1. Acceso a datos con JDBC

## Introducción a Spring Framework JDBC

El valor agregado proporcionado por la abstracción de Spring Framework JDBC es quizás la mejor manera de mostrar la secuencia de las acciones indicadas en la tabla de abajo. La tabla muestra las acciones que Spring se hará cargo y las acciones que son responsabilidad suya, el desarrollador de la aplicación.

Tabla 13.1Spring JDBC - ¿quién hace qué?

|  |  |  |
| --- | --- | --- |
| **Acción** | **Spring** | **Tú** |
| Definir los parámetros de conexión. |  | **X** |
| Abrir la conexión. | **X** |  |
| Especificar la sentencia SQL. |  | **X** |
| Declarar parámetros y proporcionar valores de los parámetros. |  | **X** |
| Elaborar y ejecutar la sentencia. | **X** |  |
| Configurar el bucle para iterar a través de los resultados (si los hay). | **X** |  |
| Hacer el trabajo para cada iteración. |  | **X** |
| Procesar cualquier excepción. | **X** |  |
| Manejar las transacciones. | **X** |  |
| Cerrar connection, statement y resultset. | **X** |  |

Spring Framework se encarga de todos los detalles de bajo nivel que puede hacer JDBC como una API tediosa para desarrollar.

## Escogiendo un enfoque para acceder a las bases de datos JDBC

Usted puede elegir entre varios enfoques para formar la base para acceder a la base de datos JDBC. Además de los tres gustos de JdbcTemplate, un nuevo enfoque SimpleJdbcInsert y SimplejdbcCall optimizan los metadatos de la base de datos, y el estilo del Objeto RDBMS tiene un enfoque más orientado a objetos similares al de Diseño de Consulta JDO. Una vez que usted comience a utilizar uno de estos enfoques, usted todavía podrá mezclarlos y combinarlos para incluir una característica de un enfoque diferente. Todos los enfoques requieren un controlador JDBC 2.0-compatible y algunas características avanzadas requieren un controlador JDBC 3.0.

**Nota**

Spring 3.0 actualiza todos los enfoques siguientes con el soporte de Java 5 tal como los genéricos y varargs.

* **JdbcTemplate** es el clásico enfoque de Spring JDBC y el más popular. Este enfoque de "nivel más bajo" y todos los demás utilizan JdbcTemplate debajo de las sábanas, y todos son actualizados con el soporta de Java 5 tal como los genéricos y varargs.

* **NamedParameterJdbcTemplate** envuelve un JdbcTemplate para proporcionar los nombres de los parámetros en lugar del tradicional marcador de posición "?" de JDBC. Este enfoque proporciona una mejor documentación y facilidad de uso cuando se tienen múltiples parámetros para una sentencia SQL.

* **SimpleJdbcTemplate** combina las operaciones de uso más frecuente de JdbcTemplate y NamedParameterJdbcTemplate.

·

* **SimpleJdbcInsert y SimpleJdbcCall** optimizan el metadato de la base de datos para limitar la cantidad de configuración necesaria. Este enfoque simplifica la codificación de manera que usted sólo necesita proporcionar el nombre de la tabla o un procedimiento y proporcionar un mapa de parámetros que coincidan con los nombres de las columnas. Esto sólo funciona si la base de datos proporciona los metadatos adecuados. Si la base de datos no proporciona estos metadatos, usted tendrá que proporcionar la configuración explícita de los parámetros.

* **Objetos RDBMS, incluyendo MappingSqlQuery, SqlUpdate y** **StoredProcedure** requieren que usted cree objetos reutilizables y thread-safe  durante la inicialización de la capa de acceso a datos. Este enfoque es modelado después de la Consulta JDO donde usted define la cadena de consulta, declara los parámetros, y compila la consulta. Una vez que usted ha hecho esto, ejecute los métodos, pueden ser llamados varias veces con distintos valores en los parámetros pasados.

## Jerarquía de paquetes

La abstracción del framework de Spring Framework JDBC consiste en cuatro paquetes diferentes, a saber core, datasource, object, y support.

El paquete org.springframework.jdbc.core contiene la clase **JdbcTemplate** y sus diferentes interfaces de retro llamada, además de una variedad de clases relacionadas. Un subpaquete llamado org.springframework.jdbc.core.simple contiene la clase SimpleJdbcTemplate y las clases relacionadas SimpleJdbcInsert y SimpleJdbcCall.

Otro subpaquete llamado org.springframework.jdbc.core.namedparam contiene la clase NamedParameterJdbcTemplate y las clases relacionadas de apoyo. Vea la Sección 13.2, "Utilizando las clases principales de JDBC para controlar el procesamiento básico de JDBC y el manejo de errores", Sección 13.4, "JDBC operaciones por lotes", y la Sección 13.5, "Simplificando las operaciones de JDBC con las clases SimpleJdbc".

El paquete org.springframework.jdbc.datasource contiene una clase de utilidad para el fácil acceso a DataSource, y varias implementaciones simples de DataSource que pueden ser utilizadas para probar y ejecutar sin modificar el código JDBC fuera de un contenedor Java EE. Un subpaquete llamado org.springfamework.jdbc.datasource.embedded proporciona soporte para crear instancias de base de datos en memoria utilizando los motores de bases de datos Java, tales como HSQL y H2. Vea la Sección 13.3, "Controlando las conexiones de base de datos" y la Sección 13.8, "Soporte de base de datos Embebidas".

El paquete org.springframework.jdbc.object contiene clases que representan las consultas RDBMS, actualizaciones, y procedimientos almacenados como thread safe y objetos reutilizables. Vea la Sección 13.6, "Modelando operaciones JDBC como objetos Java". Este enfoque es modelado por JDO, aunque, por supuesto los objetos devueltos por las consultas son "desconectados" de la base de datos. Este alto nivel de abstracción de JDBC depende de la abstracción de bajo nivel en el paquete org.springframework.jdbc.core.

El paquete org.springframework.jdbc.support proporciona una funcionalidad de traducción SQLException y algunas clases de utilidad. Las excepciones que se producen durante el procesamiento JDBC son traducidas a excepciones definidas en el paquete org.springframework.dao. Esto significa que utilizando el código, la capa de abstracción de Spring JDBC no necesita implementar un tratamiento de errores de JDBC o un RDBMS específico. Todas las excepciones traducidas no son marcadas, lo que da la opción de capturar las excepciones, las cuales usted puede recuperar mientras que permite a otras excepciones ser propagadas a la persona que la llama. Consulte la Sección 13.2.4, "SQLExceptionTranslator".

## Utilizando las clases principales de JDBC para controlar el procesamiento básico de JDBC y el manejo de errores

### JdbcTemplate

La clase JdbcTemplate es la clase central en el paquete principal de JDBC. Esta maneja la creación y liberación de recursos, que le ayuda a evitar errores comunes, como olvidar cerrar la conexión. Esta realiza las tareas básicas del flujo de trabajo principal de JDBC, como la creación de declaración y ejecución, dejando el código de la aplicación para proporcionar SQL y extraer resultados. La clase JdbcTemplate ejecuta consultas SQL, sentencias de actualización y llamadas a procedimientos almacenados, realiza iteraciones sobre ResultSets y la extracción de valores de los parámetros devueltos. También captura las excepciones JDBC y las traduce a genéricas más informativas, que se define en el paquete org.springframework.dao.

Cuando usted utiliza JdbcTemplate para su código, usted sólo necesita implementar la retro llamada de interfaces, dándoles una declaración claramente definida. La interfaz de reto llamada PreparedStatementCreator crea una sentencia preparada dada una Connection proporcionada por esta clase, proporcionando SQL y todos los parámetros necesarios. Lo mismo ocurre para la interfaz CallableStatementCreator, que crea sentencias que se pueden llamar. La interfaz RowCallbackHandler extrae los valores de cada fila de un ResultSet.

JdbcTemplate puede utilizarse dentro de una aplicación a través de una implementación DAO directa con una referencia a un DataSource, o estar configurada en un contenedor Spring IoC y dada a DAOs como una referencia a un bean.

**Nota**

El DataSource siempre debería estar configurado como un bean en el contenedor de Spring IoC. En el primer caso el bean se entrega al servicio directamente; en el segundo caso se entrega a la plantilla preparada.

Todo el SQL emitido por esta clase se registra en el nivel DEBUG dentro de la categoría correspondiente al nombre completo de la clase de la instancia de la plantilla (por lo general JdbcTemplate, pero puede ser diferente si usted está usando una subclase de la clase JdbcTemplate).

### Ejemplos del uso de la clase JdbcTemplate

Esta sección proporciona algunos ejemplos del uso de la clase JdbcTemplate. Estos ejemplos no son una lista completa de toda la funcionalidad expuesta por JdbcTemplate, ver los Javadocs acompañantes para ello.

**Consultando (SELECT)**

Aquí está una consulta sencilla para conseguir el número de filas en una relación:

**int** rowCount = **this**.jdbcTemplate.queryForInt("select count(\*) from t\_actor");

Una consulta sencilla utilizando una variable de enlace:

**int** countOfActorsNamedJoe = **this**.jdbcTemplate.queryForInt(

             "select count(\*) from t\_actor where first\_name = ?", "Joe");

Consultando por un String:

String lastName = **this**.jdbcTemplate.queryForObject(

             "select last\_name from t\_actor where id = ?",

**new** Object[] { 1212L }, String.**class**);

Consultando y poblando un *único* objeto de dominio:

Actor actor = **this**.jdbcTemplate.queryForObject(

             "select first\_name, last\_name from t\_actor where id = ?",

**new** Object[] { 1212L }, **new** RowMapper<Actor>() {

**public** Actor mapRow(ResultSet rs, **int** rowNum)

**throws** SQLException {

                           Actor actor = **new** Actor();

                           actor.setFirstName(rs.getString("first\_name"));

                           actor.setLastName(rs.getString("last\_name"));

**return** actor;

                    }

             });

Consultando y poblando un número de objetos de dominio:

List<Actor> actors = **this**.jdbcTemplate.query(

             "select first\_name, last\_name from t\_actor",

**new** RowMapper<Actor>() {

**public** Actor mapRow(ResultSet rs, **int** rowNum)

**throws** SQLException {

                           Actor actor = **new** Actor();

                           actor.setFirstName(rs.getString("first\_name"));

                           actor.setLastName(rs.getString("last\_name"));

**return** actor;

                    }

             });

Si los dos últimos fragmentos de código realmente existieran en la misma aplicación, tendría sentido quitar la actual duplicidad en las dos clases internas anónimas RowMapper, y extraerlos en una sola clase (por lo general una clase interna estatic) que luego pueden ser referenciadas por métodos DAO, según sea necesario. Por ejemplo, puede que sea mejor escribir el fragmento de código anterior de la siguiente manera:

**public** List<Actor> findAllActors() {

**return** **this**.jdbcTemplate.query(

                    "select first\_name, last\_name from t\_actor", **new** ActorMapper());

}

**private** **static** **final** **class** ActorMapper **implements** RowMapper<Actor> {

**public** Actor mapRow(ResultSet rs, **int** rowNum) **throws** SQLException {

             Actor actor = **new** Actor();

             actor.setFirstName(rs.getString("first\_name"));

             actor.setLastName(rs.getString("last\_name"));

**return** actor;

       }

}

**Actualizando (INSERT/UPDATE/DELETE) con jdbcTemplate**

Usted puede utilizar el método update(...) para realizar operaciones de inserción, actualización y eliminación. Los valores de los parámetros por lo general proporcionados como argumentos variables o, alternativamente, como una array de objetos.

**this**.jdbcTemplate.update("in*sert into t\_acto*r (first\_name, last\_name) values (?, ?)", "Leonor", "Watling");

**this**.jdbcTemplate.update("update t\_actor set = ? where id = ?", "Banjo", 5276L);

**this**.jdbcTemplate.update("delete from actor where id = ?", Long.valueOf(actorId));

**Otras operaciones jdbcTemplate**

Usted puede utilizar el método execute(..) para ejecutar cualquier SQL arbitrario, y como tal, el método se utiliza a menudo para sentencias DDL. Este es muy sobrecargado con variantes que tienen retro llamada de interfaces y enlazando array de variables, y así sucesivamente.

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

El siguiente ejemplo invoca un sencillo procedimiento almacenado. El soporte para el procedimiento almacenado mas sofisticado se describe más adelante.

**this**.jdbcTemplate.update("call SUPPORT.REFRESH\_ACTORS\_SUMMARY(?)", Long.valueOf(unionId));

### Las mejores prácticas de JdbcTemplate

Una vez *configurado*, las instancias de la clase JdbcTemplate son *threadsafe*. Esto es importante porque significa que se puede configurar una sola instancia de un JdbcTemplate y luego de forma segura inyectar esta referencia *compartida* en múltiples DAOs (o repositorios). JdbcTemplate tiene un estado útil, en el que mantiene una referencia a un DataSource, pero este estado *no* es el estado conversacional.

Una práctica común cuando se utiliza la clase JdbcTemplate (y las clases asociadas SimpleJdbcTemplate y NamedParameterJdbcTemplate) es configurar un DataSource en su archivo de configuración de Spring, y luego la dependencia-inyecta el bean DataSource que compartió en sus clases DAO, el JdbcTemplate es creado en el setter por el DataSource. Esto nos lleva a que los DAOs se vean en parte a lo siguiente:

**public** **class** JdbcCorporateEventDao **implements** CorporateEventDao {

**private** JdbcTemplate jdbcTemplate;

**public** **void** setDataSource(DataSource dataSource) {

**this**.jdbcTemplate = **new** JdbcTemplate(dataSource);

       }

       // JDBC respaldados por las implementaciones de los métodos en el

       // siguiente CorporateEventDao ...

}

La configuración correspondiente podría tener este aspecto.

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

       xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:context=*"http://www.springframework.org/schema/context"*

       xsi:schemaLocation=*"*

*http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.0.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-3.0.xsd"*>

       <bean id=*"corporateEventDao"* class=*"com.example.JdbcCorporateEventDao"*>

             <property name=*"dataSource"* ref=*"dataSource"* />

       </bean>

       <bean id=*"dataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"*

             destroy-method=*"close"*>

             <property name=*"driverClassName"* value=*"${jdbc.driverClassName}"* />

             <property name=*"url"* value=*"${jdbc.url}"* />

             <property name=*"username"* value=*"${jdbc.username}"* />

             <property name=*"password"* value=*"${jdbc.password}"* />

       </bean>

       <context:property-placeholder location=*"jdbc.properties"* />

</beans>

Una alternativa a la configuración explícita es utilizar el componente de detección y el soporte de anotación para la inyección de dependencia. En este caso, usted anota la clase con @Repository (que lo hace un candidato para el componente de detección) y anote el método setter del DataSource con @Autowired.

@Repository

**public** **class** JdbcCorporateEventDao **implements** CorporateEventDao {

**private** JdbcTemplate jdbcTemplate;

       @Autowired

**public** **void** setDataSource(DataSource dataSource) {

**this**.jdbcTemplate = **new** JdbcTemplate(dataSource);

       }

       // JDBC respaldados por las implementaciones de los métodos en el

       // siguiente CorporateEventDao ...

}

El archivo XML de configuración correspondiente se vería como la siguiente:

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

       xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:context=*"http://www.springframework.org/schema/context"*

       xsi:schemaLocation=*"*

*http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.0.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-3.0.xsd"*>

       <!-- Scans within the base package of the application for @Components to

             configure as beans -->

       <context:component-scan base-package=*"org.springframework.docs.test"* />

       <bean id=*"dataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"*

             destroy-method=*"close"*>

             <property name=*"driverClassName"* value=*"${jdbc.driverClassName}"* />

             <property name=*"url"* value=*"${jdbc.url}"* />

             <property name=*"username"* value=*"${jdbc.username}"* />

             <property name=*"password"* value=*"${jdbc.password}"* />

       </bean>

       <context:property-placeholder location=*"jdbc.properties"* />

</beans>

Si usted está utilizando la clase de Spring JdbcDaoSupport, y sus diversos respaldados-JDBC las clases DAO extendidas desde este, entonces su sub-clase hereda un método setDataSource(..) de la clase JdbcDaoSupport. Usted puede elegir si desea heredar desde esta clase. La clase JdbcDaoSupport se proporciona solo como una conveniencia.

Independientemente de cuál de las anteriores plantillas de inicialización usted decide utilizar (o no), rara vez es necesario crear una nueva instancia de una clase JdbcTemplate cada vez que desee ejecutar SQL. Una vez configurado, una instancia de JdbcTemplate es threadsafe. Es posible que usted desee varias instancias de JdbcTemplate si su aplicación tiene acceso a múltiples bases de datos, lo que requiere múltiples DataSources, y posteriormente múltiples JdbcTemplates configurados de formas diferentes.

### NamedParameterJdbcTemplate

La clase NamedParameterJdbcTemplate añade soporte para la programación de sentencias JDBC que utilizan los nombres de parámetros, a diferencia de las sentencias de programación JDBC utilizando sólo el clásico argumento de marcadores de posición ('?'). La clase NamedParameterJdbcTemplate envuelve un JdbcTemplate, y delegados a JdbcTemplate envueltos para hacer gran parte de su trabajo. Esta sección describe sólo aquellas áreas de la clase NamedParameterJdbcTemplate que difieren de la propia JdbcTemplate, es decir, la programación de las sentencias JDBC utilizando nombres de parámetros.

// some JDBC-backed DAO class...

**private** NamedParameterJdbcTemplate namedParameterJdbcTemplate;

**public** **void** setDataSource(DataSource dataSource) {

**this**.namedParameterJdbcTemplate = **new** NamedParameterJdbcTemplate(

                    dataSource);

}

**public** **int** countOfActorsByFirstName(String firstName) {

       String sql = "select count(\*) from T\_ACTOR where first\_name = :first\_name";

       SqlParameterSource namedParameters = **new** MapSqlParameterSource(

                    "first\_name", firstName);

**return** namedParameterJdbcTemplate.queryForInt(sql, namedParameters);

}

Observe el uso de la notación del nombre del parámetro en el valor asignado a la variable sql, y el valor correspondiente que es conectado en la variable namedParameters (de tipo MapSqlParameterSource).

Alternativamente, usted puede pasar a lo largo nombres de parámetros y sus valores correspondientes a una instancia de NamedParameterJdbcTemplate utilizando el estilo basado en Map. El resto de los métodos expuestos por NamedParameterJdbcOperations e implementado por la clase NamedParameterJdbcTemplate que sigue un modelo similar y no se cubre aquí.

El siguiente ejemplo muestra el uso del estilo basado en Map.

// some JDBC-backed DAO class...

**private** NamedParameterJdbcTemplate namedParameterJdbcTemplate;

**public** **void** setDataSource(DataSource dataSource) {

**this**.namedParameterJdbcTemplate = **new** NamedParameterJdbcTemplate(

                    dataSource);

}

**public** **int** countOfActorsByFirstName(String firstName) {

       String sql = "select count(\*) from T\_ACTOR where first\_name = :first\_name";

       Map namedParameters = Collections.singletonMap("first\_name", firstName);

**return** **this**.namedParameterJdbcTemplate

                    .queryForInt(sql, namedParameters);

}

Una característica interesante relacionada con NamedParameterJdbcTemplate (y que existe en el mismo paquete de Java) es la interfaz SqlParameterSource. Usted ya ha visto un ejemplo de la implementación de esta interfaz en uno de los fragmento de códigos anteriores (la clase MapSqlParameterSource). Un SqlParameterSource es una fuente de valores de los parámetros mencionados a un NamedParameterJdbcTemplate. La clase MapSqlParameterSource es una implementación muy simple que es simplemente un adaptador en torno a un java.util.Map, donde las llaves son los nombres de los parámetros y los valores son los valores de los parámetros.

Otra implementación de SqlParameterSource es la clase BeanPropertySqlParameterSource. Esta clase envuelve un JavaBean arbitrario (esto es, una instancia de una clase que se adhiere a las convenciones JavaBean), y utiliza las propiedades del JavaBean envuelto como la fuente de valores de los parámetros mencionados.

**public** **class** Actor {

**private** Long id;

**private** String firstName;

**private** String lastName;

**public** String getFirstName() {

**return** **this**.firstName;

       }

**public** String getLastName() {

**return** **this**.lastName;

       }

**public** Long getId() {

**return** **this**.id;

       }

       // setters omitted...

}

// some JDBC-backed DAO class...

**private** NamedParameterJdbcTemplate namedParameterJdbcTemplate;

**public** **void** setDataSource(DataSource dataSource) {

**this**.namedParameterJdbcTemplate = **new** NamedParameterJdbcTemplate(

                    dataSource);

}

**public** **int** countOfActors(Actor exampleActor) {

       // notice how the named parameters match the properties of the above

       // 'Actor' class

       String sql = "select count(\*) from T\_ACTOR where first\_name = :firstName and last\_name = :lastName";

       SqlParameterSource namedParameters = **new** BeanPropertySqlParameterSource(

                    exampleActor);

**return** **this**.namedParameterJdbcTemplate

                    .queryForInt(sql, namedParameters);

}

Recuerde que la clase NamedParameterJdbcTemplate *envuelve* una plantilla clásica de JdbcTemplate, si usted necesita el acceso a la instancia JdbcTemplate envuelta para acceder a la funcionalidad presente sólo en la clase JdbcTemplate, usted puede utilizar el método getJdbcOperations() para acceder al JdbcTemplate envuelto a través de la interfaz JdbcOperations.

Ver también la sección 13.2.1.2, "Las mejores prácticas de JdbcTemplate" para los lineamientos sobre el uso de la clase NamedParameterJdbcTemplate en el contexto de una aplicación.

### SimpleJdbcTemplate

La clase SimpleJdbcTemplate envuelve el clásico JdbcTemplate y aprovecha las características del lenguaje Java 5, como varargs y autoboxing.

**Nota**

En Spring 3.0, el original JdbcTemplate también es compatible con Java 5-sintaxis mejorada con genéricos y varargs. Sin embargo, SimpleJdbcTemplate proporciona una API simple que funciona mejor cuando usted no necesita el acceso a todos los métodos que el JdbcTemplate ofrece. También, porque SimpleJdbcTemplate fue diseñado para Java 5, este tiene más métodos que aprovechan los varargs debido al orden diferente de los parámetros.

El valor añadido de la clase SimpleJdbcTemplate en el área del azúcar-sintáctico se ilustra mejor con un ejemplo antes-y-después. El siguiente fragmento de código muestra el código de acceso a datos que utiliza el clásico JdbcTemplate, seguido de un fragmento de código que hace el mismo trabajo con SimpleJdbcTemplate.

// classic JdbcTemplate-style...

**private** JdbcTemplate jdbcTemplate;

**public** **void** setDataSource(DataSource dataSource) {

**this**.jdbcTemplate = **new** JdbcTemplate(dataSource);

}

**public** Actor findActor(String specialty, **int** age) {

       String sql = "select id, first\_name, last\_name from T\_ACTOR"

                    + " where specialty = ? and age = ?";

       RowMapper<Actor> mapper = **new** RowMapper<Actor>() {

**public** Actor mapRow(ResultSet rs, **int** rowNum) **throws** SQLException {

                    Actor actor = **new** Actor();

                    actor.setId(rs.getLong("id"));

                    actor.setFirstName(rs.getString("first\_name"));

                    actor.setLastName(rs.getString("last\_name"));

**return** actor;

             }

       };

       // notice the wrapping up of the argumenta in an array

**return** (Actor) jdbcTemplate.queryForObject(sql, **new** Object[] {

                    specialty, age }, mapper);

}

A continuación el mismo método, con el SimpleJdbcTemplate.

// SimpleJdbcTemplate-style...

**private** SimpleJdbcTemplate simpleJdbcTemplate;

**public** **void** setDataSource(DataSource dataSource) {

**this**.simpleJdbcTemplate = **new** SimpleJdbcTemplate(dataSource);

}

**public** Actor findActor(String specialty, **int** age) {

       String sql = "select id, first\_name, last\_name from T\_ACTOR"

                    + " where specialty = ? and age = ?";

       RowMapper<Actor> mapper = **new** RowMapper<Actor>() {

**public** Actor mapRow(ResultSet rs, **int** rowNum) **throws** SQLException {

                    Actor actor = **new** Actor();

                    actor.setId(rs.getLong("id"));

                    actor.setFirstName(rs.getString("first\_name"));

                    actor.setLastName(rs.getString("last\_name"));

**return** actor;

             }

       };

       // notice the use of varargs since the parameter values now come

       // after the RowMapper parameter

**return** **this**.simpleJdbcTemplate.queryForObject(sql, mapper, specialty,

                    age);

}

Vea la Sección 13.2.1.2, "Las mejores prácticas de JdbcTemplate" para las instrucciones sobre cómo utilizar la clase SimpleJdbcTemplate en el contexto de una aplicación.

**Nota**

La clase SimpleJdbcTemplate sólo ofrece un subconjunto de los métodos expuestos en la clase JdbcTemplate. Si usted necesita usar un método de JdbcTemplate que no está definido en SimpleJdbcTemplate, usted siempre podrá acceder al JdbcTemplate subyacente llamando el método getJdbcOperations() en el SimpleJdbcTemplate, que a su vez le permite invocar el método que desee. El único inconveniente es que los métodos de la interfaz JdbcOperations no son de carácter genérico, por lo que usted tiene que volver a emitirlos y así sucesivamente.

### SQLExceptionTranslator

SQLExceptionTranslator es una interfaz para ser implementada por las clases que se pueden traducir entre SQLExceptions y org.springframework.dao.DataAccessException propias de Springs, que es incrédula en lo que respecta a la estrategia de acceso a datos. Las implementaciones pueden ser genéricas (por ejemplo, el uso de códigos SQLSTATE para JDBC) o de propiedad (por ejemplo, el uso de códigos de error de Oracle) para una mayor precisión.

SQLErrorCodeSQLExceptionTranslator es la implementación de SQLExceptionTranslator que es utilizada por defecto. Esta implementación utiliza los códigos específicos del fabricante. Esta es más precisa que la implementación SQLState. Las traducciones de código de error están basadas en los códigos conservados en un tipo de clase JavaBean llamada SQLErrorCodes. Esta clase es creada y poblada por SQLErrorCodesFactory que como su nombre lo indica es una fábrica para crear SQLErrorCodes basados en el contenido de un archivo de configuración llamado sql-error-codes.xml. Este archivo es poblado con los códigos de los fabricantes y basado en el DatabaseProductName tomado de DatabaseMetaData. Los códigos son utilizados para la base de datos actual que usted está utilizando.

SQLErrorCodeSQLExceptionTranslator aplica las reglas de coincidencias en la siguiente secuencia:

**Nota**

SQLErrorCodesFactory es utilizada por defecto para definir los códigos de Error y traducciones de excepciones personalizadas. Ellos son buscados en un archivo llamado sql-error-codes.xml del classpath y de la correspondiente instancia SQLErrorCodes que se encuentra basada en el nombre de la base de datos para los metadatos de la base de datos de la base de datos en uso.

1.    Cualquier traducción personalizada implementada por una subclase. Normalmente, el concreto SQLErrorCodeSQLExceptionTranslator proporcionado, siempre es utilizado por lo que esta regla no se aplica. Esto sólo se aplica si usted realmente ha hecho una implementación de una subclase.

2.    Cualquier implementación personalizada de la interfaz SQLExceptionTranslator que es proporcionada como la propiedad customSqlExceptionTranslator de la clase SQLErrorCodes.

3.    La lista de instancias de la clase CustomSQLErrorCodesTranslation, proporcionada por la propiedad customTranslations de la clase SQLErrorCodes, es buscada por una igual.

4.    El código de error que coincide es aplicado.

5.    Use el traductor de reserva. SQLExceptionSubclassTranslator es el traductor de reserva por defecto. Si esta traducción no está disponible, el siguiente traductor de reserva es SQLStateSQLExceptionTranslator.

Usted puede extender SQLErrorCodeSQLExceptionTranslat:

**public** **class** CustomSQLErrorCodesTranslator **extends**

             SQLErrorCodeSQLExceptionTranslator {

**protected** DataAccessException customTranslate(String task, String sql,

                    SQLException sqlex) {

**if** (sqlex.getErrorCode() == -12345) {

**return** **new** DeadlockLoserDataAccessException(task, sqlex);

             }

**return** **null**;

       }

}

En este ejemplo, el código de error específico -12345 es traducido y otros errores se dejan para ser traducido por la implementación de traducción por defecto. Para utilizar este traductor personalizado, es necesario pasarlo a JdbcTemplate a través del método setExceptionTranslator y utilizar este JdbcTemplate para todo el proceso de acceso a los datos donde este traductor es necesitado. Aquí está un ejemplo de cómo se puede utilizar este traductor personalizado:

**private** JdbcTemplate jdbcTemoplate;

**public** **void** setDataSource(DataSource dataSource) {

       // create a JdbcTemplate and set data source

**this**.jdbcTemplate = **new** JdbcTemplate();

**this**.jdbcTemplate.setDataSource(dataSource);

       // create a custom translator and set the DataSource for the default

       // translation lookup

       CustomSQLErrorCodesTranslator tr = **new** CustomSQLErrorCodesTranslator();

       tr.setDataSource(dataSource);

**this**.jdbcTemplate.setExceptionTranslator(tr);

}

**public** **void** updateShippingCharge(**long** orderId, **long** pct) {

    // use the prepared JdbcTemplate for this update

**this**.jdbcTemplate.update(

        "update orders" +

            " set shipping\_charge = shipping\_charge \* ? / 100" +

            " where id = ?"

        pct, orderId);

}

El traductor personalizado se pasa un data source con el fin de buscar los códigos de error en sql-error-codes.xml.

### Ejecutando sentencias

La ejecución de una sentencia SQL requiere muy poco código. Usted necesita un DataSource y un JdbcTemplate, incluyendo los métodos provechosos que se proporcionan con el JdbcTemplate. El siguiente ejemplo muestra lo mínimo que usted necesita incluir en una clase pero completamente funcional que crea una nueva tabla:

**import** javax.sql.DataSource;

**import** org.springframework.jdbc.core.JdbcTemplate;

**public** **class** ExecuteAStatement {

**private** JdbcTemplate jdbcTemplate;

**public** **void** setDataSource(DataSource dataSource) {

**this**.jdbcTemplate = **new** JdbcTemplate(dataSource);

       }

**public** **void** doExecute() {

**this**.jdbcTemplate

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

       }

}

### Ejecutando consultas

Algunos métodos de consulta devuelven un solo valor. Para recuperar una cuenta o el valor específico de una fila, utilice queryForInt(..), queryForLong(..) o queryForObject(..). Este último convierte el Tipo JDBC devuelto a la clase Java que se pasa como argumento. Si la conversión del tipo no es válida, se lanza una InvalidDataAccessApiUsageException. Aquí hay un ejemplo que contiene dos métodos de consulta, una para un int y una consulta para un String.

**import** javax.sql.DataSource;

**import** org.springframework.jdbc.core.JdbcTemplate;

**public** **class** RunAQuery {

**private** JdbcTemplate jdbcTemplate;

**public** **void** setDataSource(DataSource dataSource) {

**this**.jdbcTemplate = **new** JdbcTemplate(dataSource);

       }

**public** **int** getCount() {

**return** **this**.jdbcTemplate.queryForInt("select count(\*) from mytable");

       }

**public** String getName() {

**return** (String) **this**.jdbcTemplate.queryForObject(

                           "select name from mytable", String.**class**);

       }

**public** **void** setDataSource(DataSource dataSource) {

**this**.dataSource = dataSource;

       }

}

Además de los métodos de consulta que devuelven un solo valor, algunos métodos de devuelven una lista con una entrada por cada fila que devuelve la consulta. El método más genérico es queryForList(..) que devuelve un List donde cada entrada es un Map con cada entrada en el mapa que representa el valor de la columna de esa fila. Si se agrega un método para el ejemplo anterior para recuperar una lista de todas las filas, se vería así:

**private** JdbcTemplate jdbcTemplate;

**public** **void** setDataSource(DataSource dataSource) {

**this**.jdbcTemplate = **new** JdbcTemplate(dataSource);

}

**public** List<Map<String, Object>> getList() {

**return** **this**.jdbcTemplate.queryForList("select \* from mytable");

}

La lista devuelta vería algo como esto:

[{name=Bob, id=1}, {name=Mary, id=2}]

### Actualizando la base de datos

El siguiente ejemplo muestra una columna actualizada para una determinada llave primaria. En este ejemplo, una sentencia SQL tiene marcadores de posición para los parámetros de la fila. Los valores de los parámetros se pueden pasar como varargs o, alternativamente, como un array de objetos. Por lo tanto las primitivas deberían ser envueltas explícitamente en las clases de envoltura primitivas o utilizando auto-boxing.

**import** javax.sql.DataSource;

**import** org.springframework.jdbc.core.JdbcTemplate;

**public** **class** ExecuteAnUpdate {

**private** JdbcTemplate jdbcTemplate;

**public** **void** setDataSource(DataSource dataSource) {

**this**.jdbcTemplate = **new** JdbcTemplate(dataSource);

       }

**public** **void** setName(**int** id, String name) {

**this**.jdbcTemplate.update("update mytable set name = ? where id = ?",

                           name, id);

       }

}

### Recuperando las llaves auto-generadas

Un método update() provechoso, soporta la recuperación de llaves primarias generadas por la base de datos. El método toma un PreparedStatementCreator como primer argumento, y este es el modo en que se especifica la sentencia de inserción requerida. El otro argumento es un KeyHolder, que contiene la llave generada por el regreso exitoso de la actualización. No hay una manera estándar única para crear un PreparedStatement adecuado (que explica por qué la firma del método es la manera en que lo es). El siguiente ejemplo funciona en Oracle, pero no puede funcionar en otras plataformas:

**final** String INSERT\_SQL = "insert into my\_test (name) values(?)";

**final** String name = "Rob";

KeyHolder keyHolder = **new** GeneratedKeyHolder();

jdbcTemplate.update(**new** PreparedStatementCreator() {

**public** PreparedStatement createPreparedStatement(Connection connection)

**throws** SQLException {

      PreparedStatement ps = connection.prepareStatement(INSERT\_SQL, **new** String[] {"id"});

      ps.setString(1, name);

**return** ps;

 }

}, keyHolder);

// keyHolder.getKey() now contains the generated key

## Controlando las conexiones a la base de datos

### DataSource

Spring obtiene una conexión a la base de datos a través de un DataSource. Un DataSource es parte de las especificaciones JDBC y es una fábrica de conexiones generalizada. Esto permite que un contenedor o un framework oculten la agrupación de conexiones y los problemas de la administración de transacciones desde el código de la aplicación. Como desarrollador, usted no necesita saber los detalles sobre cómo conectarse a la base de datos, esto es responsabilidad del administrador que configura el datasource. Lo más probable es que usted llene ambos roles, como desarrollar y probar el código, pero no necesariamente tiene que saber cómo se ha configurado la producción del datasource.

Cuando se utiliza la capa de Spring JDBC, usted obtiene un dataspurce del JNDI o usted configura una propia con una implementación de grupo de conexión proporcionado por un tercero. Las implementaciones más populares son Apache Jakarta Commons DBCP y C3P0. Las implementaciones en la distribución de Spring están concebidas únicamente para propósitos de prueba y no proporcionan la agrupación.

En esta sección se utiliza la implementación de Spring DriverManagerDataSource y algunas implementaciones adicionales se explican más adelante.

**Nota**

Debería utilizar la clase DriverManagerDataSource sólo para propósitos de prueba, ya que no proporciona la agrupación y funcionará mal cuando se hagan varias solicitudes de una conexión.

Usted puede obtener una conexión con DriverManagerDataSource según el procedimiento habitual de obtener una conexión JDBC. Especifique el nombre completo de la clase del controlador JDBC para que el DriverManager pueda cargar la clase del controlador. A continuación, proporcione una URL que varía entre los drivers JDBC. (Consulte la documentación de su controlador para el valor correcto.) A continuación, un nombre de usuario y una contraseña para conectarse a la base de datos. Aquí está un ejemplo de cómo configurar un DriverManagerDataSource en código Java:

DriverManagerDataSource da*taSource = new D*riverManagerDataSource();

dataSource.setDriverClassName("org.hsqldb.jdbcDriver");

dataSource.setUrl("jdbc:hsqldb:hsql://localhost:");

dataSource.setUsername("sa");

dataSource.setPassword("");

Aquí está la correspondiente configuración XML:

<bean id=*"dataSource"*class=*"org.springframework.jdbc.datasource.DriverManagerDataSource"*>

       <property name=*"driverClassName"* value=*"${jdbc.driverClassName}"* />

       <property name=*"url"* value=*"${jdbc.url}"* />

       <property name=*"username"* value=*"${jdbc.username}"* />

       <property name=*"password"* value=*"${jdbc.password}"* />

</bean>

<context:property-placeholder location=*"jdbc.properties"* />

Los siguientes ejemplos muestran la conectividad y configuración básica para el DBCP y C3P0. Para obtener información sobre más opciones que ayudan a controlar las funciones de la agrupación, consulte la documentación del producto para las respectivas implementaciones de agrupación de conexión.

Configuración DBCP:

<bean id=*"dataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"*

       destroy-method=*"close"*>

       <property name=*"driverClassName"* value=*"${jdbc.driverClassName}"* />

       <property name=*"url"* value=*"${jdbc.url}"* />

       <property name=*"username"* value=*"${jdbc.username}"* />

       <property name=*"password"* value=*"${jdbc.password}"* />

</bean>

<context:property-placeholder location=*"jdbc.properties"* />

Configuración C3P0:

<bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*

       destroy-method=*"close"*>

       <property name=*"driverClass"* value=*"${jdbc.driverClassName}"* />

       <property name=*"jdbcUrl"* value=*"${jdbc.url}"* />

       <property name=*"user"* value=*"${jdbc.username}"* />

       <property name=*"password"* value=*"${jdbc.password}"* />

</bean>

<context:property-placeholder location=*"jdbc.properties"* />

**13.1.2            DataSourceUtils**

La clase DataSourceUtils es una clase auxiliar beneficiosa y potente que proporciona métodos estatic para obtener conexiones JNDI y cerrar conexiones si es necesario. Es compatible con hilos-enlazados a las conexiones con, por ejemplo, DataSourceTransactionManager.

**13.1.3            SmartDataSource**

La interfaz SmartDataSource debería ser implementada por clases que puedan proporcionar una conexión a una base de datos relacional. Que extienda la interfaz DataSource para permitir que las clases la utilicen para consultar si la conexión debería estar cerrada después de una operación determinada. Este uso es eficiente cuando se sabe que usted va a reutilizar una conexión.

**13.1.4            AbstractDataSource**

AbstractDataSource es una clase base abstract para las implementaciones DataSource de Spring que implementan el código que es común a todas las implementaciones DataSource. Usted extiende la clase AbstractDataSource si usted está escribiendo su propia implementación de DataSource.

**13.1.5            SingleConnectionDataSource**

La clase SingleConnectionDataSource es una implementación de la interfaz SmartDataSource que envuelve una *única* Connection que no se ha cerrado después de cada uso. Obviamente, esto no es multihilo potente.

Si cualquier código cliente llama a **close** en la suposición de una conexión agrupada, como cuando se utilizan herramientas de persistencia, establezca la propiedad suppressClose a true. Esta configuración devuelve un indicador envuelto que suprime el cierre de la conexión física. Tenga en cuenta que usted no será capaz de lanzar esto a una Connection nativa de Oracle o más similares.

Esto es principalmente una clase de prueba. Por ejemplo, esto permite pruebas fáciles de código fuera un servidor de aplicaciones, junto con un sencillo ambiente JNDI. En contraste con DriverManagerDataSource, este reutiliza la misma conexión todo el tiempo, evitando la creación excesiva de conexiones físicas.

**13.1.6            DriverManagerDataSource**

La clase DriverManagerDataSource es una implementación del estándar de la interfaz DataSource que configura un controlador JDBC plano a través de propiedades bean, y devuelve cada vez una nueva Connection.

Esta implementación es útil para pruebas y ambientes independientes fuera de un contenedor Java EE, ya sea como un bean DataSource en un contenedor Spring IoC, o junto con un sencillo ambiente JNDI. Agrupar-asume que llamar a Connection.close() simplemente cerrará la conexión, por lo que cualquier código de persistencia, un DataSource bien ejecutado debería funcionar. Sin embargo, utilizando un estilo de agrupación de conexiones JavaBean como commons-dbcp es tan fácil, incluso en un ambiente de prueba, que es casi siempre preferible utilizar como una agrupación de conexiones en DriverManagerDataSource.

**13.1.7            TransactionAwareDataSourceProxy**

TransactionAwareDataSourceProxy es un indicador para un destino DataSource, que envuelve a ese destino DataSource para añadir el conocimiento de las transacciones gestionadas por Spring. En este sentido, esto es similar a un DataSource transaccional JNDI conforme a lo dispuesto por un servidor Java EE.

**Nota**

Rara vez es conveniente utilizar esta clase, menos cuando el código ya existente que debe ser llamado y pasado en una implementación de la interface DataSource del estándar JDBC. En este caso, esto es posible para tener todavía el código utilizable, y al mismo tiempo tener este código participando en las transacciones gestionadas por Spring. Generalmente es preferible escribir su propio código nuevo utilizando las abstracciones de alto nivel para la administración de recursos, como JdbcTemplate o DataSourceUtils.

(*Vea el Javadocs de TransactionAwareDataSourceProxy para más detalles.)*

**13.1.8            DataSourceTransactionManager**

La clase DataSourceTransactionManager es una implementación PlatformTransactionManager para un solo datasource JDBC. Esto vincula una conexión JDBC desde el datasource especificado al hilo que se ejecuta actualmente, permitiendo potencialmente la conexión de hilos, uno por cada datasource.

Se requiere que el código de la aplicación recupere la conexión JDBC a través DataSourceUtils.getConnection(DataSource) en lugar del estándar de Java EE DataSource.getConnection. Esto lanza excepciones org.springframework.dao no comprobadas en vez de SQLExceptions comprobadas. Todas las clases del framework como JdbcTemplate utilizan esta estrategia de manera implícita. Si no se utiliza con este gestor de transacciones, la estrategia de búsqueda se comporta exactamente como el common, este puede utilizarse en cualquier caso.

La clase DataSourceTransactionManager soporta los niveles personalizados de aislamiento, y los tiempos de espera que son aplicados como tiempos de espera de sentencias de consulta JDBC. Para soportar este último, el código de la aplicación debe utilizar JdbcTemplate o llamar el método DataSourceUtils.applyTransactionTimeout(..) para cada sentencia creada.

Esta implementación puede utilizarse en lugar de JtaTransactionManager en caso de ser el único recurso, ya que esto no requiere que el contenedor soporte JTA. El cambio entre ambos es sólo una cuestión de configuración, si nos atenemos al modelo de conexión de búsqueda deseado. JTA no soporta niveles de aislamiento personalizados!

**13.1.9            NativeJdbcExtractor**

A veces usted necesita acceder a los métodos JDBC específicos de un fabricante que difieren del estándar de la API JDBC. Esto puede ser problemático si se está ejecutando en un servidor de aplicaciones o con un DataSource que envuelve los objetos Connection, Statement y ResultSet con sus propios objetos envolventes. Para acceder a los objetos nativos usted puede configurar su JdbcTemplate o OracleLobHandler con un NativeJdbcExtractor.

NativeJdbcExtractor viene en una variedad de gustos para adaptarse a su ambiente de ejecución:

·         SimpleNativeJdbcExtractor

·         C3P0NativeJdbcExtractor

·         CommonsDbcpNativeJdbcExtractor

·         JBossNativeJdbcExtractor

·         WebLogicNativeJdbcExtractor

·         WebSphereNativeJdbcExtractor

·         XAPoolNativeJdbcExtractor

Por lo general, SimpleNativeJdbcExtractor es suficiente para desenvolver un objeto Connection en la mayor parte de ambientes. Vea los Javadocs para más detalles.

**13.1 Operaciones por lotes JDBC**

La mayor parte de los controladores JDBC proporcionan un mejor rendimiento si usted llama múltiples lotes a la misma sentencia preparada. Agrupando actualizaciones en lotes usted limita el número de viajes de ida y vuelta a la base de datos. Esta sección cubre el procesamiento por lotes utilizando tanto el JdbcTemplate como el SimpleJdbcTemplate.

**13.1.1            Operaciones básicas de operaciones por lotes con el JdbcTemplate**

Usted logra el procesamiento por lotes JdbcTemplate mediante la implementación de dos métodos de una interfaz especial, BatchPreparedStatementSetter, y pasando el segundo parámetro en la llamada al método batchUpdate. Utilizar el método getBatchSize para proporcionar el tamaño del lote actual. Use el método SetValues para establecer los valores de los parámetros de la sentencia preparada. Este método será llamado el número de veces que usted ha especificado en la llamada de getBatchSize. El siguiente ejemplo actualiza la tabla actor basada en las entradas de una lista. La lista completa se utiliza como el lote en este ejemplo:

**public** **class** JdbcActorDao **implements** ActorDao {

**private** JdbcTemplate jdbcTemplate;

**public** **void** setDataSource(DataSource dataSource) {

**this**.jdbcTemplate = **new** JdbcTemplate(dataSource);

       }

**public** **int**[] batchUpdate(**final** List<Actor> actors) {

**int**[] updateCounts = jdbcTemplate.batchUpdate(

                  "update t\_actor set first\_name = ?, last\_name = ? where id = ?",

**new** BatchPreparedStatementSetter() {

**public** **void** setValues(PreparedStatement ps, **int** i) **throws** SQLException {

                    ps.setString(1, actors.get(i).getFirstName());

                    ps.setString(2, actors.get(i).getLastName());

                    ps.setLong(3, actors.get(i).getId().longValue());

                  }

**public** **int** getBatchSize() {

**return** actors.size();

                  }

             });

**return** updateCounts;

       }

       // ... additional methods

}

Si usted está procesando un flujo de actualizaciones o leyendo de un archivo, entonces usted podría tener un tamaño de lote preferido, pero el último lote puede que no tenga ese número de entradas. En este caso usted puede usar la interfaz InterruptibleBatchPreparedStatementSetter, que le permite interrumpir un lote una vez que la fuente de entrada está agotada. El método isBatchExhausted permite que usted indique el final del lote.

**13.1.2            Operaciones por lotes con una Lista de objetos**

Tanto el JdbcTemplate como el NamedParameterJdbcTemplate proporcionan una forma alternativa de proporcionar la actualización por lotes. En lugar de implementar una interfaz de lote especial, usted proporcionará todos los valores de los parámetros en la llamada como una lista. El marco de bucles sobre estos valores y usos en una sentencia interna setter preparada. La API varía dependiendo de si usted utiliza nombres de parámetros. Para los nombres de los parámetros usted proporciona un array de SqlParameterSource, una entrada para cada miembro del lote. Usted puede utilizar el método SqlParameterSource.createBatch para crear este array, pasando, ya sea en un array de JavaBeans o un array de Maps que contienen los valores de los parámetros.

Este ejemplo muestra una actualización por lotes utilizando nombres de parámetros:

**public** **class** JdbcActorDao **implements** ActorDao {

**private** NamedParameterTemplate namedParameterJdbcTemplate;

**public** **void** setDataSource(DataSource dataSource) {

**this**.namedParameterJdbcTemplate = **new** NamedParameterJdbcTemplate(dataSource);

   }

**public** **int**[] batchUpdate(**final** List<Actor> actors) {

      SqlParameterSource[] batch = SqlParameterSourceUtils.createBatch(actors.toArray());

**int**[] updateCounts = namedParameterJdbcTemplate.batchUpdate(

        "update t\_actor set first\_name = :firstName, last\_name = :lastName where id = :id",

        batch);

**return** updateCounts;

   }

   // ... additional methods

}

Para una sentencia SQL con el clásico marcador de posición "?", usted pasa el contenido de una lista en un array de objetos con los valores actualizados. Esta array de objetos debe tener una entrada por cada marcador de posición en la sentencia SQL, y debe estar en el mismo orden en que están definidos en la sentencia SQL.

El mismo ejemplo utilizando el clásico marcador de posición JDBC "?":

**public** **class** JdbcActorDao **implements** ActorDao {

**private** JdbcTemplate jdbcTemplate;

**public** **void** setDataSource(DataSource dataSource) {

**this**.jdbcTemplate = **new** JdbcTemplate(dataSource);

   }

**public** **int**[] batchUpdate(**final** List<Actor> actors) {

       List<Object[]> batch = **new** ArrayList<Object[]>();

**for** (Actor actor : actors) {

             Object[] values = **new** Object[] { actor.getFirstName(),

                           actor.getLastName(), actor.getId() };

             batch.add(values);

       }

**int**[] updateCounts = jdbcTemplate.batchUpdate(

             "update t\_actor set first\_name = ?, last\_name = ? where id = ?",   batch);

**return** updateCounts;

    }

    // ... additional methods

}

Todos los métodos anteriores de actualización por lotes devuelven un array int que contiene el número de filas afectadas por cada entrada por lotes. Este conteo es reportado por el controlador JDBC. Si el conteo no se encuentra disponible, el controlador JDBC devuelve un valor de -2.

**13.1.3            Operaciones por lotes con múltiples lotes**

El último ejemplo de una actualización por lotes trata con los lotes que son tan grandes que usted desea que se dividan en varios lotes más pequeños. Por supuesto, usted puede hacer esto con los métodos mencionados anteriormente haciendo varias llamadas al método batchUpdate, pero ahora hay un método más beneficioso. Este método tiene, además de la sentencia SQL, unos objetos Collection que contienen los parámetros, el número de actualizaciones a realizar por cada lote y un ParameterizedPreparedStatementSetter para establecer los valores de los parámetros de la sentencia preparada. El marco de los bucles sobre los valores proporcionados y interrumpen las llamadas de las actualización en lotes del tamaño especificado.

Este ejemplo muestra una actualización por lotes usando un tamaño de lote de 100:

**public** **class** JdbcActorDao **implements** ActorDao {

**private** JdbcTemplate jdbcTemplate;

**public** **void** setDataSource(DataSource dataSource) {

**this**.jdbcTemplate = **new** JdbcTemplate(dataSource);

   }

**public** **int**[][] batchUpdate(**final** Collection<Actor> actors) {

       Collection<Object[]> batch = **new** ArrayList<Object[]>();

**for** (Actor actor : actors) {

             Object[] values = **new** Object[] { actor.getFirstName(), actor.getLastName(),

                                               actor.getId() };

              batch.add(values);

       }

**int**[][] updateCounts = jdbcTemplate.batchUpdate(

             "update t\_actor set first\_name = ?, last\_name = ? where id = ?", actors, 100,

**new** ParameterizedPreparedStatementSetter<Actor>() {

**public** **void** setValues(PreparedStatement ps, Actor argument)

**throws** SQLException {

                   ps.setString(1, argument.getFirstName());

                   ps.setString(2, argument.getLastName());

                   ps.setLong(3, argument.getId().longValue());

                }

             });

**return** updateCounts;

   }

   // ... additional methods

}

Los métodos de actualización por lotes para esta llamada devuelven un array de arrays int que contienen array de entrada por cada lote con un array del número de filas afectadas por cada actualización. La longitud del nivel superior del array indica el número de lotes ejecutados y la longitud del segundo nivel del array indica el número de actualizaciones en ese lote. El número de actualizaciones en cada lote debería ser el tamaño proporcionado para todos los lotes excepto el último que podría ser menos, dependiendo del número total de objetos actualizados proporcionados. El conteo de actualizaciones por cada sentencia de actualización es la reportada por el controlador JDBC. Si el conteo no se encuentra disponible, el controlador JDBC devuelve un valor de -2.

**13.1 Simplificando las operaciones JDBC con las clases SimpleJdbc**

Las clases SimpleJdbcInsert y SimpleJdbcCall proporcionan una configuración simplificada aprovechando los metadatos de la base de datos que pueden ser recuperados a través del controlador JDBC. Esto significa menos configuración desde el principio, aunque usted puede reemplazar o desactivar el procesamiento de metadatos, si usted prefiere proporcionar todos los detalles en el código.

**13.1.1            Insertando datos utilizando SimpleJdbcInsert**

Vamos a empezar observando la clase SimpleJdbcInsert con la mínima cantidad de opciones de configuración. Usted debería crear una instancia del SimpleJdbcInsert en el método de inicialización de la capa de acceso de datos. Para este ejemplo, el método de inicialización es el método setDataSource. Usted no necesita una subclase de la clase SimpleJdbcInsert, basta con crear una nueva instancia y establecer el nombre de la tabla utilizando el método withTableName. Los métodos de configuración para esta clase siguen el estilo "fluido" que devuelve la instancia de SimpleJdbcInsert, que le permiten encadenar todos los métodos de configuración. En este ejemplo se utiliza un solo método de configuración, podrás ver ejemplos de múltiples mas tarde.

**public** **class** JdbcActorDao **implements** ActorDao {

**private** SimpleJdbcTemplate simpleJdbcTemplate;

**private** SimpleJdbcInsert insertActor;

**public** **void** setDataSource(DataSource dataSource) {

**this**.simpleJdbcTemplate = **new** SimpleJdbcTemplate(dataSource);

**this**.insertActor = **new** SimpleJdbcInsert(dataSource).withTableName("t\_actor");

       }

**public** **void** add(Actor 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);

       }

       // ... additional methods

}

El método execute utilizado aquí tiene un java.utils.Map plano como su único parámetro. Lo importante a señalar aquí es que las llaves utilizadas para el Map deben coincidir con los nombres de las columnas de la tabla tal como se definen en la base de datos. Esto se debe porque leemos los metadatos con el fin de construir la sentencia de inserción actual.

**13.1.1            Recuperando llaves generadas automáticamente utilizando SimpleJdbcInsert**

En este ejemplo se utiliza la misma inserción como el anterior, pero en lugar de pasar en el id se recupera la llave generada automáticamente y lo establece en el nuevo objeto Actor. Cuando usted crea el SimpleJdbcInsert, además de especificar el nombre de tabla, usted especifica el nombre de la columna de la llave generada con el método usingGeneratedKeyColumns.

**public** **class** JdbcActorDao **implements** ActorDao {

**private** SimpleJdbcTemplate simpleJdbcTemplate;

**private** SimpleJdbcInsert insertActor;

**public** **void** setDataSource(DataSource dataSource) {

**this**.simpleJdbcTemplate = **new** SimpleJdbcTemplate(dataSource);

**this**.insertActor = **new** SimpleJdbcInsert(dataSource).withTableName(

                           "t\_actor").usingGeneratedKeyColumns("id");

       }

**public** **void** add(Actor actor) {

             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());

       }

       // ... additional methods

}

La principal diferencia cuando ejecutamos la inserción de este segundo enfoque es que no se agrega el id al Map y se llama al método executeReturningKey. Este devuelve un objeto java.lang.Number con la que se puede crear una instancia de tipo numérico que se utiliza en nuestra clase de dominio. Usted no puede confiar en todas las bases de datos para devolver una determinada clase de Java aquí; java.lang.Number es la clase base en la cual usted puede confiar. Si usted tiene varias columnas generadas automáticamente, o los valores generados no son numéricos, entonces usted puede utilizar un KeyHolder que es devuelto desde el método executeReturningKeyHolder.

**13.1.2            Especificando columnas para un SimpleJdbcInsert**

Usted puede limitar las columnas para una inserción especificando una lista de nombres de columnas con el método usingColumns:

**public** **class** JdbcActorDao **implements** ActorDao {

**private** SimpleJdbcTemplate simpleJdbcTemplate;

**private** SimpleJdbcInsert insertActor;

**public** **void** setDataSource(DataSource dataSource) {

**this**.simpleJdbcTemplate = **new** SimpleJdbcTemplate(dataSource);

**this**.insertActor = **new** SimpleJdbcInsert(dataSource)

                           .withTableName("t\_actor")

                           .usingColumns("first\_name", "last\_name")

                           .usingGeneratedKeyColumns("id");

       }

**public** **void** add(Actor actor) {

             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());

       }

       // ... additional methods

}

La ejecución de la inserción es el mismo como si usted se hubiese basado en los metadatos para determinar las columnas que desea utilizar.

**13.1.3            Utilizando SqlParameterSource para proporcionar los valores de los parámetros**

Utilizar un Map para proporcionar los valores de los parámetros funciona bien, pero no es la clase más conveniente a utilizar. Spring proporciona un par de implementaciones de la interfaz SqlParameterSource que puede ser utilizado en su lugar. La primera es BeanPropertySqlParameterSource, que es una clase muy conveniente si usted tiene una clase JavaBean compatible que contiene los valores. Esto utilizará el método getter correspondiente para extraer los valores de los parámetros. Aquí está un ejemplo:

**public** **class** JdbcActorDao **implements** ActorDao {

**private** SimpleJdbcTemplate simpleJdbcTemplate;

**private** SimpleJdbcInsert insertActor;

**public** **void** setDataSource(DataSource dataSource) {

**this**.simpleJdbcTemplate = **new** SimpleJdbcTemplate(dataSource);

**this**.insertActor = **new** SimpleJdbcInsert(dataSource).withTableName(

                           "t\_actor").usingGeneratedKeyColumns("id");

       }

**public** **void** add(Actor actor) {

             SqlParameterSource parameters = **new** BeanPropertySqlParameterSource(

                           actor);

             Number newId = insertActor.executeAndReturnKey(parameters);

             actor.setId(newId.longValue());

       }

       // ... additional methods

}

Otra opción es el MapSqlParameterSource que se asemeja a un Map, pero proporciona un método AddValue más conveniente que se pueden encadenar.

**public** **class** JdbcActorDao **implements** ActorDao {

**private** SimpleJdbcTemplate simpleJdbcTemplate;

**private** SimpleJdbcInsert insertActor;

**public** **void** setDataSource(DataSource dataSource) {

**this**.simpleJdbcTemplate = **new** SimpleJdbcTemplate(dataSource);

**this**.insertActor = **new** SimpleJdbcInsert(dataSource).withTableName(

                           "t\_actor").usingGeneratedKeyColumns("id");

       }

**public** **void** add(Actor actor) {

             SqlParameterSource parameters = **new** MapSqlParameterSource().addValue(

                           "first\_name", actor.getFirstName()).addValue("last\_name",

                           actor.getLastName());

             Number newId = insertActor.executeAndReturnKey(parameters);

             actor.setId(newId.longValue());

       }

       // ... additional methods

}

Como puede ver, la configuración es la misma, sólo el código que se ejecuta tiene que cambiar para utilizar estas clases de entrada alternativas.

**13.1.1            Llamando un procedimiento almacenado con SimpleJdbcCall**

La clase SimpleJdbcCall aprovecha los metadatos en la base de datos para buscar los nombres de los parámetros IN y OUT, de modo que usted no tiene que declararlos explícitamente. Usted puede declarar parámetros si lo prefiere para hacer esto, o si usted tiene parámetros tales como ARRAY o STRUCT que no tienen un mapa automático de una clase Java. El primer ejemplo muestra un procedimiento simple que sólo devuelve valores escalares en formato VARCHAR y DATE de una base de datos MySQL. El procedimiento de ejemplo lee una entrada de un actor especificado y devuelve las columnas first\_name, last\_name, y birth\_date en la forma de parámetros out.

**CREATE** **PROCEDURE** read\_actor **(**

**IN** in\_id **INTEGER,**

**OUT** out\_first\_name **VARCHAR(**100**),**

**OUT** out\_last\_name **VARCHAR(**100**),**

**OUT** out\_birth\_date **DATE)**

**BEGIN**

**SELECT** first\_name**,** last\_name**,** birth\_date

**INTO** out\_first\_name**,** out\_last\_name**,** out\_birth\_date

**FROM** t\_actor **where** id **=** in\_id**;**

**END;**

El parámetro in\_id contiene el identificador del actor que usted está buscando. Los parámetros out devuelven los datos leídos de la tabla.

SimpleJdbcCall es declarado en una manera similar a SimpleJdbcInsert. Usted debería instanciar y configurar la clase en el método de inicialización de la capa de acceso a datos. En comparación con la clase StoredProcedure, usted no tiene que crear una subclase y no tienen que declarar los parámetros que se pueden consultar en los metadatos de la base de datos. Lo que sigue es un ejemplo de una configuración de SimpleJdbcCall utilizando el anterior procedimiento almacenado. La única opción de configuración, además del DataSource, es el nombre del procedimiento almacenado.

**public** **class** JdbcActorDao **implements** ActorDao {

**private** SimpleJdbcTemplate simpleJdbcTemplate;

**private** SimpleJdbcCall procReadActor;

**public** **void** setDataSource(DataSource dataSource) {

**this**.simpleJdbcTemplate = **new** SimpleJdbcTemplate(dataSource);

**this**.procReadActor = **new** SimpleJdbcCall(dataSource)

                           .withProcedureName("read\_actor");

       }

**public** Actor readActor(Long id) {

             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"));

             actor.setBirthDate((Date) out.get("out\_birth\_date"));

**return** actor;

       }

       // ... additional methods

}

El código que usted escriba para la ejecución de la llamada implica la creación de un SqlParameterSource que contiene el parámetro IN. Esto es importante para que coincida con el nombre proporcionado por el valor de la entrada con la del nombre del parámetro declarado en el procedimiento almacenado. El caso no tiene por qué coincidir, ya que usted utiliza los metadatos para determinar cómo los objetos de la base de datos deberían ser referidos en un procedimiento almacenado. Que es especificado en el código para el procedimiento almacenado, no es necesariamente la forma en que se almacena en la base de datos. Algunas bases de datos transforman los nombres de todos a mayúscula, mientras que otros utilizan minúsculas o utilizan la letra como es especificado.

El método execute toma los parámetros IN y devuelve un Map que contiene los parámetros out introducido por el nombre tal y como fue especificado en el procedimiento almacenado. En este caso son out\_first\_name, out\_last\_name y out\_birth\_date.

La última parte del método execute crea una instancia de Actor para utilizar y devolver los datos recuperados. Una vez más, es importante utilizar los nombres de los parámetros out, ya que se declaran en el procedimiento almacenado. Además, en el caso de los nombres de los parámetros almacenados out en los resultados del mapa de coincidencias que los nombres de los parámetros out en la base de datos, podría variar entre bases de datos. Para hacer que el código sea más portátil usted debería hacer una búsqueda  insensible a las mayúsculas y minúsculas o instruir a Spring para utilizar un CaseInsensitiveMap del proyecto Jakarta Commons. Para hacer esto último, puede crear su propio JdbcTemplate y establecer la propiedad setResultsMapCaseInsensitive a true. A continuación, usted pasa esta instancia personalizada del JdbcTemplate al constructor de su SimpleJdbcCall. Usted debe incluir el commons-collections.jar en su classpath para que esto funcione. Aquí está un ejemplo de esta configuración:

**public** **class** JdbcActorDao **implements** ActorDao {

**private** SimpleJdbcCall procReadActor;

**public** **void** setDataSource(DataSource dataSource) {

             JdbcTemplate jdbcTemplate = **new** JdbcTemplate(dataSource);

             jdbcTemplate.setResultsMapCaseInsensitive(**true**);

**this**.procReadActor = **new** SimpleJdbcCall(jdbcTemplate)

                           .withProcedureName("read\_actor");

       }

       // ... additional methods

}

Al realizar esta acción, usted evita conflictos en el caso utilizado para los nombres de los parámetros out devueltos.

**13.1.2            Declarando explícitamente los parámetros a utilizar para un SimpleJdbcCall**

Usted ha visto cómo los parámetros se deducen basándose en metadatos, pero usted puede declararlos de forma explícita si lo desea. Usted hace esto creando y configurando SimpleJdbcCall con el método declareParameters, que toma un número variable de objetos SqlParameter como entrada. Consulte la siguiente sección para obtener detalles sobre cómo definir un objeto SqlParameter.

**Nota**

Las declaraciones explícitas son necesarias si la base de datos que usted utiliza no es una base de datos soportada por Spring. Actualmente Spring soporta los metadatos de búsqueda de llamadas a procedimientos almacenados para las siguientes bases de datos: Apache Derby, DB2, MySQL, Microsoft SQL Server, Oracle y Sybase. También soporta la búsqueda de metadatos de funciones almacenadas para: MySQL, Microsoft SQL Server y Oracle.

Usted puede optar por declarar un, algunos o todos los parámetros de forma explícita. Los parámetros del metadato todavía se utilizan en las que no se declaran los parámetros de forma explícita. Para anular todo el proceso de búsquedas de metadatos para los parámetros potenciales y sólo utilizar los parámetros declarados, usted llama al método withoutProcedureColumnMetaDataAccess como parte de la declaración. Supongamos que usted tiene declaradas dos o más firmas de llamada diferentes para una función de la base de datos. En este caso usted llama a useInParameterNames para especificar la lista de los nombres de los parámetros IN a incluir para una firma determinada.

El siguiente ejemplo muestra una llamada a un procedimiento totalmente declarado, utilizando la información del ejemplo anterior.

**public** **class** JdbcActorDao **implements** ActorDao {

**private** SimpleJdbcCall procReadActor;

**public** **void** setDataSource(DataSource dataSource) {

             JdbcTemplate jdbcTemplate = **new** JdbcTemplate(dataSource);

             jdbcTemplate.setResultsMapCaseInsensitive(**true**);

**this**.procReadActor = **new** SimpleJdbcCall(jdbcTemplate)

                    .withProcedureName("read\_actor")

                    .withoutProcedureColumnMetaDataAccess()

                    .useInParameterNames("in\_id")

                    .declareParameters(**new** SqlParameter("in\_id", Types.NUMERIC),

**new** SqlOutParameter("out\_first\_name", Types.VARCHAR),

**new** SqlOutParameter("out\_last\_name", Types.VARCHAR),

**new** SqlOutParameter("out\_birth\_date", Types.DATE));

       }

       // ... additional methods

}

La ejecución y resultados finales de los dos ejemplos son los mismos, éste especifica todos los detalles de forma explícita en lugar de confiar en los metadatos.

**13.1.1            ¿Cómo definir SqlParameters?**

Para definir un parámetro para las clases SimpleJdbc y también para las clases de operaciones RDBMS, contempladas en la [Sección 13.6, "Modelando operaciones JDBC como objetos Java"](file:///D:\LIBROS%20-%20TUTORIALES\SPRING%20FRAMEWORK\13.%20Acceso%20a%20datos%20con%20JDBC.docx#_Modelando_operaciones_JDBC), usted utiliza SqlParameter o una de sus subclases. Por lo general, usted especifica el nombre del parámetro y el tipo SQL en el constructor. El tipo SQL se especifica utilizando las constantes java.sql.Types. Ya hemos visto estas declaraciones como:

**new** SqlParameter("in\_id", Types.NUMERIC),

**new** SqlOutParameter("out\_first\_name", Types.VARCHAR),

La primera línea con el SqlParameter declara un parámetro IN. Los parámetros IN pueden ser utilizados tanto para las llamadas a procedimientos almacenados como para las consultas utilizando el sqlQuery y sus subclases tratadas en la sección siguiente.

La segunda línea con el SqlOutParameter declara un parámetro out para ser utilizado en una llamada a un procedimiento almacenado. También hay un SqlInOutParameter para los parámetros de InOut, los parámetros que proporcionan un valor IN al procedimiento y que también devuelve un valor.

**Nota**

Sólo los parámetros declarados como SqlParameter y SqlInOutParameter serán utilizados para proporcionar valores de entrada. Esto es diferente de la clase StoredProcedure, que por razones anteriores de compatibilidad permite que valores de entrada sean proporcionados para los parámetros declarados como SqlOutParameter.

Para los parámetros IN, además del nombre y el tipo SQL, usted puede especificar una escala para los datos numéricos o un nombre de tipo para los tipos de bases de datos personalizados. Para los parámetros out, usted puede proporcionar un RowMapper para manejar el mapeo de filas devueltas desde un cursor REF. Otra opción es especificar un SqlReturnType que proporciona una oportunidad para definir el manejo personalizado de los valores devueltos.

**13.1.1            Llamando una función almacenada con SimpleJdbcCall**

Usted llama a una función almacenada casi de la misma forma en que usted llama a un procedimiento almacenado, salvo que usted proporciona un nombre de función en lugar de un nombre de procedimiento. Usted utiliza el método withFunctionName como parte de la configuración para indicar que queremos hacer una llamada a una función, y la cadena correspondiente para la llamada a una función es generada. Una ejecución especializada llamada, executeFunction, se utiliza para ejecutar la función y devuelve el valor devuelto por una función como un objeto de un tipo especificado, lo que significa que no tiene que recuperar el valor de retorno del mapa de resultados. Un método beneficioso similar llamado executeObject también está disponible para los procedimientos almacenados que sólo tienen un parámetro out. El siguiente ejemplo se basa en una función almacenada llamado get\_actor\_name que devuelve el nombre completo de un actor. Aquí está el código fuente de MySQL para esta función:

**CREATE** **FUNCTION** get\_actor\_name **(**in\_id **INTEGER)**

**RETURNS** **VARCHAR(**200**)** **READS** **SQL** **DATA**

**BEGIN**

**DECLARE** out\_name **VARCHAR(**200**);**

**SELECT** **concat(**first\_name**,** ' '**,** last\_name**)**

**INTO** out\_name

**FROM** t\_actor **where** id **=** in\_id**;**

**RETURN** out\_name**;**

**END;**

Para llamar a esta función volvemos a crear un SimpleJdbcCall en el método de inicialización.

**public** **class** JdbcActorDao **implements** ActorDao {

**private** SimpleJdbcTemplate simpleJdbcTemplate;

**private** SimpleJdbcCall funcGetActorName;

**public** **void** setDataSource(DataSource dataSource) {

**this**.simpleJdbcTemplate = **new** SimpleJdbcTemplate(dataSource);

             JdbcTemplate jdbcTemplate = **new** JdbcTemplate(dataSource);

             jdbcTemplate.setResultsMapCaseInsensitive(**true**);

**this**.funcGetActorName = **new** SimpleJdbcCall(jdbcTemplate)

                           .withFunctionName("get\_actor\_name");

       }

**public** String getActorName(Long id) {

             SqlParameterSource in = **new** MapSqlParameterSource().addValue("in\_id",

                           id);

             String name = funcGetActorName.executeFunction(String.**class**, in);

**return** name;

       }

       // ... additional methods

}

El método de ejecución utilizado devuelve una String que contiene el valor devuelto desde la llamada a la función.

**13.1.1            Devolviendo un ResultSet/Cursor REF desde un SimpleJdbcCall**

Llamar un procedimiento almacenado o a una función que devuelve un conjunto de resultados es un poco complicado. Algunas bases de datos devuelven conjuntos de resultados durante el procesamiento de los resultados JDBC mientras que otras requieren registrar de forma explícita los parámetros out de un tipo específico. Ambos enfoques necesitan un procesamiento adicional al iterar sobre el conjunto de resultados y procesar las filas devueltas. Con SimpleJdbcCall usted utiliza el método returningResultSet y declarar una implementación RowMapper que se utilizará para un parámetro específico. En el caso de que se devuelva el conjunto de resultados durante el procesamiento de los resultados, no hay nombres definidos, por lo que los resultados devueltos tendrán que coincidir con el orden en el que se declaran las implementaciones RowMapper. El nombre especificado todavía es utilizado para almacenar la lista de los resultados procesados en el mapa de resultados que se devuelve desde la sentencia execute.

El siguiente ejemplo utiliza un procedimiento almacenado que no toma parámetros IN y devuelve todas las filas de la tabla t\_actor. Aquí está el código fuente de MySQL para este procedimiento:

**CREATE** **PROCEDURE** read\_all\_actors**()**

**BEGIN**

**SELECT** a**.**id**,** a**.**first\_name**,** a**.**last\_name**,** a**.**birth\_date **FROM** t\_actor a**;**

**END;**

Para llamar este procedimiento se declara el RowMapper. Debido a que la clase que usted desea mapear sigue las reglas de JavaBean, usted puede utilizar un ParameterizedBeanPropertyRowMapper que es creada pasando en la clase requerida para mapear en el método newInstance.

**public** **class** JdbcActorDao **implements** ActorDao {

**private** SimpleJdbcTemplate simpleJdbcTemplate;

**private** SimpleJdbcCall procReadAllActors;

**public** **void** setDataSource(DataSource dataSource) {

**this**.simpleJdbcTemplate = **new** SimpleJdbcTemplate(dataSource);

             JdbcTemplate jdbcTemplate = **new** JdbcTemplate(dataSource);

             jdbcTemplate.setResultsMapCaseInsensitive(**true**);

**this**.procReadAllActors = **new** SimpleJdbcCall(jdbcTemplate)

                    .withProcedureName("read\_all\_actors").returningResultSet("actors",

                    ParameterizedBeanPropertyRowMapper.newInstance(Actor.**class**));

       }

**public** List getActorsList() {

             Map m = procReadAllActors.execute(**new** HashMap<String, Object>(0));

**return** (List) m.get("actors");

       }

       // ... additional methods

}

La llamada a execute pasa un Map vacío porque esta llamada no tiene ningún parámetro. La lista de Actors es recuperada de los resultados del mapa y devuelta a quien lo llamo.

**13.1 Modelando operaciones JDBC como objetos Java**

El paquete org.springframework.jdbc.object contiene clases que permiten acceder a la base de datos de una manera más orientada a objetos. A modo de ejemplo, usted puede ejecutar consultas y obtener los resultados como una lista que contiene los objetos de negocio con los datos de las columnas relacionales mapeadas a las propiedades del objeto de negocio. Usted también puede ejecutar los procedimientos almacenados y ejecutar sentencias de actualización, eliminación e inserción.

**Nota**

Muchos desarrolladores de Spring creen que varias clases de operación RDBMS descritas a continuación (con la excepción de la clase [StoredProcedure](file:///D:\LIBROS%20-%20TUTORIALES\SPRING%20FRAMEWORK\13.%20Acceso%20a%20datos%20con%20JDBC.docx#_StoredProcedure)) a menudo puede ser remplazada con llamadas directas de JdbcTemplate. A menudo es más fácil escribir un método DAO que simplemente llamar directamente a un método en un JdbcTemplate (en lugar de encapsular una consulta como una clase en toda regla).

Sin embargo, si usted está recibiendo un valor medible al utilizar clases de operaciones RDBMS, continúe utilizando estas clases.

**13.1.1            SqlQuery**

Sqlquery es una clase reutilizable, threadsafe que encapsula una consulta SQL. Las subclases deben implementar el newRowMapper(..) para ofrecer una instancia de RowMapper que puede crear un objeto por cada fila obtenida al iterar sobre el ResultSet que se crea durante la ejecución de la consulta. La clase SqlQuery rara vez es utilizada directamente, porque la subclase MappingSqlQuery proporciona una implementación mucho más provechosa para el mapeo de las filas de las clases de Java. Otras implementaciones que extienden a SqlQuery son MappingSqlQueryWithParameters y UpdatableSqlQuery.

**13.1.2            MappingSqlQuery**

MappingSqlQuery es una consulta reutilizable en la que las subclases concretas deben implementar el método abstracto mapRow(..) para convertir cada fila del ResultSet suministrado en un objeto del tipo especificado. El siguiente ejemplo muestra una consulta personalizada que mapea los datos de la relación t\_actor a una instancia de la clase Actor.

**public** **class** ActorMappingQuery **extends** MappingSqlQuery<Actor> {

**public** ActorMappingQuery(DataSource ds) {

**super**(ds, "select id, first\_name, last\_name from t\_actor where id = ?");

**super**.declareParameter(**new** SqlParameter("id", Types.INTEGER));

             compile();

       }

       @Override

**protected** Actor mapRow(ResultSet rs, **int** rowNumber) **throws** SQLException {

             Actor actor = **new** Actor();

             actor.setId(rs.getLong("id"));

             actor.setFirstName(rs.getString("first\_name"));

             actor.setLastName(rs.getString("last\_name"));

**return** actor;

       }

}

La clase extiende a MappingSqlQuery parametrizada con el tipo Actor. El constructor para esta consulta del cliente toma el DataSource como el único parámetro. En este constructor usted llama al constructor de la superclase con el DataSource y el SQL que debería ejecutarse para recuperar las filas de esta consulta. Este SQL será utilizado para crear un PreparedStatement por lo que puede contener marcadores de posición para los parámetros que deben pasarse durante la ejecución. Usted debe declarar cada parámetro utilizando el método declareParameter pasándolo en un SqlParameter. El SqlParameter toma un nombre y el tipo JDBC como se define en java.sql.Types. Después de definir todos los parámetros, usted llama al método compile() de modo que la sentencia puede ser preparada y ejecutada más tarde. Esta clase es un thread-safe después de que se haya compilado, por lo que, siempre y cuando estas instancias sean creadas cuando el DAO es inicializado se pueden mantener como variables de instancia y ser reutilizadas.

**private** ActorMappingQuery actorMappingQuery;

@Autowired

**public** **void** setDataSource(DataSource dataSource) {

**this**.actorMappingQuery = **new** ActorMappingQuery(dataSource);

}

**public** Customer getCustomer(Long id) {

**return** actorMappingQuery.findObject(id);

}

El método en este ejemplo recupera el cliente con el id que se pasa como único parámetro. Dado que sólo queremos devolver un objeto, simplemente llamamos al método provechoso findObject con el id como parámetro. Si hubiésemos tenido en cambio una consulta que devuelve una lista de objetos y tome parámetros adicionales, utilizaríamos uno de los métodos execute que toma un array de valores de parámetros pasados como varargs.

**public** List<Actor> searchForActors(**int** age, String namePattern) {

       List<Actor> actors = actorSearchMappingQuery.execute(age, namePattern);

**return** actors;

}

**13.1.3            SqlUpdate**

La clase SqlUpdate encapsula una actualización de SQL. Al igual que una consulta, un objeto update es reutilizable, y al igual que todas las clases RdbmsOperation, un update puede tener parámetros y se definen en SQL. Esta clase proporciona un numero de métodos update(..) análogos a los métodos execute(..) de objetos de consulta. La clase SQLUpdate es concreta. Esta puede ser una subclase, por ejemplo, para agregar un método de update personalizado, como en el siguiente fragmento de código donde simplemente se llama a execute. Sin embargo, usted no tiene que subclasificar la clase SqlUpdate ya que fácilmente se puede paramétrizar estableciendo SQL y declarando parámetros.

**import** java.sql.Types;

**import** javax.sql.DataSource;

**import** org.springframework.jdbc.core.SqlParameter;

**import** org.springframework.jdbc.object.SqlUpdate;

**public** **class** UpdateCreditRating **extends** SqlUpdate {

**public** UpdateCreditRating(DataSource ds) {

        setDataSource(ds);

        setSql("update customer set credit\_rating = ? where id = ?");

        declareParameter(**new** SqlParameter("creditRating", Types.*NUMERIC*));

        declareParameter(**new** SqlParameter("id", Types.*NUMERIC*));

        compile();

    }

    /\*\*

     \* **@param** id for the Customer to be updated

     \* **@param** rating the new value for credit rating

     \* **@return** number of rows updated

     \*/

**public** **int** execute(**int** id, **int** rating) {

**return** update(rating, id);

    }

}

**13.1.4            StoredProcedure**

La clase StoredProcedure es un superclase para abstracciones de objetos de procedimientos almacenados RDBMS. Esta clase es abstract, y sus diversos métodos execute(..) tienen acceso protected, previniendo que el uso no sea a través de una subclase que ofrezca una mayor escritura.

La herencia de la propiedad sql será el nombre del procedimiento almacenado en el RDBMS.

Para definir un parámetro para la clase StoredProcedure, usted utiliza SqlParameter o una de sus subclases. Usted debe especificar el nombre del parámetro y el tipo SQL en el constructor como en el fragmento de código siguiente. El tipo SQL se especifica utilizando las constantes java.sql.Types.

**new** SqlParameter("in\_id", Types.NUMERIC),

**new** SqlOutParameter("out\_first\_name", Types.VARCHAR),

La primera línea con el SqlParameter declara un parámetro IN. Los parámetros IN pueden ser utilizados tanto para las llamadas a procedimientos almacenados como para las consultas utilizando el sqlQuery y sus subclases tratadas en la sección siguiente.

La segunda línea con el SqlOutParameter declara un parámetro out para ser utilizado en una llamada a un procedimiento almacenado. También hay un SqlInOutParameter para los parámetros de InOut, los parámetros que proporcionan un valor IN al procedimiento y que también devuelve un valor.

Para los parámetros IN, además del nombre y el tipo SQL, usted puede especificar una escala para los datos numéricos o un nombre de tipo para los tipos de bases de datos personalizados. Para los parámetros out, usted puede proporcionar un RowMapper para manejar el mapeo de filas devueltas desde un cursor REF. Otra opción es especificar un SqlReturnType que le permite definir el manejo personalizado de los valores devueltos.

He aquí un ejemplo de un simple DAO que utiliza un StoredProcedure para llamar a una función, sysdate(), que viene con cualquier base de datos de Oracle. Para utilizar la funcionalidad del procedimiento almacenado usted tiene que crear una clase que extienda a StoredProcedure. En este ejemplo, la clase StoredProcedure es una clase interna, pero si usted necesita reutilizar a StoredProcedure usted la declara como una clase de nivel superior. Este ejemplo no tiene parámetros de entrada, pero es declarado un parámetro de salida como un tipo date utilizando la clase SqlOutParameter. El método execute() ejecuta el procedimiento y extrae la fecha devuelta de los resultados del Map. Los resultados del Map tienen una entrada para cada parámetro de salida declarado, en este caso sólo uno, utilizando el nombre del parámetro como llave.

**import** java.sql.Types;

**import** java.util.Date;

**import** java.util.HashMap;

**import** java.util.Map;

**import** javax.sql.DataSource;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.jdbc.core.SqlOutParameter;

**import** org.springframework.jdbc.object.StoredProcedure;

**public** **class** StoredProcedureDao {

**private** GetSysdateProcedure getSysdate;

       @Autowired

**public** **void** init(DataSource dataSource) {

**this**.getSysdate = **new** GetSysdateProcedure(dataSource);

       }

**public** Date getSysdate() {

**return** getSysdate.execute();

       }

**private** **class** GetSysdateProcedure **extends** StoredProcedure {

**private** **static** **final** String *SQL* = "sysdate";

**public** GetSysdateProcedure(DataSource dataSource) {

                    setDataSource(dataSource);

                    setFunction(**true**);

                    setSql(*SQL*);

                    declareParameter(**new** SqlOutParameter("date", Types.*DATE*));

                    compile();

             }

**public** Date execute() {

                    // the 'sysdate' sproc has no input parameters, so an empty Map is

                    // supplied...

                    Map<String, Object> results = execute(**new** HashMap<String, Object>());

                    Date sysdate = (Date) results.get("date");

**return** sysdate;

             }

       }

}

El siguiente ejemplo de un StoredProcedure tiene dos parámetros de salida (en este caso, cursores REF de Oracle).

**import** oracle.jdbc.OracleTypes;

**import** org.springframework.jdbc.core.SqlOutParameter;

**import** org.springframework.jdbc.object.StoredProcedure;

**import** javax.sql.DataSource;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** TitlesAndGenresStoredProcedure **extends** StoredProcedure {

**private** **static** **final** String *SPROC\_NAME* = "AllTitlesAndGenres";

**public** TitlesAndGenresStoredProcedure(DataSource dataSource) {

**super**(dataSource, *SPROC\_NAME*);

             declareParameter(**new** SqlOutParameter("titles", OracleTypes.CURSOR,

**new** TitleMapper()));

             declareParameter(**new** SqlOutParameter("genres", OracleTypes.CURSOR,

**new** GenreMapper()));

             compile();

       }

**public** Map<String, Object> execute() {

             // again, this sproc has no input parameters, so an empty Map is

             // supplied

**return** **super**.execute(**new** HashMap<String, Object>());

       }

}

Observe cómo las variantes sobrecargadas del método declareParameter(..) que han sido utilizados en el constructor TitlesAndGenresStoredProcedure son pasadas a implementaciones de instancias RowMapper, esto es una forma muy conveniente y de gran alcance para reutilizar la funcionalidad existente. El código para las dos implementaciones RowMapper se proporciona a continuación.

La clase TitleMapper mapea un ResultSet a un objeto de dominio Title por cada fila del ResultSet suministrada:

**import** org.springframework.jdbc.core.RowMapper;

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** com.foo.domain.Title;

**public** **final** **class** TitleMapper **implements** RowMapper<Title> {

**public** Title mapRow(ResultSet rs, **int** rowNum) **throws** SQLException {

        Title title = **new** Title();

        title.setId(rs.getLong("id"));

        title.setName(rs.getString("name"));

**return** title;

    }

}

La clase GenreMapper mapea un ResultSet a un objeto de dominio Genre por cada fila del ResultSet suministrada.

**import** org.springframework.jdbc.core.RowMapper;

**import** java.sql.ResultSet;

**import** java.sql.SQLException;

**import** com.foo.domain.Genre;

**public** **final** **class** GenreMapper **implements** RowMapper<Genre> {

**public** Genre mapRow(ResultSet rs, **int** rowNum) **throws** SQLException {

**return** **new** Genre(rs.getString("name"));

    }

}

Para pasar parámetros a un procedimiento almacenado que tiene uno o más parámetros de entrada en su definición en el RDBMS, usted puede codificar un método inflexible de tipo execute(..) que delegaría a la superclase sin tipo el método execute(parámetros Map) (que tiene acceso protected), por ejemplo:

**import** oracle.jdbc.OracleTypes;

**import** org.springframework.jdbc.core.SqlOutParameter;

**import** org.springframework.jdbc.core.SqlParameter;

**import** org.springframework.jdbc.object.StoredProcedure;

**import** javax.sql.DataSource;

**import** java.sql.Types;

**import** java.util.Date;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** TitlesAfterDateStoredProcedure **extends** StoredProcedure {

**private** **static** **final** String *SPROC\_NAME* = "TitlesAfterDate";

**private** **static** **final** String *CUTOFF\_DATE\_PARAM* = "cutoffDate";

**public** TitlesAfterDateStoredProcedure(DataSource dataSource) {

**super**(dataSource, SPROC\_NAME);

        declareParameter(**new** SqlParameter(CUTOFF\_DATE\_PARAM, Types.DATE);

        declareParameter(**new** SqlOutParameter("titles", OracleTypes.CURSOR,

**new** TitleMapper()));

        compile();

    }

**public** Map<String, Object> execute(Date cutoffDate) {

             Map<String, Object> inputs = **new** HashMap<String, Object>();

             inputs.put(*CUTOFF\_DATE\_PARAM*, cutoffDate);

**return** **super**.execute(inputs);

       }

}

**13.1 Los problemas más comunes con los parámetros y manejo del valor de los datos**

Los problemas comunes con los parámetros y valores de los datos existen en los diferentes enfoques proporcionados por Spring Framework JDBC.

**13.1.1            Proporcionando información para los parámetros de tipo SQL**

Por lo general, Spring determina el tipo SQL de los parámetros basados en el tipo de parámetros IN que son pasados. Es posible proporcionar explícitamente el tipo SQL que se utilizará a la hora de establecer valores de los parámetros. A veces esto es necesario para configurar correctamente los valores NULL.

Usted puede proporcionar información del tipo SQL de varias maneras:

·         Muchos métodos de actualización y consultas de JdbcTemplate toman un parámetro IN adicional en forma de un array int. Este array es utilizado para indicar el tipo SQL del parámetro correspondiente utilizando valores constantes de la clase java.sql.Types. Proporcionan una entrada para cada parámetro.

·         Usted puede utilizar la clase SqlParameterValue para ajustar el valor del parámetro que necesita esta información adicional. Cree una nueva instancia para cada valor y páselo en el tipo SQL y el valor del parámetro en el constructor. Usted también puede proporcionar un parámetro opcional de escala para valores numéricos.

·         Para los métodos que trabajan con nombre de parámetros, utilice las clases SqlParameterSource: BeanPropertySqlParameterSource o MapSqlParameterSource. Ambos tienen métodos para registrar el tipo SQL para cualquiera de los valores de los nombre de los parámetros.

**13.1.2            Manejando objetos BLOB y CLOB**

Usted puede almacenar imágenes, otros objetos binarios, y grandes trozos de texto. Estos objetos grandes son llamados BLOB para datos binarios y CLOB para datos de caracteres. En Spring se pueden manejar estos objetos de gran tamaño usando JdbcTemplate directamente y también cuando se usan abstracciones más altas que proporcionan los objetos RDBMS y las clases SimpleJdbc. Todos estos enfoques utilizan una implementación de la interfaz LobHandler para el manejo actual de los datos LOB. LobHandler proporciona acceso a una clase LobCreator, a través del método getLobCreator, utilizado para crear nuevos objetos LOB para ser insertado.

El LobCreator/LobHandler proporcionan el siguiente soporte para la entrada y salida de LOB:

·         BLOB

o   byte[] – getBlobAsBytes y setBlobAsBytes

o   InputStream – getBlobAsBinaryStream y setBlobAsBinaryStream

·         CLOB

o   String – getClobAsString y setClobAsString

o   InputStream – getClobAsAsciiStream y setClobAsAsciiStream

o   Reader – getClobAsCharacterStream y setClobAsCharacterStream

El siguiente ejemplo muestra cómo crear e insertar un BLOB. Posteriormente se verá cómo leer de nuevo desde la base de datos.

Este ejemplo utiliza un JdbcTemplate y una implementación de AbstractLobCreatingPreparedStatementCallback. Esto implementa un método, SetValues. Este método proporciona un LobCreator que usted utiliza para establecer los valores para las columnas LOB en la sentencia SQL de inserción.

Para este ejemplo suponemos que hay una variable, lobHandler, que ya está establecida en una instancia de un DefaultLobHandler. Por lo general, establezca este valor a través de la inyección de dependencia.

**final** File blobIn = **new** File("spring2004.jpg");

**final** InputStream blobIs = **new** FileInputStream(blobIn);

**final** File clobIn = **new** File("large.txt");

**final** InputStream clobIs = **new** FileInputStream(clobIn);

**final** InputStreamReader clobReader = **new** InputStreamReader(clobIs);

jdbcTemplate.execute("INSERT INTO lob\_table (id, a\_clob, a\_blob) VALUES (?, ?, ?)",

**new** AbstractLobCreatingPreparedStatementCallback(lobHandler) {

**protected** **void** setValues(PreparedStatement ps, LobCreator lobCreator)

**throws** SQLException {

        ps.setLong(1, 1L);

        lobCreator.setClobAsCharacterStream(ps, 2, clobReader, (**int**)clobIn.length());

        lobCreator.setBlobAsBinaryStream(ps, 3, blobIs, (**int**)blobIn.length());

      }

  }

);

blobIs.close();

clobReader.close();

1.    Pase en el lobHandler que en este ejemplo es un simple DefaultLobHandler.

2.    Utilizando el método setClobAsCharacterStream, pase el contenido del CLOB.

3.    Utilizando el método setBlobAsBinaryStream, pase el contenido del BLOB.

Ahora es el momento para leer los datos LOB de la base de datos. Una vez más, usted utiliza un JdbcTemplate con la misma variable de instancia lobHandler y una referencia a un DefaultLobHandler.

List<Map<String, Object>> l = jdbcTemplate.query(

             "select id, a\_clob, a\_blob from lob\_table",

**new** RowMapper<Map<String, Object>>() {

**public** Map<String, Object> mapRow(ResultSet rs, **int** i)

**throws** SQLException {

                           Map<String, Object> results = **new** HashMap<String, Object>();

                           String clobText = lobHandler.getClobAsString(rs, "a\_clob");

                           results.put("CLOB", clobText);

**byte**[] blobBytes = lobHandler.getBlobAsBytes(rs, "a\_blob");

                           results.put("BLOB", blobBytes);

**return** results;

                    }

             });

2.    Utilizando el método getClobAsString, recupere el contenido del CLOB.

3.    Utilizando el método getBlobAsBytes, recupere el contenido del BLOB.

**13.1.3            Pasando en listas de valores para la cláusula IN**

El estándar SQL permite la selección de filas en función de una expresión que incluye una lista variables de valores. Un ejemplo típico sería select \* from T\_ACTOR where id in (1, 2, 3). Esta lista variables no es directamente compatible con sentencias preparadas por el estándar JDBC, usted no puede declarar un número variable de marcadores de posición. Usted necesita un número de variaciones con el número deseado de marcadores de posición preparadas, o usted tiene que generar la cadena SQL de forma dinámica una vez que usted sabe cuantos marcadores de posición se requieren. El nombre del parámetro soportado proporcionado en el NamedParameterJdbcTemplate y SimpleJdbcTemplate toma este último enfoque. Pase los valores IN como un java.util.List de objetos primitivos. Esta lista será utilizada para insertar los marcadores de posición necesarios y pasar los valores durante la ejecución de la sentencia.

**Nota**

Tenga cuidado al pasar muchos valores. El estándar JDBC no garantiza que usted pueda utilizar más de 100 valores para una lista de expresiones IN. Varias bases de datos exceden este número, pero por lo general tienen un límite estricto de cuántos valores están permitidos. El límite de Oracle es de 1000.

Además de los valores primitivos en la lista de valores, usted puede crear un java.util.List de arrays de objetos. Esta lista soportaría varias expresiones definidas para la cláusula IN, tal como select \* from T\_ACTOR where (id, last\_name) in ((1, 'Johnson'), (2, 'Harrop')). Por supuesto, esto requiere que su base de datos soporte esta sintaxis.

**13.1.4            Manejando tipos complejos para llamar procedimientos almacenados**

Cuando usted llama procedimientos almacenados a veces usted puede utilizar tipos complejos específicos para la base de datos. Para acomodar estos tipos, Spring ofrece un SqlReturnType para manejarlos cuando son devueltos desde la llamada al procedimiento almacenado y SqlTypeValue cuando son pasados como parámetro IN al procedimiento almacenado.

Aquí hay un ejemplo para devolver el valor de un objeto STRUCT de Oracle del tipo de usuario declarado ITEM\_TYPE. La interfaz SqlReturnType tiene un único método llamado getTypeValue que debe implementarse. Esta interfaz se utiliza como parte de la declaración de un SqlOutParameter.

**final** TestItem - **new** TestItem(123L, "A test item",

**new** SimpleDateFormat("yyyy-M-d").parse("2010-12-31"););

declareParameter(**new** SqlOutParameter("item", OracleTypes.STRUCT, "ITEM\_TYPE",

**new** SqlReturnType() {

**public** Object getTypeValue(CallableStatement cs, **int** colIndx, **int** sqlType,

  String typeName) **throws** SQLException {

        STRUCT struct = (STRUCT)cs.getObject(colIndx);

        Object[] attr = struct.getAttributes();

        TestItem item = **new** TestItem();

        item.setId(((Number) attr[0]).longValue());

        item.setDescription((String)attr[1]);

        item.setExpirationDate((java.util.Date)attr[2]);

**return** item;

      }

    }));

Usted utiliza SqlTypeValue para pasar en el valor de un objeto Java como TestItem en un procedimiento almacenado. La interfaz SqlTypeValue tiene un único método llamado createTypeValue que usted debe implementar. La conexión activa es pasada, y usted puede utilizar esto para crear objetos de bases de datos específicas tales como StructDescriptors, como se muestra en el siguiente ejemplo, o ArrayDescriptors.

**final** TestItem - **new** TestItem(123L, "A test item",

**new** SimpleDateFormat("yyyy-M-d").parse("2010-12-31"););

SqlTypeValue value = **new** AbstractSqlTypeValue() {

**protected** Object createTypeValue(Connection conn, **int** sqlType, String typeName)

**throws** SQLException {

    StructDescriptor itemDescriptor = **new** StructDescriptor(typeName, conn);

    Struct item = **new** STRUCT(itemDescriptor, conn,

**new** Object[] {

            testItem.getId(),

            testItem.getDescription(),

**new** java.sql.Date(testItem.getExpirationDate().getTime())

        });

**return** item;

  }

};

Este SqlTypeValue puede ser añadido ahora al Map que contiene los parámetros de entrada para la llamada de ejecutar el procedimiento almacenado.

Otro uso para SqlTypeValue es pasando en una array de valores a un procedimiento almacenado de Oracle. Oracle tiene su propia clase interna ARRAY que debe ser utilizada en este caso, y usted puede utilizar el SqlTypeValue para crear una instancia del ARRAY de Oracle y poblarlo con los valores del ARRAY de Java.

**final** Long[] ids = **new** Long[] { 1L, 2L };

SqlTypeValue value = **new** AbstractSqlTypeValue() {

**protected** Object createTypeValue(Connection conn, **int** sqlType,

             String typeName) **throws** SQLException {

             ArrayDescriptor arrayDescriptor = **new** ArrayDescriptor(typeName,    conn);

             ARRAY idArray = **new** ARRAY(arrayDescriptor, conn, ids);

**return** idArray;

       }

};

**13.1 Soporte de base de datos Embebidas**

El paquete org.springframework.jdbc.datasource.embedded proporciona soporte para motores de base de datos embebidas de Java. El soporte para [HSQL](http://www.hsqldb.org/), [H2](http://www.h2database.com/), y [Derby](http://db.apache.org/derby) es proporcionado de forma nativa. Usted también puede utilizar un API extensible para conectar nuevos tipos de bases de datos embebidas e implementaciones de DataSource.

**13.1.1            ¿Por qué utilizar una base de datos embebida?**

Una base de datos embebida es muy útil durante la fase de desarrollo de un proyecto debido a su naturaleza ligera. Los beneficios incluyen la facilidad de configuración, el tiempo de puesta en marcha rápido, capacidad de prueba, y capacidad de evolucionar rápidamente SQL durante el desarrollo.

**13.1.2            Creando una instancia de base de datos embebida utilizando Spring XML**

Si usted desea exponer una instancia de una base de datos embebida como un bean en un Spring ApplicationContext, utilice la etiqueta embedded-database en el espacio de nombres spring-jdbc:

<jdbc:embedded-database id=*"dataSource"*>

       <jdbc:script location=*"classpath:schema.sql"* />

       <jdbc:script location=*"classpath:test-data.sql"* />

</jdbc:embedded-database>

La configuración anterior crea una base de datos HSQL embebida poblada con los recursos SQL schema.sql y testdata.sql en el classpath. La instancia de la base de datos está a disposición del contenedor de Spring como un bean de tipo javax.sql.DataSource. Este bean puede ser inyectado en los objetos de acceso a datos según sea necesario.

**13.1.3            Creando una instancia de base de datos embebida mediante programación**

La clase EmbeddedDatabaseBuilder proporciona un API fluida para la construcción de una base de datos embebida mediante programación. Utilice esta opción cuando usted necesite crear una instancia de una base de datos embebida en un entorno independiente, como un objeto  de acceso a datos de la unidad de pruebas:

EmbeddedDatabaseBuilder builder = **new** EmbeddedDatabaseBuilder();

EmbeddedDatabase db = builder.setType(H2).addScript("my-schema.sql").addScript("my-test-data.sql").build();

// do stuff against the db (EmbeddedDatabase extends javax.sql.DataSource)

db.shutdown()

**13.1.4            Extendiendo el soporte a base de datos embebida**

Spring JDBC soporta bases de datos embebidas que pueden ser extendidas de dos formas:

1.    Implementar EmbeddedDatabaseConfigurer para soportar un nuevo tipo de base de datos embebida, como por ejemplo Apache Derby.

2.    Implementar DataSourceFactory para soportar una nueva implementación de DataSource, como un pool de conexiones, para gestionar las conexiones de bases de datos embebidas.

Usted está invitado a contribuir a nuevas extensiones para la comunidad de Spring en [jira.springframework.org](http://static.springsource.org/spring/docs/current/spring-framework-reference/html/jira.springframework.org).

**13.1.5            Utilizando HSQL**

Spring soporta HSQL 1.8.0 y superiores. HSQL es la base de datos embebida por defecto, si no se especifica el tipo de forma explícita. Para especificar explícitamente HSQL, establezca el atributo type de la etiqueta embedded-database a HSQL. Si usted está utilizando la API de constructor, llame al método setType(EmbeddedDatabaseType) con EmbeddedDatabaseType.HSQL.

**13.1.6            Utilizando H2**

Spring también soporta la base de datos H2. Para habilitar H2, establezca el atributo type de la etiqueta embedded-database a H2. Si usted está utilizando la API de constructor, llame al método setType(EmbeddedDatabaseType) con EmbeddedDatabaseType.H2.

**13.1.7            Utilizando Derby**

Spring también soporta Apache Derby 10.5 y superiores. Para habilitar Derby, establezca el atributo type de la etiqueta embedded-database a Derby. Si usted está utilizando la API de constructor, llame al método setType(EmbeddedDatabaseType) con EmbeddedDatabaseType.Derby.

**13.1.8            Probando la lógica de acceso a datos con una base de datos embebida**

Las bases de datos embebidas proporcionan una forma sencilla para probar el código de acceso a datos. La siguiente es una plantilla prueba de unidad de acceso a datos que utiliza una base de datos embebida:

**public** **class** DataAccessUnitTestTemplate {

**private** EmbeddedDatabase db;

       @Before

**public** **void** setUp() {

             // creates a HSQL in-memory db populated from default scripts

             // classpath:schema.sql and classpath:test-data.sql

             db = **new** EmbeddedDatabaseBuilder().addDefaultScripts().build();

       }

@Test

**public** **void** testDataAccess() {

        JdbcTemplate template = **new** JdbcTemplate(db);

        template.query(...);

       }

       @After

**public** **void** tearDown() {

             db.shutdown();

       }

}

**13.1 Inicializando un DataSource**

El paquete org.springframework.jdbc.datasource.init proporciona soporte para la inicialización de un DataSource ya existente. El soporte de base de datos embebida ofrece una opción para la creación y la inicialización de un DataSource para una aplicación, pero a veces usted necesita inicializar una instancia que se ejecute en un servidor en alguna parte.

**13.1.1            Inicializando una instancia de base de datos utilizando XML Spring**

Si usted desea inicializar una base de datos y usted puede proporcionar una referencia a un bean DataSource, utilice la etiqueta initialize-database del espacio de nombres spring-jdbc:

<jdbc:initialize-database data-source=*"dataSource"*>

       <jdbc:script location=*"classpath:com/foo/sql/db-schema.sql"* />

       <jdbc:script location=*"classpath:com/foo/sql/db-test-data.sql"* />

</jdbc:initialize-database>

El ejemplo anterior se ejecutan dos scripts que especifican la base de datos: el primer script es una creación de esquemas, y el segundo es una inserción de  datos de prueba. Las ubicaciones de los scripts también pueden ser patrones con comodines en el estilo ant habitual utilizado por los recursos en Spring (por ejemplo classpath\*:/com/foo/\*\*/sql/\*-data.sql). Si un patrón es utilizado, los scripts se ejecutan en el orden léxico de su URL o nombre de archivo.

El comportamiento predeterminado del inicializador de la base de datos es ejecutar de manera incondicional los scripts proporcionados. Esto no siempre será lo que usted quiere, por ejemplo, si se ejecuta contra una base de datos existente que ya tiene los datos de prueba en el mismo. La probabilidad de la eliminación accidental de datos es reducida por el patrón más común (como se muestra arriba), que crea las tablas y luego inserta los datos - el primer paso producirá un error si las tablas ya existen.

Sin embargo, para obtener un mayor control sobre la creación y eliminación de los datos existentes, el espacio de nombres XML proporciona un par más de opciones. La primera es la bandera para cambiar la inicialización encendido y apagado. Esto se puede establecer de acuerdo al entorno (por ejemplo, para sacar un valor booleano de las propiedades del sistema o un entorno bean), por ejemplo.

<jdbc:initialize-database data-source=*"dataSource"*

       enabled=*"#{systemProperties.INITIALIZE\_DATABASE}"*>

       <jdbc:script location=*"..."* />

</jdbc:initialize-database>

La segunda opción para controlar lo que sucede con los datos existentes es ser más tolerantes con los fallos. Para ello usted puede controlar la capacidad del inicializador para ignorar ciertos errores en el SQL que se ejecuta desde los scripts, por ejemplo.

<jdbc:initialize-database data-source=*"dataSource"*    ignore-failures=*"DROPS"*>

       <jdbc:script location=*"..."* />

</jdbc:initialize-database>

En este ejemplo estamos diciendo que esperamos que a veces los scripts se ejecutaran contra una base de datos vacía y hay algunas sentencias DROP en los scripts que por lo tanto sería un fracaso. Entonces las sentencias SQL DROP falladas se tendrán en cuenta, pero otros fallos provocaran una excepción. Esto es útil si su lenguaje SQL no es compatible con DROP ... IF EXIST (o similar), pero que desea eliminar incondicionalmente todos los datos de prueba antes de volver a crearla. En ese caso, el primer script es generalmente un conjunto de drops, seguido por un conjunto de sentencias CREATE.

La opción ignore-failures se puede establecer a NONE (por defecto), DROPS (ignorar drops fallidos) o ALL (ignorar todos los fallos).

Si usted necesita más control de lo que recibe del espacio de nombres XML, puede simplemente utilizar DataSourceInitializer directamente, y definirlo como un componente en su aplicación.

**13.1.1.1          Inicialización de Otros Componentes que Dependen de la Base de Datos**

Una gran clase de aplicaciones sólo puede utilizar el inicializador de base de datos sin complicaciones: aquellos que no utilizan la base de datos hasta después de que el Spring context ha comenzado. Si su aplicación *no* es uno de aquellos entonces usted puede que tenga que leer el resto de esta sección.

El inicializador de base de datos depende de una instancia de un datasource y ejecuta los scripts proporcionados en su retro llamada de inicialización (c.f. init-method en una definición bean XML o InitializingBean). Si otros beans dependen del mismo datasource y también utilizan el datasource en una retro llamada de inicialización entonces podría haber un problema porque los datos aún no se han inicializado. Un ejemplo común de esto es un caché que se inicializa con entusiasmo y carga los datos desde la base de datos al iniciar la aplicación.

Para solucionar este problema usted tiene dos opciones: cambie su estrategia de inicialización de caché a una fase posterior, o asegúrese de que el inicializador de la base de datos se inicia en primer lugar.

La primera opción podría ser fácil si la aplicación está bajo su control, y no lo contrario. Algunas sugerencias sobre cómo implementar este son:

·         Haga que el caché inicie perezosamente en el primer uso, que mejora el tiempo de inicio de la aplicación.

·         Tenga su caché o un componente separado que inicializa la memoria caché de la implementación Lifecycle o SmartLifecycle. Cuando el contexto de la aplicación inicializa un SmartLifecycle puede iniciarse automáticamente si su bandera autoStartup está establecida, y un Lifecycle puede iniciar manualmente llamando a ConfigurableApplicationContext.start() en el contexto adjuntando.

·         Utilice un Spring ApplicationEvent o un mecanismo observador similar personalizado para activar la inicialización del caché. ContextRefreshedEvent siempre se publica por el contexto cuando está listo para su uso (después de que todos los beans se hayan inicializado), por lo que a menudo es un gancho útil (así es como el SmartLifecycle trabaja por defecto).

La segunda opción también puede ser fácil. Algunas sugerencias sobre cómo implementar este son:

·         Confíe en el comportamiento por defecto de Spring BeanFactory, que es que los beans son inicializados en el orden de registración. Usted puede ponerse de acuerdo fácilmente adoptando la práctica común de un conjunto de elementos <import/> que ordenan sus módulos de aplicación, y aseguran que la base de datos y la inicialización de la base de datos figuran en primer lugar.

·         Separe el datasource y los componentes de negocio que utilizan y controlan su orden de inicio, poniéndolos en diferentes instancias ApplicationContext (por ejemplo, el padre tiene el datasource y el hijo tiene los componentes de negocio). Esta estructura es común en aplicaciones web de Spring, pero pueden aplicarse de manera más general.

·         Utilice un modular en tiempo de ejecución, como el servidor de SpringSource dm y separe el datasource y los componentes que dependen de ello. Por ejemplo, especifique el paquete para la puesta en marcha, como datasource -> inicializador -> componentes de negocio.