


Acceso a Base de Datos con Java



© JMA 2016. All rights reserved

INTRODUCCIÓN

© JMA 2016. All rights reserved

Introducción

- Cuando nos ponemos a desarrollar, en el código final lo que tenemos que hacer a veces es mucho.
- Deberemos desarrollar código para:
 - Lógica de la aplicación
 - Acceso a Base de Datos
 - EJB para modularizar las aplicaciones.
- Deberemos también programar en diferentes Lenguajes:
 - Java (mayoritariamente)
 - SQL
 - HTML, etc
- "La vida es corta, dedique menos tiempo a escribir código para la unión BBDD-Aplicación JAVA y más tiempo añadiendo nuevas características"

© JMA 2016. All rights reserved

Introducción a Hibernate

- Las bases de datos relacionales son indiscutiblemente el centro de la empresa moderna.
- Los datos de la empresa se basan en entidades que están almacenadas en ubicaciones de naturaleza relacional. (Base de Datos)
- Los actuales lenguajes de programación, como Java, ofrecen una visión intuitiva, orientada a objetos de las entidades de negocios a nivel de aplicación.
- Se han realizado mucho intentos para poder combinar ambas tecnologías (relacionales y orientados a objetos), o para reemplazar uno con el otro, pero la diferencia entre ambos es muy grande.

© JMA 2016. All rights reserved

Discrepancia del Paradigma

- Problemas de identidad
 - Objetos Java definen dos nociones diferentes de identidad:
 - Identidad de objeto o referencia (equivalente a la posición de memoria, comprobar con `un ==`).
 - La igualdad como determinado por la aplicación de los métodos `equals()`.
 - La identidad de una fila de base de datos se expresa como la clave primaria.
 - Ni `equals()` ni `==` es equivalente a la clave principal.
- Problemas de Asociaciones
 - El lenguaje Java representa a las asociaciones mediante utilizar referencias a objetos
 - Las asociaciones entre objetos son punteros unidireccionales.
 - Relación de pertenencia → Elementos contenidos
 - Modelo de composición (colecciones) → Modelo jerárquico
 - Las asociaciones en BBDD están representados mediante la migración de claves.
 - Todas las asociaciones en una base de datos relacional son bidireccional

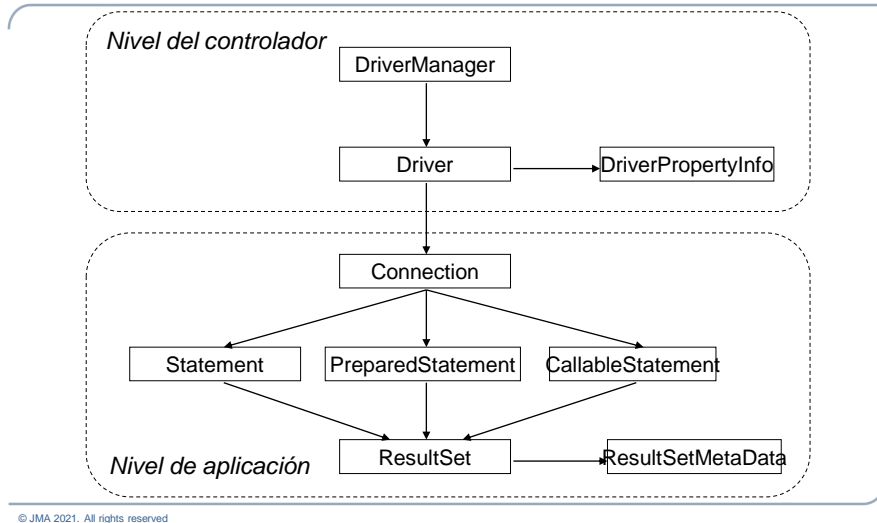
© JMA 2016. All rights reserved

Paquete `java.sql`

JDBC (JAVA DATABASE CONNECTIVITY)

© JMA 2021. All rights reserved

Arquitectura de JDBC



Esquema de JDBC

El trabajo con datos de una base de datos sigue siempre los mismos pasos.

Pasos a seguir:

- Cargar el controlador.
- Establecer una conexión.
- Operaciones con datos:
 - Preparar la sentencia.
 - Opcionalmente: pasar parámetros.
 - Ejecutar la sentencia.
 - Si recupera datos:
 - Recorrer el conjunto de resultados.
 - Realizar su tratamiento.
 - Opcionalmente: confirmar o deshacer los cambios.
- Opcionalmente: confirmar o deshacer los cambios.
- Cerrar la conexión.

Carga del controlador

La primera operación que debemos realizar para disponer de acceso a la base de datos es cargar la clase del controlador con el método:

```
Class.forName("<NombreClase>");
```

La clase cargada se encarga de registrarse de forma automática en el DriverManager utilizando el método:

```
DriverManager.registerDriver(this);
```

La clase puente JDBC-ODBC es:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

© JMA 2021. All rights reserved

Tipos de controladores

Hay cuatro tipos de controladores definidos por JDBC:

- Clase 1: Puente JDBC-ODBC
 - La clase del controlador utiliza las funciones nativas estandarizadas de ODBC. El cliente debe disponer de ODBC y conexión a la base de datos.
- Clase 2: JDBC a API nativa
 - La clase del controlador utiliza las funciones nativas suministradas por el fabricante del SGDB.
- Clase 3: Protocolo de red, todo Java
 - La clase del controlador se implementa en Java y se comunica en red con un servidor de aplicaciones mediante un protocolo de red estándar, que a su vez es el encargado de comunicarse con el servidor de base de datos. Es el más flexible.
- Clase 4: Protocolo nativa, todo Java
 - La clase del controlador se implementa en Java y se comunica directamente con el SGDB a través de un protocolo nativo de servidor.

© JMA 2021. All rights reserved

Tipos de Drivers / Controladores

Driver JDBC-ODBC:

- `sun.jdbc.odbc.JdbcOdbcDriver` (incluido en la API de JDBC de la plataforma J2SE)

Driver MySQL:

- `com.mysql.jdbc.Driver` (no incluido)

Driver Oracle:

- `oracle.jdbc.driver.OracleDriver` (no incluido)

Driver PostgreSQL

- `org.postgresql.Driver`

Driver DB2:

- `com.ibm.db2.jdbc.app.DB2Driver` (no incluido)

© JMA 2021. All rights reserved

Conexión

Para realizar una conexión invocamos el método:

```
Connection con =
DriverManager.getConnection("<CadenaConexión>", "<Usuario>",
"<Clave>");
```

Donde:

- CadenaConexión: Es la cadena de conexión a la base de datos siguiendo el formato establecido por el fabricante. Por regla general siguen el siguiente formato:
 - `jdbc:<TipoDeControlador>:<BaseDeDatos>`
- Usuario: Cadena con el nombre del usuario registrado en el gestor de la base de datos.
- Clave: Cadena con la clave que autentica al usuario.

© JMA 2021. All rights reserved

Conexión

Para realizar la Conexión necesitaremos:

- URL de la base de datos o dirección IP
- Nombre de Usuario
- Contraseña de autenticación en la Base de Datos
- Puerto (oracle)
- Nombre de la Base de Datos (oracle)

ORACLE

```
String cadena="jdbc:oracle:thin:System/oracle@Localhost:1521:xe"
Connection conexion= DriverManager.getConnection(cadena);
```

ODBC

```
cadena = "jdbc:odbc:Tutorial";
Connection conexion= DriverManager.getConnection(cadena, " ", " ");
```

MYSQL

```
cadena = "jdbc:mysql://localhost:3306/sakila";
Connection con=DriverManager.getConnection(cadena, "root", "mimono");
```

© JMA 2021. All rights reserved

Conexión

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
```

```
public final class JDBCbridged {
```

```
    static {
        try {
            Class.forName("oracle.jdbc.OracleDriver");
        } catch (ClassNotFoundException e) {
            System.err.println("No encuentro el driver");
        }
    }
}
```

```
    public static Connection getConnection() throws SQLException {
        return DriverManager.getConnection(
            "jdbc:oracle:thin:@localhost:1521:xe", "usuario", "contraseña");
    }
}
```

© JMA 2021. All rights reserved

Conexión: Métodos

<code>close();</code>	Se encarga de cerrar la conexión con la Base de Datos.
<code>setAutocommit(false/true);</code>	Activa o desactiva el método autocommit. Después de cada transacción se realiza un commit.
<code>commit();</code>	Todos los cambios realizados desde el último commit se vuelven permanentes.
<code>rollback();</code>	Desecha todos los cambios realizados desde el último commit o roolback
<code>createStatement()</code>	Crea un objeto Statement que ejecutará sentencias SQL.
<code>prepareStatement(String)</code>	Crea un objeto PreparedStatement, para realizar llamadas a sentencias SQL que contengan parámetros de entrada.
<code>prepareCall(String)</code>	Crea un objeto CallableStatement, para ejecutar llamada a Stored Procedures.

© JMA 2021. All rights reserved

Ejecución de sentencias

Una vez que se ha establecido la conexión a la base de datos, podemos enviar sentencias SQL al Gestor.

Las sentencias SQL se pueden enviar al gestor de tres modos diferentes:

- Statement: creación de consultas de estructura estática.
- PreparedStatement: creación de consultas de estructura dinámica.
- CallableStatement: ejecución de procedimientos almacenados en el gestor.

Pueden ser ejecutadas múltiples tipos de instrucciones SQL:

- DQL (Select)
- DML (Insert, Update, Delete)
- DDL (Create, Drop)

© JMA 2021. All rights reserved

Statement

Se utilizan instancias de Statement para utilizar sentencias SQL. Es necesario crear una instancia:

```
Statement stmt = con.createStatement();
```

Dentro de la creación del método createStatement, se puede indicar las características del objeto que devolverá Statement en su ejecución (ResultSet):

- ResultSet.TYPE_FORWARD_ONLY: Solo recorrido adelante
- ResultSet.TYPE_SCROLL_INSENSITIVE: Recorrido Delante-Atrás sin tener en cuenta los cambios producidos
- ResultSet.CONCUR_READ_ONLY: Solo de lectura
- ResultSet.CONCUR_UPDATABLE: Lectura / Escritura

© JMA 2021. All rights reserved

Statement

Se pueden realizar tres tipos de operaciones:

- Consultas de datos: Ejecutar una selección y recuperar un conjunto de resultados:
 - `ResultSet rs = stmt.executeQuery("SELECT ...");`
- Modificaciones: Ejecutar una sentencia que modifique los datos: inserciones, modificaciones y borrado de filas de una tabla, instrucciones DDL y DCL.
 - `[int NumFilas =] stmt.executeUpdate("<CadenaSQL>");`
- General: Se usa cuando no se sabe el tipo de la sentencia o puede devolver varios conjuntos de resultados y/o recuentos. Los resultados se recuperan con los métodos `getResultSet()`, `getMoreResult()` y `getUpdateCount()`.
 - `stmt.execute("<CadenaSQL>");`

© JMA 2021. All rights reserved

PreparedStatement

Las mayoría de los SGBD permiten precompilar las sentencias SQL que van a ser utilizadas reiteradamente, optimizado de esta forma los accesos.

JDBC dispone del interface PreparedStatement (subclase de Statement) para utilizar dicho mecanismo. Es necesario crear una instancia:

```
PreparedStatement prStmt = con.prepareStatement("<SQL>");
```

La cadena SQL puede contar con parámetros, se representan con ? en la posición donde debe ir el argumento.

Es necesario introducir los parámetros antes de ejecutar la sentencia:

```
set<Tipo>(<NúmeroParámetro>, <Valor>);
```

- donde <Tipo> es el tipo del valor introducido y el número (empezando de 1) identifica el orden del parámetro.

Se ejecuta la sentencia con el correspondiente método executeQuery(), executeUpdate() o execute().

```
PreparedStatement pstmt = con.prepareStatement("SELECT * FROM productos WHERE id=? and fecha=?" );
```

```
pstmt.setInt(1,"10");
```

```
pstmt.setDate(2,"03/09/2001");
```

```
ResultSet rs = pstmt.executeQuery();
```

© JMA 2021. All rights reserved

CallableStatement

Como en el caso anterior, la mayoría de SGBD permiten trabajar con procedimientos almacenados, la interface CallableStatement (subclase de PreparedStatement). Se crean las instancias:

```
CallableStatement callStmt = con.prepareCall("<Llamada>");
```

Donde la cadena de invocación al procedimiento sigue el formato:

- Devuelve un valor: {?= call <Procedimiento>[<arg1>,<arg2>, ...]}
- No devuelve valor: {call <Procedimiento>[<arg1>,<arg2>, ...]}

Los parámetros de entrada son manejados de la misma forma que en PreparedStatement.

Los parámetros de salida han de ser registrados para poder recuperarlos:

- callStmt.registerOutParameter(<NúmeroParámetro>, <Tipo>);
- Se ejecuta el procedimiento almacenado con el correspondiente método executeQuery(), executeUpdate() o execute().
- <Tipo> get<Tipo>(<NúmeroParámetro>);

© JMA 2021. All rights reserved

Transacciones

El JDBC soporta el manejo de transacciones aportando los métodos:

- `setTransactionIsolation(<Nivel>)`: Fija el nivel de aislamiento.
 - `TRANSACTION_NONE`
 - No puede realizar transacciones.
 - `TRANSACTION_READ_COMMITTED`
 - Solo accede a data con transacciones terminadas.
 - `TRANSACTION_READ_UNCOMMITTED`
 - Permite consultar los valores aunque no haya terminado la transacción.
 - `TRANSACTION_REPEATABLE_READ`
 - Los datos leídos repetidamente conservan el valor inicial hasta que termine la transacción.
 - `TRANSACTION_SERIALIZABLE`
 - Los datos mantienen el valor inicial hasta que termine la transacción.
- `getTransactionIsolation()`: Devuelve el nivel de aislamiento utilizado.
- `commit()`, `rollback()`: Confirma o descarta la transacción.
- `setAutoCommit()`, `getAutoCommit()` : Fija o indica si se realiza un commit automáticamente después de cada transacción.

© JMA 2021. All rights reserved

ResultSet

Contiene un cursor con los resultados de una consulta o un procedimiento almacenado. El cursor devuelto se encuentra posicionado delante de la primera fila.

Métodos:

- `next()`: Posiciona el cursor en la siguiente fila, devolverá false cuando no tenga mas filas.
- `close()`: cierra el cursor.
- `getMetaData()`: Devuelve una instancia de `ResultSetMetaData` con la estructura de columnas y tipos.

Las columnas deben ser recuperadas de izquierda a derecha y una vez consultado su valor lo pierden. La primera columna tiene el número 1. Se utilizan los métodos:

- `<Tipo> get<Tipo>(<NúmeroColumna> o "<NombreColumna>")`;

© JMA 2021. All rights reserved

ResultSet

El tratamiento del objeto ResultSet depende de las características del mismo:

- Solo lectura
- Lectura y Escritura
- Solo movimiento hacia delante
- Movimiento adelante y atrás

Para obtener los diferentes resultados el cursor es recorrido de forma similar a los objetos Iterator, Enumeration, foreach

```
ResultSet resultado = stm.executeQuery(sql);
while ( resultado.next() ){
    System.out.println(" id producto--> "+resultado.getInt("id"));
    System.out.println(" producto--> "+resultado.getString("nombre"));
}
```

© JMA 2021. All rights reserved

ResultSet: Métodos

getArray(columna)	Devuelve el valor actual de la columna como un array de Java
getByte(columna)	Devuelve el valor actual de la columna como un byte de Java
getDate(columna)	Devuelve el valor actual de la columna como una Fecha de Java
getDouble(columna)	Devuelve el valor actual de la columna como un Double de Java
getInt(columna)	Devuelve el valor actual de la columna como un Int de Java
getString(columna)	Devuelve el valor actual de la columna como un String de Java

absolute(int)	Mueve el cursor a la fila indicada.
beforeFirst()	Mueve el cursor delante de la primera fila del objeto
first()	Mueve el cursor a la primera fila del objeto
previous()	Mueve el cursor a la anterior fila del objeto
next()	Mueve el cursor a la siguiente fila del objeto
afterLast()	Mueve el cursor detrás de la última fila del objeto

cancelRowUpdate()	Cancela los cambios realizados en la fila actual de este objeto
deleteRow()	Elimina la fila actual de este objeto y de la base de datos

© JMA 2021. All rights reserved

Recuperación Datos. Equivalencias

Tabla de equivalencia entre tipos de datos SQL y JAVA	SQL	Java
	CHAR	String
	VARCHAR	String
	LONGVARCHAR	String
	NUMERIC	java.math.BigDecimal
	DECIMAL	java.math.BigDecimal
	BIT	boolean
	TINYINT	byte
	SMALLINT	short
	INTEGER	int
	BIGINT	long
	REAL	float
	FLOAT	double
	DOUBLE	double
	BINARY	byte[]
	VARBINARY	byte[]
	LONGVARBINARY	byte[]
	DATE	java.sql.Date
	TIME	java.sql.Time
	TIMESTAMP	java.sql.Timestamp

© JMA 2021. All rights reserved

ResultSetMetaData

Esta clase está preparada para obtener información acerca de las propiedades de cada una de las columnas, que conforman el cursor/tabla, que fue obtenida en la ejecución de una consulta.

El objeto `ResultSetMetaData` se obtiene a partir del objeto `ResultSet` obtenido en la ejecución de la consulta

```
ResultSetMetaData rsmd = ResultSet.getMetaData();
```

El objeto `ResultSetMetaData` dispone de métodos para obtener:

- `getColumnCount()`: Devuelve el número de columnas del `ResultSet`
- `getTableName(columna)`: Devuelve el nombre de la tabla de la columna indicada
- `getColumnName(columna)`: Devuelve el nombre de la columna indicada
- `getColumnDisplaySize (columna)`: Devuelve el tamaño de la columna indicada
- `getColumnType(columna)`: Devuelve el tipo de la columna como un TIPO Java
- `isNullable(columna)`: Indica si la columna permite nulos

© JMA 2021. All rights reserved

ResultSetMetaData

```
ResultSet conjuntoResultados = instruccion.executeQuery("SELECT nombre,
apellido FROM TEMPLA");
StringBuffer resultados = new StringBuffer();
ResultSetMetaData metaDatos = conjuntoResultados.getMetaData();
int numeroDeColumnas = metaDatos.getColumnCount();
for(int i = 1; i <= numeroDeColumnas; i++)
    resultados.append(metaDatos.getColumnName(i)+"\t");
resultados.append("\n");
while(conjuntoResultados.next()) {
    for(int i = 1; i <= numeroDeColumnas; i++)
        resultados.append(conjuntoResultados.getObject(i) + "\t");
    resultados.append("\n");
}
System.out.println(resultados.toString());
```

© JMA 2021. All rights reserved

DatabaseMetaData

Hay casos en los que se requiere conocer la estructura de una base de datos (nombre y diseño de las tablas, tipos de los campos, etc.).

Los datos que describen la estructura de las bases de datos es lo que se conoce como metadatos.

Los metadatos se obtienen utilizando el método getMetaData de la clase Connection.

```
DatabaseMetaData metadatos=con.getMetaData();
```

Una vez obtenido el objeto se pueden utilizar métodos para obtener información sobre la base de datos.

- Nombre de la Base de Datos
- Tablas de la BBDD
- Esquemas, etc

© JMA 2021. All rights reserved

DatabaseMetaData

- `getDatabaseProductName()`: Devuelve el nombre comercial del sistema gestor de base de datos en uso
- `getDatabaseProductVersion()`: Devuelve la versión de producto del sistema de base de datos en uso
- `getDriverName()`: Devuelve el nombre del driver JDBC en uso
- `getUserName()`: Devuelve el nombre de usuario actual del gestor de bases de datos.
- `getSchemas()`: Devuelve los esquemas de la Base de Datos.
- `getTables(String catálogo, String Esquema, String Tabla, String tipos[])`: Obtiene una tabla de resultados (ResultSet) en la que cada fila es una tabla del catálogo.

© JMA 2021. All rights reserved

DatabaseMetaData

```
DatabaseMetaData dmd = conn.getMetaData();
ResultSet rs1 = dmd.getSchemas();
while (rs1.next()) {
    String ss = rs1.getString(1);
    ResultSet rs2 = dmd.getTables(null, ss, "%", null);
    while (rs2.next())
        System.out.println(rs2.getString(3) + " " +
                           rs2.getString(4));
}
conn.close();
```

© JMA 2021. All rights reserved

Procesos por Lotes

Los procesos por lotes permiten recopilar y lanzar en una sola consulta un conjunto de instrucciones.

Sólo se permiten colocar instrucciones SQL DML y DDL (UPDATE, INSERT,DELETE, CREATE TABLE,..).

Esto se realizar mediante los métodos de la clase Statement:

- addBatch: Permite añadir nuevas instrucciones al proceso por lotes.
- executeBatch: Lanza las instrucciones almacenadas como una única consulta. El resultado de executeBatch es un array de enteros donde cada elemento es el número de filas modificadas por la acción lanzada correspondiente.

```
Statement stat=con.createStatement();
stat.addBatch("CREATE TABLE.....
stat.addBatch("INSERT INTO....
...
int cuentaFilas[]=stat.executeBatch();
```

© JMA 2021. All rights reserved

Auto cierre (v7)

```
try (Connection con = JDBCbridged.getConnection()) {
    String sql = "SELECT country_id, country_name FROM hr.countries WHERE country_name LIKE ?";
    try (PreparedStatement cmd = con.prepareStatement(sql)) {
        cmd.setString(1, "U%");
        try (ResultSet rs = cmd.executeQuery()) {
            while (rs.next()) {
                System.out.println(rs.getString("country_id") + " " + rs.getString(2));
            }
        } catch (SQLException e) {
            System.out.println("ERROR: " + e.getMessage());
        }
    } catch (SQLException e) {
        System.out.println("ERROR: " + e.getMessage());
    }
} catch (SQLException e) {
    System.out.println("ERROR: " + e.getMessage());
}
```

© JMA 2021. All rights reserved

<https://docs.spring.io/spring-framework/docs/current/reference/html/data-access.html#jdbc>

SPRING FRAMEWORK

© JMA 2016. All rights reserved

Introducción

- Spring Framework se encarga de todos los detalles de bajo nivel que pueden hacer de JDBC una API tan tediosa.
- Se puede elegir entre varios enfoques para formar la base del acceso JDBC a la base de datos o combinar varios.
 - JdbcTemplate: es el enfoque clásico y más popular de Spring JDBC. Este es el enfoque de "nivel más bajo" y todos los demás utilizan un JdbcTemplate bajo cubierta.
 - NamedParameterJdbcTemplate: envuelve a JdbcTemplate para proporcionar parámetros con nombre en lugar de los tradicionales ? Como marcadores de posición de JDBC . Este enfoque proporciona una mejor documentación y facilidad de uso cuando se tienen varios parámetros para una declaración SQL.
 - SimpleJdbcInsert y SimpleJdbcCall: optimizar los metadatos de la base de datos para limitar la cantidad de configuración necesaria. Este enfoque simplifica la codificación, basta con proporcionar el nombre de la tabla o procedimiento y un mapa de parámetros que coincidan con los nombres de las columnas. Esto solo funciona si la base de datos proporciona los metadatos adecuados, en caso contrario, se debe proporcionar una configuración explícita de los parámetros.
 - Los objetos RDBMS, incluidos MappingSqlQuery, SqlUpdate y StoredProcedure, requieren que se creen objetos reutilizables y seguros para subprocesos durante la inicialización de la capa de acceso a datos. Este enfoque se basa en JDO Query, en el que define su cadena de consulta, declara parámetros y compila la consulta. Una vez hecho esto, los métodos execute(...), update(...), y findObject(...) se pueden llamar varias veces con diferentes valores en los parámetros.

© JMA 2016. All rights reserved

Conexiones a la base de datos

- Spring obtiene una conexión a la base de datos a través de un DataSource. El DataSource es parte de la especificación JDBC y es una fábrica de conexiones generalizada. Permite que un contenedor o un marco oculten la agrupación de conexiones y los problemas de gestión de transacciones del código de la aplicación.

```
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/sakila"/>
    <property name="username" value="root"/>
    <property name="password" value="root"/>
</bean>
```

- Con Sprint Boot, en application.properties:
spring.datasource.url=jdbc:mysql://localhost:3306/sakila
spring.datasource.username=root
spring.datasource.password=root

© JMA 2016. All rights reserved

Conexiones a la base de datos

```
@Configurable
public class AppConfig {
    @Bean(destroyMethod = "close")
    DataSource dataSource(Environment env) {
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName("com.mysql.jdbc.Driver");
        ds.setUrl("jdbc:mysql://localhost:3306/sakila");
        ds.setUsername("root");
        ds.setPassword("root");
        return ds;
    }

    @Bean
    JdbcTemplate jdbcTemplate(DataSource dataSource) {
        return new JdbcTemplate(dataSource);
    }

    @Bean
    DataSourceTransactionManager dataSourceTransactionManager(DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }
}
```

© JMA 2016. All rights reserved

JdbcTemplate

- JdbcTemplate es la clase central del paquete principal de JDBC.
- Maneja la creación y liberación de recursos, lo que le ayuda a evitar errores comunes, como olvidar cerrar la conexión.
- Realiza las tareas básicas del flujo de trabajo principal de JDBC (como la creación y ejecución de declaraciones), dejando que el código de la aplicación proporcione SQL y extraiga los resultados.
- La clase JdbcTemplate:
 - Ejecuta consultas SQL
 - Actualiza declaraciones y llamadas a procedimientos almacenados
 - Realiza iteraciones sobre las instancias ResultSet y la extracción de valores de devueltos.
 - Detecta las excepciones de JDBC y las traduce a la jerarquía de excepciones genérica y más informativa definida en el paquete org.springframework.dao.

© JMA 2016. All rights reserved

Consultas

- La siguiente consulta obtiene un valor escalar:


```
int rowCount = this.jdbcTemplate.queryForObject("select count(*) from t_actor",
Integer.class);
```
- La siguiente consulta usa una consulta parametrizada:


```
int countOfActorsNamedJoe = this.jdbcTemplate.queryForObject(
    "select count(*) from t_actor where first_name = ?", Integer.class, "Joe");
```
- Para objetos complejos se requiere un mapeador:


```
Actor actor = jdbcTemplate.queryForObject(
    "select first_name, last_name from t_actor where id = ?",
    (resultSet, rowNum) -> {
        Actor newActor = new Actor();
        newActor.setFirstName(resultSet.getString("first_name"));
        newActor.setLastName(resultSet.getString("last_name"));
        return newActor;
    }, 1L);
```

© JMA 2016. All rights reserved

Consultas

- Si el mapeador se reutiliza, se crea como una clase que implementa :

```
class ActorRowMapper implements RowMapper<Actor> {
    @Override
    public Actor mapRow(ResultSet resultSet, int rowNum) throws SQLException {
        Actor newActor = new Actor();
        newActor.setFirstName(resultSet.getString("first_name"));
        newActor.setLastName(resultSet.getString("last_name"));
        return newActor;
    }
}
```
- La siguiente consulta obtiene una lista de entidades:

```
List<Actor> actors = this.jdbcTemplate.query(
    "select first_name, last_name from t_actor",
    new ActorRowMapper());
```
- Existen versiones especializadas: `queryForList`, `queryForMap`, `queryForRowSet` o `queryForStream`.

© JMA 2016. All rights reserved

Modificaciones

- Se puede utilizar el método `update(..)` para realizar operaciones de inserción, actualización y eliminación. Los valores de los parámetros se proporcionan normalmente como argumentos variables o, alternativamente, como una matriz de objetos.

```
jdbcTemplate.update(
    "insert into t_actor (first_name, last_name) values (?, ?)",
    "Leonor", "Watling");
jdbcTemplate.update(
    "update t_actor set last_name = ? where id = ?",
    "Banjo", 5276L);
jdbcTemplate.update(
    "delete from t_actor where id = ?", 5276L);
```

© JMA 2016. All rights reserved

Otras operaciones

- El método `execute(..)` se utiliza para ejecutar cualquier SQL arbitrario. En consecuencia, el método se usa a menudo para declaraciones DDL. Está muy sobrecargado con variantes que aceptan interfaces de devolución de llamada, vinculando matrices de variables, etc.

```
jdbcTemplate.execute("create table mytable (id integer,
name varchar(100))");
```

- Para invocar un procedimiento almacenado:

```
jdbcTemplate.update(
    "call SUPPORT.REFRESH_ACTORS_SUMMARY(?)",
    Long.valueOf(unionId));
```

© JMA 2016. All rights reserved

NamedParameterJdbcTemplate

- La clase `NamedParameterJdbcTemplate` agrega soporte para programar declaraciones JDBC usando parámetros con nombre, en vez de programar declaraciones JDBC usando el clásico comodín '?' como marcador de posición de argumentos.

```
String sql = "select count(*) from T_ACTOR where first_name = :first_name";
int count = this.namedParameterJdbcTemplate.queryForObject(sql, new
    MapSqlParameterSource("first_name", firstName), Integer.class);
```

- Alternativamente, se pueden pasar los parámetros en un `Map` con los nombre y sus valores correspondientes:

```
int count = this.namedParameterJdbcTemplate.queryForObject(sql,
    Collections.singletonMap("first_name", firstName), Integer.class);
```

- Otra opción es `BeanPropertySqlParameterSource` clase, que envuelve un `JavaBean` arbitrario y utiliza las propiedades del `JavaBean` envuelto como fuente de valores de parámetros con nombre.

```
int count = this.namedParameterJdbcTemplate.queryForObject(sql, new
    BeanPropertySqlParameterSource(actor), Integer.class);
```

© JMA 2016. All rights reserved

SimpleJdbcInsert

- La clase SimpleJdbcInsert proporciona una configuración simplificada para las inserciones al aprovechar los metadatos de la base de datos que se pueden recuperar a través del controlador JDBC.

```
SimpleJdbcInsert insertActor = new
    SimpleJdbcInsert(dataSource).withTableName("t_actor");
Map<String, Object> parameters = new HashMap<String,
    Object>(3);
parameters.put("id", actor.getId());
parameters.put("first_name", actor.getFirstName());
parameters.put("last_name", actor.getLastName());
insertActor.execute(parameters);
```

© JMA 2016. All rights reserved

SimpleJdbcInsert

- Se puede BeanPropertySqlParameterSource utilizar para obtener los parámetros.
- Se puede limitar las columnas para una inserción especificando una lista de nombres de columna con el método usingColumns:

```
insertActor = new SimpleJdbcInsert(dataSource)
    .withTableName("t_actor")
    .usingColumns("first_name", "last_name")
    .usingGeneratedKeyColumns("id");
Map<String, Object> parameters = new HashMap<String, Object>(2);
parameters.put("first_name", actor.getFirstName());
parameters.put("last_name", actor.getLastName());
Number newId = insertActor.executeAndReturnKey(parameters);
actor.setId(newId.longValue());
```

© JMA 2016. All rights reserved

SimpleJdbcCall

- La clase SimpleJdbcCall usa los metadatos de la base de datos para buscar los parámetros in y out de un procedimiento almacenado para que no se tenga que declararlos explícitamente.

```
SimpleJdbcCall procReadActor = new SimpleJdbcCall(dataSource)
    .withProcedureName("read_actor");
SqlParameterSource in = new MapSqlParameterSource()
    .addValue("in_id", id);
Map out = procReadActor.execute(in);
Actor actor = new Actor();
actor.setId(id);
actor.setFirstName((String) out.get("out_first_name"));
actor.setLastName((String) out.get("out_last_name"));
```

© JMA 2016. All rights reserved



<https://mybatis.org/mybatis-3/>

© JMA 2016. All rights reserved

Contenidos

- Introducción
- Configuración
- Arquitectura
- Consultas
- Mapeo de resultados
- DML
- SQL Dinámico
- Anotaciones

© JMA 2016. All rights reserved

INTRODUCCIÓN

© JMA 2016. All rights reserved

Introducción

- 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 diccionarios (mapas) y POJOs (Plain Old Java Objects) con registros de base de datos.
- A diferencia de las herramientas ORM, MyBatis no mapea objetos Java a tablas de base de datos sino métodos a sentencias SQL, objetos a parámetros y resultados a objetos.
- MyBatis es software libre y se ha desarrollado bajo Licencia Apache 2.0
- MyBatis es una bifurcación de Apache iBATIS 3.0 (ahora obsoleto) y es mantenido por un equipo que incluye a los creadores originales de iBATIS.

© JMA 2016. All rights reserved

Instalación

- Para usar MyBatis sólo tienes que incluir el fichero mybatis-x.x.x.jar en el classpath.
 - <https://github.com/mybatis/mybatis-3/releases>
- Si usas Maven añade esta dependencia en tu pom.xml:

```
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>x.x.x</version>
</dependency>
```
- Plugin:
 - MyBatisEclipse: Eclipse plugin for MyBatis
 - MyBatis Generator: Code generator for MyBatis and iBATIS
 - MyBatis Dynamic SQL: SQL Generator for MyBatis and Spring JDBC Templates

© JMA 2016. All rights reserved

Arquitectura

- El interfaz principal para trabajar con MyBatis es el `SqlSession`. A través de este interfaz puedes ejecutar comandos, obtener mappers y gestionar transacciones.
- 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.

© JMA 2016. All rights reserved

Arquitectura

- La potencia de MyBatis reside en los Mapped Statements. Ahí 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.
- Desde sus comienzos, MyBatis ha sido siempre un framework XML. La configuración se basa en XML y los mapped statements se definen en XML. MyBatis 3 ofrece una nueva configuración basada en anotaciones. Las anotaciones simplemente ofrecen una forma más sencilla de implementar los mapped statements sin introducir un montón de sobrecarga.

© JMA 2016. All rights reserved

Configuración

- El `SqlSessionFactoryBuilder` tiene métodos `build()` para construir una `SqlSessionFactory` desde un documento XML o un objeto `Configuration` creado por código.

```
String resource = "com/example/mybatis-config.xml";
InputStream inputStream =
    Resources.getResourceAsStream(resource);
SqlSessionFactoryBuilder builder = new
    SqlSessionFactoryBuilder();
SqlSessionFactory factory = builder.build(inputStream);
```

© JMA 2016. All rights reserved

Configuración

```
<?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>
  <settings>
    <setting name="mapUnderscoreToCamelCase" value="true" />
  </settings>
  <environments default="development">
    <environment id="development">
      <transactionManager type="JDBC" />
      <dataSource type="POOLED">
        <property name="driver" value="com.mysql.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://localhost:3306/sakila" />
        <property name="username" value="root" />
        <property name="password" value="root" />
      </dataSource>
    </environment>
  </environments>
  <mappers>
    :
  </mappers>
</configuration>
```

© JMA 2016. All rights reserved

settings

- Son muy importantes para definir cómo se comporta MyBatis en ejecución.

```
<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="autoMappingBehavior" value="PARTIAL"/>
  <setting name="autoMappingUnknownColumnBehavior" value="WARNING"/>
  <setting name="defaultExecutorType" value="SIMPLE"/>
  <setting name="defaultStatementTimeout" value="25"/>
  <setting name="defaultFetchSize" value="100"/>
  <setting name="safeRowBoundsEnabled" value="false"/>
  <setting name="mapUnderscoreToCamelCase" value="false"/>
  <setting name="localCacheScope" value="SESSION"/>
  <setting name="jdbcTypeForNull" value="OTHER"/>
  <setting name="lazyLoadTriggerMethods" value="equals,clone,hashCode,toString"/>
</settings>
```

© JMA 2016. All rights reserved

mappers

- Hay que indicarle a MyBatis dónde encontrar los SQL mapped statements (sentencias SQL mapeadas). Java no ofrece muchas posibilidades de auto-descubrimiento así que la mejor forma es simplemente decirle a MyBatis donde encontrar los ficheros de mapeo:

```
<mappers>
  <!-- Using classpath relative resources -->
  <mapper resource="com/example/mappers/AuthorMapper.xml"/>
  <!-- Using url fully qualified paths -->
  <mapper url="file:///var/mappers/BlogMapper.xml"/>
  <!-- Using mapper interface classes -->
  <mapper class="com.example.mappers.PostMapper"/>
  <!-- Register all interfaces in a package as mappers -->
  <package name="com.example.mappers"/>
</mappers>
```

© JMA 2016. All rights reserved

Configuración por código

- La clase Configuration contiene todo lo que posiblemente necesites conocer de la instancia de SqlSessionFactory.

```
DataSource dataSource = BlogDataSourceFactory.getBlogDataSource();
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);
```

© JMA 2016. All rights reserved

SqlSessionFactory

- SqlSessionFactory tiene métodos que se usan para crear instancias de SqlSession en función a si 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), quieres que MyBatis obtenga una conexión de un datasource o quieres proporcionar tu propia conexión, quieres que MyBatis reúse PreparedStatements y/o haga batch updates (incluyendo inserts y deletes).
- Por defecto: se arranca una transacción (sin auto-commit), se obtiene una nueva conexión con el nivel de aislamiento por defecto de la base de datos y no se reutilizaran PreparedStatements y ni actualizaciones batch.
try (SqlSession session = factory.openSession()) { // auto-close

© JMA 2016. All rights reserved

SqlSession

- 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.
- Métodos de ejecución de sentencias
 - `<T> T selectOne(String statement, Object parameter)`
 - `<E> List<E> selectList(String statement, Object parameter)`
 - `<T> Cursor<T> selectCursor(String statement, Object parameter)`
 - `<K,V> Map<K,V> 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)`

© JMA 2016. All rights reserved

Control de transacción

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

© JMA 2016. All rights reserved

Mapped statements

- Los mapped statements pueden establecerse en ficheros XML, varios por fichero pero todos con un identificador único respecto al namespace (espacio de nombres) del fichero:


```
<?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="com.example.BlogMapper">
  <select id="selectBlog" resultType="Blog">
    select * from Blog where id = #{id}
  </select>
</mapper>
```
- Para realizar una llamada hay que especificar el nombre completo (fully qualified):


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

© JMA 2016. All rights reserved

Mapped statements

- Los mapped statements pueden establecerse mediante anotaciones en interfaces, tantos como se desee. Las anotaciones son mucho más claras para sentencias sencillas, sin embargo, las anotaciones java son limitadas y más complicadas de usar para sentencias complejas:


```
package com.example.mappers;
public interface BlogMapper {
  @Select("SELECT * FROM blog WHERE id = #{id}")
  Blog selectBlog(int id);
}
```
- Para obtener la implementación del mapped statements se solicita al SqlSession:


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

© JMA 2016. All rights reserved

Ciclo de vida

- El `SessionFactoryBuilder` puede instanciarse, usarse y desecharse. No es necesario mantenerla una vez que ya has creado la `SessionFactory`.
- Una vez creado, el `SessionFactory` debería existir durante toda la ejecución de tu aplicación, no debería haber ningún o casi ningún motivo para eliminarlo o recrearlo. El mejor ámbito para el `SessionFactory` es el de aplicación, usando el patrón Singleton o el Static Singleton.
- Las instancias de `Session` deben seguir un patrón Unit of Work, el ámbito adecuado es el de petición (request), método o transacción. No son seguras para multiprocesos y no deben ser compartidas. Cerrar la `Session` es muy importante.
- Los mappers son interfaces que creas como enlace con los mapped statements. Las instancias de mappers se obtienen de una `Session` y por tanto, su ámbito no puede exceder el de la `Session` de la que fueron creados.

© JMA 2016. All rights reserved

CONSULTAS

© JMA 2016. All rights reserved

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.

© JMA 2016. All rights reserved

select

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

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

© JMA 2016. All rights reserved

Tipos

- Los tipos de datos debe ser nombres de clase cualificados (fully qualified): paquete.clase.
- 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:

```
<typeAliases>
  <typeAlias alias="Author" type="com.example.Author"/>
  <typeAlias alias="Blog" type="com.example.Blog"/>
  <typeAlias alias="Post" type="com.example.Post"/>
</typeAliases>
```

© JMA 2016. All rights reserved

Tipos

- Los tipos de datos debe ser nombres de clase cualificados (fully qualified): paquete.clase.
- 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:

```
<typeAliases>
  <typeAlias alias="Author" type="com.example.Author"/>
  <typeAlias alias="Blog" type="com.example.Blog"/>
  <typeAlias alias="Post" type="com.example.Post"/>
</typeAliases>
```

- Hay muchos type aliases pre construidos. No son sensibles a mayúsculas/minúsculas.

© JMA 2016. All rights reserved

Tipos

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

© JMA 2016. All rights reserved

Parameters

- Los parámetros son elementos muy potentes en MyBatis. La sintaxis `#{}` hace que MyBatis genere parámetros de `PreparedStatement` (?) y que asigne los valores a los parámetros de forma segura.


```
<select id="selectPerson" parameterType="int" resultType="hashmap">
  SELECT * FROM PERSON WHERE ID = #{id}
</select>
```
- 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.
- En un objeto complejo el comportamiento es distinto, se buscarán en las propiedades los parámetros.


```
<select id="authUsers" parameterType="User">
  select count(id)
  from users
  where username = #{usr} and password = #{pwd}
</select>
```

© JMA 2016. All rights reserved

Parameters

- Acompañando al nombre del parámetro se puede especificar:
 - el tipo de Java (`javaType`) y el tipo JDBC (`jdbcType`)
 - para los tipos numéricos, `numericScale` permite especificar cuantas posiciones decimales son relevantes.
 - se puede indicar un `TypeHandler` específico (o un alias) para la conversión del tipo.
 - `mode` permite especificar parámetros IN, OUT o INOUT. Si el `mode=OUT` (o INOUT) y el `jdbcType=CURSOR`, se debe especificar un `resultMap`.

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

© JMA 2016. All rights reserved

Sustitución de Strings

- Por defecto, usar la sintaxis `#{}` hace que MyBatis genere propiedades de `PreparedStatement`, esto es más seguro, más rápido y casi siempre la opción adecuada.
- En algunos casos se quiere inyectar un trozo de texto sin modificaciones dentro de la sentencia SQL, para lo que se utiliza `${}` como marcador.


```
SELECT * FROM user WHERE ${column} = #{value}
ORDER BY ${columnName}
```
- No es seguro pasar un texto introducido por el usuario a una sentencia SQL, esto permite ataques de inyección de SQL y se debe sanear antes de interpolar.

© JMA 2016. All rights reserved

Fragmentos

- Este elemento se utiliza para definir un fragmento reusable de código SQL que puede ser incluido en otras sentencias. Puede parametrizarse estáticamente (durante la fase de carga). Los diferentes valores de propiedad pueden variar en las instancias incluidas.

```
<sql id="userColumns"> ${alias}.id,${alias}.username,${alias}.password
</sql>
```
- Puede ser incluido en otras sentencias:

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

© JMA 2016. All rights reserved

MAPEO DE RESULTADOS

© JMA 2016. All rights reserved

Auto mapeo

- MyBatis puede auto mapear los resultados por ti y en el resto de los casos es posible que tengas que crear un ResultMap. También se puede combinar ambas estrategias.
- Al auto mapear resultados, MyBatis obtiene el nombre de columna y busca una propiedad con el mismo nombre sin tener en cuenta las mayúsculas.
- Normalmente las columnas de base de datos se nombran usando mayúsculas y separando las palabras con un subrayado, mientras que las propiedades java se nombran habitualmente siguiendo la notación tipo camelcase. Para habilitar el auto-mapeo entre ellas hay que establecer el parámetro de configuración `mapUnderscoreToCamelCase` a `true`.
- El auto-mapeo funciona incluso cuando hay un ResultMap específico. Cuando esto sucede, para cada result map, todas las columnas que están presentes en el ResultSet y que no tienen un mapeo manual se auto-mapearán. Posteriormente se procesarán los mapeos manuales.
- Hay tres niveles de auto mapeo:
 - NONE: desactiva el auto mapeo. Solo las propiedades mapeadas manualmente se informarán.
 - PARTIAL: auto mapea todos los resultados que no tienen un mapeo anidado definido en su interior (joins).
 - FULL: lo auto mapea todo.

© JMA 2016. All rights reserved

Auto mapeo

- Para utilizar el auto mapeo, se pueden solucionar las discrepancias entre columnas y propiedades reescribiendo las consultas asignando a cada columna un alias que se corresponda con la propiedad:


```
<select id="selectUsers" resultType="com.example.User">
  select
    user_id as "id",
    user_name as "name",
    hashed_password as "password"
  from some_table
  where user_id = #{id}
</select>
```

© JMA 2016. All rights reserved

Mapecto manual

- El elemento resultMap es el elemento más importante y potente de MyBatis. Permite eliminar el 90% del código que requiere el JDBC para obtener datos de ResultSets, y en algunos casos incluso 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 (auto-mapeó), y los statements más complejos requieren sólo la información imprescindible para describir relaciones.

© JMA 2016. All rights reserved

Result Maps

- El elemento resultMap permite crear mapeadores de resultados reutilizables. Se debe indicar un identificador único (id) dentro del namespace que se utiliza para identificar el result map, el nombre completamente cualificado de la clase o el alias del parámetro que se pasará al statement (type) y, opcionalmente, autoMapping que sobrescribe el parametro global autoMappingBehavior.


```
<resultMap id="detailedBlogResultMap" type="Blog">
  </resultMap>
```
- Dentro del elemento se define la estructura del mapeo.

© JMA 2016. All rights reserved

Result Maps

- **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`
- **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
- **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**: una colección de tipos complejos
 - **result mapping anidado**: las colecciones son `resultMaps` en sí mismas o pueden apuntar a otro `resultMap`
- **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.

© JMA 2016. All rights reserved

Mapecto simple

- Estos son los mapeadores 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.). Opcionalmente se puede indicar el tipo de Java (`javaType`), el tipo JDBC (`jdbcType`) y un `TypeHandler` específico (o un alias) para la conversión del tipo.
- 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).


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

© JMA 2016. All rights reserved

constructor

- Por defecto, MyBatis usa el constructor sin parametros para crear una instancia y los setter para rellenar la propiedades.
- La inyección en el constructor permite informar valores durante la instanciación de la clase, sin necesidad de exponer métodos públicos.

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

© JMA 2016. All rights reserved

association

- El elemento association trata las relaciones de tipo “tiene-un”, relaciones 1 a 1 y n a 1. Convierte la clave ajena en un objeto. MyBatis puede hacerlo de dos formas distintas:
 - Nested Select: Ejecutando otra select que devuelve el tipo complejo deseado.


```
<resultMap id="blogResult" type="Blog">
  <association property="author" column="author_id" javaType="Author"
    select="selectAuthor"/>
</resultMap>
```
 - Nested Results: Usando un resultMap anidado que trata con los datos repetidos de resultsets provenientes de joins.


```
<association property="author" javaType="Author">
  <id property="id" column="author_id"/>
  <result property="username" column="author_username"/>
</association>
```
- Por defecto MyBatis sólo crea objetos hijos si al menos una de las columnas mapeadas a las propiedades de dicho objeto es no nula. El atributo columnPrefix permite mapear columnas de un join a un Result Map externo.

© JMA 2016. All rights reserved

collection

- El elemento collection trata las relaciones de tipo “tiene-varios”, relaciones 1 a n. Funciona de una forma muy similar al association.
- La principal diferencia es que la propiedad debe ser una lista o un conjunto.

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

- Se puede mapear sobre un join de forma anidada o utilizar un prefijo para indicar las columnas adecuadas:

```
<collection property="posts" ofType="Post"
resultMap="blogPostResult" columnPrefix="post_"/>
```

© JMA 2016. All rights reserved

discriminator

- El elemento discriminator permite mapear un conjunto de resultados a distintos tipos de datos o a una jerarquía de herencia de clases. El discriminador funciona muy parecido la sentencia switch de Java.
- Una definición de discriminator especifica los atributos columna, que contiene los valores que actúan como selectores, y javaType, para asegurar que se utiliza el tipo de comparación adecuada, por defecto cadena.
- Esta compuesta por un conjuntos de elementos case, y cada uno establece el valor que debe tener la columna selectora para que se aplique su mapeo.
- Si coincide con alguno de los casos del discriminador entonces usará el resultMap especificado en cada caso.
- Si no coincide ninguno de los casos se utilizará solo el resultMap definido fuera del bloque discriminator en caso de que existiera.

© JMA 2016. All rights reserved

discriminator

```
<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>
```

© JMA 2016. All rights reserved

DML

© JMA 2016. All rights reserved

insert

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

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

© JMA 2016. All rights reserved

update

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

<update id="updateAuthor">
  update Author set
    username = #{username}, password = #{password},
    email = #{email}, bio = #{bio}
  where id = #{id}
</update>
```

© JMA 2016. All rights reserved

delete

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

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

© JMA 2016. All rights reserved

Procedimientos almacenados

```
<select id="selectBlog" resultSets="blogs,authors"
  resultMap="blogResult"
  statementType="CALLABLE">
  { call getBlogsAndAuthors(
    #{id, jdbcType=INTEGER, mode=IN},
    #{count, jdbcType=INTEGER, mode=OUT},
    )
  }
</select>
```

© JMA 2016. All rights reserved

SQL DINÁMICO

© JMA 2016. All rights reserved

SQL Dinámico

- Una de las características más potentes de MyBatis ha sido siempre sus capacidades de SQL dinámico. MyBatis aporta 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
-

© JMA 2016. All rights reserved

if

```
<select id="findActiveBlogLike"
      responseType="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>
```

© JMA 2016. All rights reserved

choose, when, otherwise

```
<select id="findActiveBlogLike" responseType="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>
```

© JMA 2016. All rights reserved

where

- 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.

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

© JMA 2016. All rights reserved

set

- El elemento set se puede usar para incluir dinámicamente columnas a modificar y dejar fuera las demás: prefijará dinámicamente con SET y eliminará todas las comas sobrantes que pudieran quedar tras las asignaciones de valor después de que se hayan aplicado las condiciones.

```
<update id="updateAuthorIfNecessary">
  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>
```

© JMA 2016. All rights reserved

trim

- El elemento trim prefijará dinámicamente con el valor del atributo prefix si se cumple alguna de las condiciones contenidas. Del resultado se elimina cualquier cosa que se haya especificado en el atributo prefixOverrides, y que se inserta todo lo incluido en with. El atributo prefixOverrides acepta una lista de textos delimitados por el carácter "|" donde el espacio en blanco es relevante. where y set son especializaciones de trim:

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

```
...
```

```
</trim>
```

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

```
...
```

```
</trim>
```

© JMA 2016. All rights reserved

foreach

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

```
<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>
```

© JMA 2016. All rights reserved

bind

- El elemento bind permite crear una variable a partir de una expresion OGNL y asociarla al contexto.

```
<select id="selectBlogsLike" resultType="Blog">
  <bind name="pattern" value=
    "'%' + _parameter.getTitle() + '%'" />
  SELECT * FROM BLOG
  WHERE title LIKE #{pattern}
</select>
```

© JMA 2016. All rights reserved

selectKey

- MyBatis puede tratar automáticamente las claves autogeneradas. Cuando las bases de datos que no soportan columnas autogeneradas o el driver JDBC no haya incluido aun dicho soporte, se puede utilizar selectKey.

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

- Si order es BEFORE, entonces la obtención de la clave se realizará primero, se informará el campo indicado en keyProperty y se ejecutará la insert. Si es AFTER se ejecuta primero la insert y después la selectKey.

© JMA 2016. All rights reserved

Dialectos de bases de datos

- Si se ha configurado un `databaseIdProvider`, la variable `"_databaseId"` estará disponible en el código dinámico, de esta forma se pueden ajustar las sentencias a las diferencias del SQL según el fabricante de la base de datos.

```
<insert id="insert">
  <selectKey keyProperty="id" resultType="int" order="BEFORE">
    <if test="_databaseId == 'oracle'">
      select seq_users.nextval from dual
    </if>
    <if test="_databaseId == 'db2'">
      select nextval for seq_users from sysibm.sysdummy1"
    </if>
  </selectKey>
  insert into users values ({id}, #{name})
</insert>
```

© JMA 2016. All rights reserved

SQL Builder

- 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. 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.
- MyBatis 3 introduce un concepto un tanto distinto para tratar con el problema, con la clase `SQL`, se puede crear una sentencia SQL en un sólo paso invocando a sus métodos.

© JMA 2016. All rights reserved

SQL Builder

```
String sql =
" 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";
```

© JMA 2016. All rights reserved

SQL Builder

```
private String selectPersonSql() {
return new SQL() {{
    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");
}}.toString();
}
```

© JMA 2016. All rights reserved

ANOTACIONES

© JMA 2016. All rights reserved

Interface

@Mapper

```
public interface UserMapper {
    @Select("SELECT id, name FROM users WHERE name = #{name}")
    User selectById(@Param("name") String value);
    @MapKey("id")
    @Select("SELECT id, name FROM users WHERE name LIKE #{name} || '%'")
    Map<Integer, User> selectByStartingWithName(String name);
    @Options(useGeneratedKeys = true, keyProperty = "id")
    @SelectKey(statement = "SELECT identity('users')", keyProperty = "id", before =
        true, resultType = int.class)
    @Insert("INSERT INTO users (id, name) VALUES(#{id}, #{name})")
    void insert(User user);
    @Update("UPDATE users SET name = #{name} WHERE id = #{id}")
    boolean update(User user);
    @Delete("DELETE FROM users WHERE id = #{id}")
    boolean deleteById(int id);
}
```

© JMA 2016. All rights reserved

Mapeo de resultados

```
@Results(id="UserResult", value = {
    @Result(property = "id", column = "id", id = true),
    @Result(property = "name", column = "name"),
    @Result(property = "email" column = "id", one =
        @One(select = "selectUserEmailById", fetchType =
            FetchType.LAZY)),
    @Result(property = "telephoneNumbers" column = "id",
        many = @Many(select =
            "selectAllUserTelephoneNumberById", fetchType =
                FetchType.LAZY))
})
```

```
@ResultMap("UserResult")
```

© JMA 2016. All rights reserved

Mapeo de resultados

```
@ConstructorArgs({
    @Arg(column = "id", javaType = int.class, id = true),
    @Arg(column = "name", javaType = String.class),
    @Arg(javaType = UserEmail.class, select = "selectUserEmailById", column = "id")
})
```

```
@AutomapConstructor
public User(int id, String name) { ... }
```

```
@TypeDiscriminator(column = "type", javaType = String.class,
    cases = {
        @Case(value = "1", type = PremiumUser.class),
        @Case(value = "2", type = GeneralUser.class),
        @Case(value = "3", type = TemporaryUser.class)
    }
)
```

© JMA 2016. All rights reserved

SQL dinámico

```
@Update({"<script>",
"update Author",
"  <set>",
"    <if test='user != null'>username=#{user},</if>",
"    <if test='pwd != null'>password=#{pwd},</if>",
"    <if test='email != null'>email=#{email},</if>",
"    <if test='bio != null'>bio=#{bio}</if>",
"  </set>",
"where id=#{id}",
"</script>"}
void updateAuthorValues(Author author);
```