Licenciatura en Sistemas - Bases de Datos II – 2018

Titular: Ing. Federico Ribeiro federico.ribeiro@gmail.com
Ayudante: APU Leandro Ríos leandro.rios.unla@gmail.com

Apuntes Hibernate ORM

Manual Hibernate: http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html/
Hibernate API: https://docs.jboss.org/hibernate/orm/4.3/javadocs/overview-summary.html

Estrategias de mapeo de herencia en Hibernate

Hibernate ofrece tres estrategias distintas para mapear una relación de herencia entre objetos en un modelo relacional. Éstas son Tabla por jerarquía de clases, tabla por clase y tabla por clase concreta.

Tabla por jerarquía de clases

En este caso, se utiliza y mapea una sola tabla representando la jerarquía de herencia completa. Vamos a hacer un ejemplo con una jerarquía de pagos: supongamos que tenemos un sistema de registro de pagos que acepta pagos en efectivo, con tarjeta o con cheque. A fin de modelarlo, establecemos una clase Pago con subclases PagoEfectivo, PagoTarjeta y PagoCheque. Pago es una clase abstracta, ya que no tiene sentido tener una instancia de pago sin especificar de qué tipo de pago se trata.

```
<class name="Pago" table="pago">
    <id name="idPago" type="long" column="idPago">
        <generator class="native"/>
    </id>
    <discriminator column="tipo_pago" type="string"/>
    <property name="monto" column="monto"/>
    ...
    <subclass name="PagoTarjeta" discriminator-value="TARJETA">
        <property name="numeroTarjeta" column="numero_tarjeta".../>
        ...
    </subclass>
    <subclass name="PagoEfectivo" discriminator-value="EFECTIVO">
        ...
    </subclass>
    <subclass name="PagoCheque" discriminator-value="CHEQUE">
        ...
```

```
    </subclass>
</class>
```

Tabla por clase

En este caso se emplea una tabla por cada subclase definida y una más para la superclase. Existe una relación uno a uno entre la tabla de la superclase y las tablas que representan a las subclases, debido a que la clave primaria de éstas últimas es también una clave foránea obtenida de la superclase. Esto implica que cada vez que traigamos objetos del tipo de alguna de las subclases, la base de datos efectuará un join para poder acceder a todos los atributos del objeto (definidos en la superclase y la subclase).

```
<class name="Pago" table="pago">
    <id name="idPago" type="long" column="idPago">
        <generator class="native"/>
    </id>
    <property name="monto" column="monto"/>
    ...
    <joined-subclass name="PagoTarjeta" table="pago_tarjeta">
        <key column="idPago"/>
        <property name="numeroTarjeta" column="numero_tarjeta".../>
        ...
    </joined-subclass>
    <joined-subclass name="PagoEfectivo" table="pago_efectivo">
        <key column="idPago"/>
        ...
    </joined-subclass>
    <joined-subclass name="PagoCheque" table="pago_cheque">
        <key column="idPago"/>
        ...
    </joined-subclass>
</class>
```

En caso de querer traer todas las instancias de la clase Pago, Hibernate emitirá el SQL necesario para que el motor de base de datos realice un inner join entre las tablas que representan a las subclases y la que representa a la superclase.

Tabla por clase concreta

En este caso se emplea una tabla por cada clase concreta y los atributos de la superclase se repiten en cada tabla. Esto implica que la clave primaria ya no podrá ser un campo autonumérico, ya que la clave debe compartirse entre las tablas de cada clase concreta.

```
<class name="Pago">
    <id name="id" type="long" column="idPago">
        <generator class="sequence"/>
    </id>
    <property name="monto" column="monto"/>
    ...
    <union-subclass name="PagoTarjeta" table="pago_tarjeta">
        <property name="numeroTarjeta" column="numero_tarjeta".../>

        ...
    </union-subclass>
    <union-subclass name="PagoEfectivo" table="pago_efectivo">
        ...
    </union-subclass>
    <union-subclass name="PagoCheque" table="pago_cheque">
        ...
    </union-subclass>
</class>
```

En caso de querer traer todas las instancias de la clase Pago, Hibernate emitirá el SQL necesario para que el motor de base de datos realice una unión entre las tablas que representan a las subclases.

Para todos los casos, los mapeos de clases y subclases pueden existir dentro del mismo archivo xml como venimos haciendo, o cada clase puede tener su propio archivo. De ser así, los mapeos correspondientes a las subclases deben incluir la palabra reservada extends y el nombre de la superclase de la clase que está siendo mapeada. Ejemplo:

```
<joined-subclass name="PagoTarjeta" table="pago_tarjeta" extends="Pago">
```

Cómo crear el esquema desde Hibernate

Para que Hibernate cree el esquema de nuestra base de datos y defina las tablas y sus relaciones, tenemos dos estrategias: Por medio del archivo de configuración (hibernate.cfg.xml) o programáticamente.

Por medio del archivo de configuración: se debe modificar el mapeo de modo que quede como sigue:

```
<hibernate-configuration>
    <session-factory>
       <property name="connection.driver_class">
            com.mysql.jdbc.Driver
       </property>
       <property name="connection.url">
            jdbc:mysql://localhost/nombrebd?createDatabaseIfNotExist=true
```

```
        </property>
        <property name="connection.username">aaaa</property>
        <property name="connection.password">aaaa</property>
        <property name="connection.pool_size">1</property>
        <property me="dialect">
             org.hibernate.dialect.MySQLDialect
        </property>
        <property name="show_sql">false</property>
        <!-- valores posibles: create create-drop validate update -->
        <property name="hibernate.hbm2ddl.auto">create-drop</property>

        <!--Mapeo Entidades -->
        ...
    </session-factory>
</hibernate-configuration>
```

La primera línea resaltada indica que debe crearse el esquema si éste no existe. La segunda que deben eliminarse y recrearse las tablas del mismo. Existen otras posibilidades, indicadas en el comentario que antecede a la línea.
Para realizarlo de manera programática, se deberá ejecutar el siguiente código:

```
// para recrear la bd
Configuration configuration = new Configuration().configure();
ServiceRegistry serviceRegistry = new
ServiceRegistryBuilder().applySettings(configuration.getProperties()).buil
dServiceRegistry();
SchemaExport schemaExport = new SchemaExport(serviceRegistry,
configuration);
schemaExport.create(Target.EXPORT); // SCRIPT o BOTH
```

Estados de los objetos en Hibernate

Los objetos de datos, con respecto a su manejo por Hibernate, pueden encontrarse en uno de tres estados: Transient, Persistent o Detached.

Los objetos se encuentran en estado **Transient** (transitorio) cuando están recién creados con new y no se encuentran asociados a una sesión. Tampoco tiene una representación en la base de datos.

En el estado **Persistent** (persistente o persistido), los objetos se han asociado a una sesión con session.save() o session.persist() y ya cuentan con una representación en la base de datos. Los objetos permanecen en este estado mientras la sesión se encuentre abierta, y todas las modificaciones que hagamos sobre los mismos se reflejarán en la base de datos al realizar session.flush(). La diferencia entre los métodos save y persist es que el primero realiza el insert inmediatamente y devuelve un identificador (se asegura de que el mismo se

haya generado, por eso el insert), mientras que el segundo no garantiza que se genere el identificador y puede demorar el insert hasta la llamada a flush().
Cuando se cierra la sesión, los objetos quedan en estado **Detached** (desconectado), es decir, disponemos de los mismos pero no existe conexión con la base de datos. Podemos manipularlos y modificarlos, pero estos cambios no se verán reflejados en la base de datos. Para reflejarlos, debemos reconectar los objetos con una nueva sesión y actualizarlos mediante los métodos update() o merge().
Update intenta actualizar la base de datos con los datos del objeto desconectado y si no tiene éxito (porque se ha cargado un objeto con el mismo id desde la base de datos en la nueva sesión, por ejemplo) levantará una excepción. Merge, en cambio, actualizará el objeto persistente que ya existe con los atributos del objeto que recibe como parámetro.
Para no tener que decidir entre utilizar save, update , o merge, Hibernate ofrece el método saveOrUpdate, que procede al salvado de un objeto en estado transient o a la reconexión del mismo si se encuentra en estado detached.

Ejemplos de consultas en Hibernate HQL (Capítulo 16 del manual)

Hibernate ofrece un lenguaje para elaborar consultas denominado HQL (Hibernate Query Language, lenguaje de consultas de Hibernate). Si bien tiene una sintaxis similar a la de SQL, se diferencia de éste en que no opera sobre entidades de nuestro modelo relacional, sino que lo hace sobre nuestra jerarquía de objetos. Esto implica que las consultas deben pensarse y componerse desde el modelo de objetos.
Con el fin de representar y ejecutar consultas, Hibernate nos ofrece una clase Query, de la que podemos pedir instancias sobre las que trabajar a nuestra sesión. Por ejemplo:

```
List personas = session.createQuery(
    "from Persona as p where p.fechaNacimiento < ?")
    .setDate(0, fecha)
    .list();
```

Esta consulta nos trae todas la instancias de Persona cuya fecha de nacimiento sea menor que fecha y las deja en una lista. Persona, en este caso, es el nombre de la clase a la que pertenecen los objetos a traer; en las consultas HQL siempre hacemos referencia al modelo de objetos.
Vemos aquí también uno de los modos de pasar parámetros a la consulta: el parámetro fecha se pasa con el método setDate, que nos evita tener que formatear fechas como lo haríamos si armáramos el query concatenando strings. El 0 que pasamos como primer parámetro del método setDate indica la posición del parámetro en el query. Otra forma de hacer esto es utilizando parámetros con nombre (named parameters), por ejemplo:

```
List madres = session.createQuery(
    "select madre from Persona as p join p.madre as m where p.apellido =
:apellido")
    .setString(“apellido”, "Fernandez")
    .list();
```

Query tiene otro método denominado setParameter(nombre, valor), que usualmente es lo suficientemente inteligente como para determinar el tipo del parámetro (lo usaríamos en lugar de setDate, setString, etc).

```
List madres = session.createQuery(
    "select madre from Persona as p join p.madre as m where p.apellido =
:apellido")
    .setParameter(“apellido”, "Fernandez")
    .list();
```

Los parámetros con nombre tienen la ventaja de que el query queda más legible y no nos importa el orden en que se declaran.
En este caso estamos realizando un (inner) join entre Persona (los hijos) y Persona (sus madres), y traemos sólo las madres de las Personas con apellido Fernández. La lista madres, por supuesto, contendrá instancias de persona.

Podemos usar múltiples joins: para traer a todas las Personas con sus parejas y sus hijos:

```
List personas = session.createQuery(
      "from Persona as p
      inner join p.pareja as pareja
      left join p.hijos as hijos").list();
```

El alias de pareja e hijos no es estrictamente necesario. Sí lo será si debemos referirnos a los objetos que los representan:

```
List personas = session.createQuery(
      "from Persona as p
      inner join fetch p.pareja
      left join fetch p.hijos hijos
      left join fetch hijos.hijos").list();
```

Debido a que trabajamos con lazy loading (las dependencias de los objetos no se cargan hasta que son necesarias) Hibernate por defecto no cargará los objetos contenidos ni las colecciones. Para forzar la carga, utilizamos la palabra reservada “fetch” en los joins. Esto provocará que las colecciones y los objetos contenidos se inicialicen.

Si queremos traer todos los hijos de una instancia de Persona (Persona madre):

```
List hijos = session.createQuery(
    "from Persona as p where p.madre = ?")
    .setEntity(0, madre)
    .list();
```

Si queremos traer la madre de una instancia de Persona determinada (Persona hijo):

```
Persona madre = (Persona) session.createQuery(
    "select p.madre from Persona as p where p = ?")
    .setEntity(0, hijo)
    .uniqueResult();
```

Los joins que hemos realizado hasta ahora son explícitos. Hay otra sintaxis que nos permite realizar lo que se denomina joins implícitos:

```
Persona persona = (Persona) session.createQuery(
      "from Persona p where p.pareja.nombre = :nombre")
      .setString("nombre", "Adrián").uniqueResult();
```

Las consultas en Hibernate son polimórficas: Si pedimos todos los objetos de pertencen a la superclase (`from pago p where p.fecha=...`) traerá las instancias de la superclase que existan si ésta es concreta y las instancias de las subclases también.