cómo hayas llamado los métodos (la única excepción es el método OR()).

Los dos métodos que puede que te hayan llamado la atención son: BEGIN() y SQL(). Simplificando, siempre debe comenzarse a usar el SelectBuilder llamado a BEGIN() y finalizar llamando a SQL(). Por supuesto puedes incluir métodos entre medio como lo requiera tu lógica pero el ámbito de la generación siempre comienza con un BEGIN() y acaba con un SQL(). El método BEGIN() limpia la variable ThreadLocal, para asegurar que no arrastras información de un estado anterior, y el método SQL() ensambla la sentencia SQL basándose en las llamadas que has ido haciendo desde la última llamada a BEGIN(). Debes saber que BEGIN() tiene un sinónimo llamado RESET() que hace exactamente lo mismo pero se lee mejor en ciertos contextos.

Para usar el SelectBuilder como en los ejemplos de arriba simplemente tienes que importarlo estáticamente de la siguiente forma:

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

Una vez que se ha importado, la clase en la que estés trabajando tendrá accesibles todos los métodos del SelectBuilder. La lista completa de métodos es la siguiente:

Método	Descripción

Método	Descripción
BEGIN() / RESET()	Estos métodos limpian estado guardad en el ThreadLocal de la clase SelectBuilder, y la preparan para construir una nueva sentencia. BEGIN() se lee mejor cuando se está creando una sentencia. RESET() se lee mejor cuando se está borrando lo hecho anteriormente en medio de una ejecución (quizá porque la lógica necesita una sentencia completamente distinta según las condiciones).
SELECT(String)	Comienza o añade a una sentencia SELECT. Se puede invocar más de una vez y los parámetros se irán añadiendo a la sentencia SELECT. Los parámetros son normalmente una lista de columnas o alias separados por comas, pero puede ser cualquier cosa que acepte el driver de base de datos.
SELECT_DISTINCT(String)	Comienza o añade a una sentencia SELECT, también añade la palabra clave "DISTINCT" a la sentencia generada. Se puede invocar más de una vez y los parámetros se irán añadiendo a la sentencia SELECT. Los parámetros son normalmente una lista de columnas o alias separados por comas, pero puede ser cualquier cosa que acepte el driver de base de datos.
FROM(String)	Comienza o añade a una cláusula FROM. Se puede invocar más de una vez y los parámetros se irán añadiendo a la clausula FROM. Los parámetros son normalmente un nombre de tabla o alias o cualquier cosa que acepte el driver de base de datos.
JOIN(String) INNER_JOIN(String) LEFT_OUTER_JOIN(String) RIGHT_OUTER_JOIN(String)	Añade una nueva clausula JOIN del tipo apropiado, dependiendo al método que se haya llamado. El parámetro puede incluir un join estándar que consiste en las columnas y las condiciones sobre las que hacer la join.
WHERE(String)	Añade una nueva condición a la clausula WHERE concatenada con un AND. Puede llamarse más de una vez, lo cual hará que se añadan más condiciones todas ellas concatenadas con un AND. O usa OR() para partirlas con un OR().
OR()	Parte las condiciones actuales de la WHERE con un OR. Puede llamarse más de una vez, pero llamarlas más de una vez en la misma línea puede producir sentencias incorrectas.
AND()	Parte las condiciones actuales de la WHERE con un AND. Puede llamarse más de una vez, pero llamarlas más de una vez en la misma línea puede producir sentencias incorrectas. Dado que WHERE y HAVING concatenan automáticamente el AND, es muy infrecuente que sea necesario invocar a este método y se incluye realmente por completitud.
GROUP_BY(String)	Añade una nueva clausula GROUP BY grupo, concatenada con una coma. Se le puede llamar más de una vez, lo cual hará que se concatenen nuevas condiciones separadas también por coma.
HAVING(String)	Añade una nueva clausula HAVING, concatenada con un AND. Se le puede llamar más de una vez, lo cual hará que se concatenen nuevas condiciones separadas también por AND. Usa OR() para dividirlas por OR.

Método	Descripción
ORDER_BY(String)	Añade un Nuevo elemento a la clausula ORDER BY concatenado por coma. Se le puede llamar más de una vez, lo cual hará que se concatenen nuevas condiciones separadas también por coma.
SQL()	Devuelve la SQL generada y restablece el estado del SelectBuilder (como si se hubiera llamado a un BEGIN() o a un RESET()). Por tanto este método solo se puede llamar una vez!

SqlBuilder

De forma similar a la SelectBuilder, MyBatis también proporciona una SqlBuilder generalizada. Incluye los métodos de SelectBuilder y también métodos para crear inserts, updates y deletes. Esta clase puede ser de utilidad para usarla en la creación de SQLs en DeleteProvider, InsertProvider o UpdateProvider (y también la SelectProvider)

Para u

Para usar el SqlBuilder como en los ejemplos de arriba simplemente tienes que importarlo estáticamente de la siguiente forma:

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

SqlBuilder contiene todos los métodos de SelectBuilder, y estos métodos adicionales:

Method	Descripción
DELETE_FROM(String)	Comienza una sentencia delete y especifica la tabla donde borrar. Generalmente suele ir seguida de una clausula WHERE!
INSERT_INTO(String)	Comienza una sentencia insert y especifica al tabla en la que insertar. Suele ir seguida de una o más llamadas a VALUES().
SET(String)	Añade a la lista “set” de una update.
UPDATE(String)	Comienza una sentencia update y especifica la tabla que modificar Suele ir seguida de una o más llamadas a SET() y normalmente de una llamada a WHERE().
VALUES(String, String)	Añade a una sentencia insert. El primer parámetro es el nombre de columna y el Segundo el valor(es).

Aquí hay algunos ejemplos:

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

public String insertPersonSql() {
    BEGIN(); // Clears ThreadLocal variable
```

```
        INSERT_INTO("PERSON");
        VALUES("ID, FIRST_NAME", "${id}, ${firstName}");
        VALUES("LAST_NAME", "${lastName}");
        return SQL();
    }

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

Logging

MyBatis proporciona información de logging mediante el uso interno de una factoría. La factoría interna delega la información de logging en alguna de las siguientes implementaciones.

- SLF4J
- Jakarta Commons Logging (JCL – NO Job Control Language!)
- Log4J
- JDK logging

La solución de logging elegida se basa en una introspección en tiempo de ejecución de la propia factoría interna de MyBatis. La factoría usará la primera implementación de logging que encuentre (el orden de búsqueda es el de la lista de más arriba). Si no se encuentra ninguna, el logging se desactivará.

Muchos entornos vienen con JCL incluido como pare del classpath del servidor (por ejemplo Tomcat y WebSphere). Es importante conocer que en esos entorno, MyBatis usará JCL como implementación de logging. En un entorno como WebSphere esto significa que tu configuración de log4j será ignorada dado que WebSphere proporciona su propia implementación de JCL. Esto puede ser muy frustrante porque parece que MyBatis está ignorando tu configuración de logging (en realidad, MyBatis está ignorando tu configuración de log4j porque está usando JCL en dicho entorno). Si tu aplicación se ejecuta en un entorno que lleva JCL incluido pero quieres usar un método distinto de logging puedes llamar a los siguientes métodos:

```
org.apache.ibatis.logging.LogFactory.useSlf4jLogging();
org.apache.ibatis.logging.LogFactory.useLog4JLogging();
org.apache.ibatis.logging.LogFactory.useJdkLogging();
org.apache.ibatis.logging.LogFactory.useCommonsLogging();
org.apache.ibatis.logging.LogFactory.useStdOutLogging();
```

Si eliges llamar a alguno de estos métodos, deberías hacerlo antes de llamar a ningún otro método de MyBatis. Además, estos métodos solo establecerán la implementación de log indicada si dicha

implementación está disponible en el classpath. Por ejemplo, si intentas seleccionar log4j y log4j no está disponible en el classpath, MyBatis ignorará la petición y usará su algoritmo normal de descubrimiento de implementaciones de logging.

Los temas específicos de JCL, Log4j y el Java Logging API quedan fuera del alcance de este documento. Sin embargo la configuración de ejemplo que se proporciona más abajo te ayudará a comenzar. Si quieres conocer más sobre estos frameworks, puedes obtener más información en las siguientes ubicaciones:

- Jakarta Commons Logging: http://jakarta.apache.org/commons/logging/index.html
- Log4j: http://jakarta.apache.org/log4j/docs/index.html
- JDK Logging API: http://java.sun.com/j2se/1.4.1/docs/guide/util/logging/

Log Configuration

MyBatis loguea la mayoría de su actividad usando clases de logging que no están en paquetes de MyBatis. Para ver el log de las sentencias debes activar el log en el paquete java.sql – específicamente en las siguientes clases:

```
java.sql.Connection
java.sql.PreparedStatement
java.sql.Resultset
java.sql.Statement
```

Nuevamente, como hagas esto es dependiente de la implementación de logging que se esté usando. Mostraremos cómo hacerlo con Log4j. Configurar los servicios de logging es simplemente cuestión de añadir uno o varios ficheros de configuración (por ejemplo log4j.properties) y a veces un nuevo JAR (por ejemplo log4j.jar). El ejemplo siguiente configura todos los servicios de logging para que usen log4j como proveedor. Sólo son dos pasos:

Paso 1: Añade el fichero Log4J JAR
Dado que usamos Log4j, necesitaremos asegurarnos que el fichero JAR está disponible para nuestra aplicación. Para usar Log4j, necesitas añadir el fichero JAR al classpath de tu aplicación. Puedes descargar Log4j desde la URL indicada más arriba.

En aplicaciones Web o de empresa debes añadir tu fichero log4j.java a tu directorio WEB-INF/lib, y en una aplicación standalone simplemente añádela al parámetro –classpath de la JVM.

Paso 2: Configurar Log4J
Configurar Log4j es sencillo – debes crear un fichero log4j.properties que tiene el siguientes aspecto:

```
# Configuración global…
log4j.rootLogger=ERROR, stdout
# Configuración de MyBatis…
#log4j.logger.org.apache.ibatis=DEBUG
#log4j.logger.java.sql.Connection=DEBUG
```

```
#log4j.logger.java.sql.Statement=DEBUG
#log4j.logger.java.sql.PreparedStatement=DEBUG
#log4j.logger.java.sql.ResultSet=DEBUG
# Salida a consola…
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
```

El fichero anterior es la configuración mínima que hará que el logging sólo reporte errores. La línea 2 del fichero es la que indica que solo se reporten errores al appender stdout. Un appender es un componente que recolecta la salida (ej. consola, ej. fichero, base de datos, etc.). Para maximizar el nivel de reporting cambia la línea 2 de la siguiente forma:

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

Al cambiar la línea 2 como se ha indicado, Log4j reportará todos los eventos al apender 'stdout' (consola). Si quieres ajustar el logging a un nivel más fino, puedes configurar qué clase envía sus logs al sistema usando la sección "MyBatis logging configuration..." del fichero anterior (líneas comentadas desde la 4 a la 8). Por ejemplo si queremos loguear la actividad de los PreparedStatement a la consola a nivel DEBUG, podemos cambiar simplemente la línea 7 a lo siguiente (fíjate que ya no está comentada):

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

El resto de la configuración sirve para configurar los appenders, lo cual queda fuera del ámbito de este documento. Sin embargo, puedes encontrar más información en el site de Log4j (la url está más arriba). O, puedes simplemente experimentar para ver los efectos que consigues con las distintas opciones de configuración.