Programación

Bloque 09 - Acceso a bases de datos orientadas a objetos

Índice

1.- Introducción....2
2.- Las bases de datos orientadas a objetos....3
3.- Una base de datos orientada a objetos (db4o)....4
4.- Conexión con una base de datos....5
4.1.- Cierre de conexiones....5
5.- Operaciones....6
5.1.- Almacenamiento....6
5.2.- Eliminación....6
6.- Consultas....8
6.1.- Proceso de los resultados....8
6.2.- Consultas Query By Class....9
6.3.- Consultas Query By Example....10
6.4.- Consultas Native Queries....11
6.4.1.- Ordenación de los resultados....13
7.- Referencias....15

1.- Introducción

En bloques anteriores hemos aprendido a emplear ficheros y bases de datos relaionales para persistir datos.

El empleo de ficheros exige que el programador realice muchas tareas de bajo nivel como controlar el formato de los datos, etc.

El uso de bases de datos relacionales mejora todo esto ofreciendo un interfaz relativamente más sencillo para almacenar y consultar datos empleando la potencia del lenguaje SQL. Sin embargo esta técnica también plantea algunos problemas:

- El programador sigue siendo responsable de "convertir" un objeto a algo que pueda almacenarse en una tabla, tanto a la hora de almacenar información como a la hora de recuperarla desde la base de datos. Por ejemplo, un objeto de clase `Usuario` que contiene un nombre y edad (entera) debe almacenarse en una tabla con campos que pueden tener otros nombres y debe saber en que columna están esos datos.
- Las relaciones entre objetos son problemáticas y no tienen un soporte directo. Si un objeto contiene en uno de sus atributos otro objeto, las mayoría de bases de datos relacionales exigen que los datos de las distintas clases estén almacenados en distintas tablas y que se relacionen por clave (primaria y foránea). Esta conversión debe hacerla el programador de forma explícita.

Estos dos problemas, junto con alguno más, forman lo que se conoce como la impedancia objeto-relacional. Esta es una barrera conceptual entre las dos formas de almacenar información que tienen por un lado los objetos y por otro las tablas del modelo relacional.

Para atravesar esta barrera se han desarrollado a lo largo del tiempo una buena cantidad de librerías y frameworks que intentan simplificar esta tarea para el programador pero no dejan de ser un parche que exige en muchos casos tunear la solución implementada o especificar una gran cantidad de opciones o configuración para que la conversión se haga de forma lógica.

Otra solución completamente distinta ha sido la aparición e implementación de bases de datos orientadas a objetos, que es el objeto de este tema

2.- Las bases de datos orientadas a objetos

Las bases de datos orientadas a objetos son sistemas que permiten el almacenamiento persistente de objetos de forma directa, sin necesidad de realizar "adaptaciones" como sería el caso la emplear ficheros o bases de datos relacionales.

En una base de datos un objeto se almacena simplemente indicándolo y la base de datos es capaz de determinar la información que contiene el objeto, incluyendo los posibles objetos que pueda contener como atributos y almacenar esta información directamente en almacenamiento persistente, recuperándola más tarde.

3.- Una base de datos orientada a objetos (db4o)

db4o (Data Base for Objects) es una base de datos orientada a objetos diseñada para trabajar con Java (y también .NET).

Permite funcionar tanto en modo remoto (cliente-servidor) como en modo embebido. En este último caso la librería almacena los datos en el sistema de ficheros local y se puede mover la información junto con la aplicación simplemente copiando estos ficheros. Este es el modo ideal para aplicaciones que no requieren acceso concurrente a la misma información desde varias aplicaciones y el que vamos a emplear en este bloque.

La instalación, en este caso, es entonces bastante sencilla. Para comenzar a emplear db4o simplemente debemos incluir la librería adecuada en nuestro projecto (y asegurarnos de que la distribuimos con la aplicación).

En nuestro caso la librería es la db4o.jar. Una vez la tengamos incorporada ya podemos comenzar a emplearla.

4.- Conexión con una base de datos

db4o emplea un único archivo para una base de datos de forma que se pueden almacenar múltiples objetos. Si queremos emplear otra base de datos simplemente deberemos emplear otro archivo.

La conexión siempre se hace de la misma manera, empleando:

```
import com.db4o.Db4o;
import com.db4o.ext.Db4oException;
......
try {
  ObjectContainer db = Db4o.openFile(RUTA_ARCHIVO);
} catch (Db4oException e) {
  // Procesa error al acceder al archivo
}
```

Se puede con atributo

donde `RUTA_ARCHIVO` es la ruta al archivo de base de datos a emplear. Por constumbre suele tener la extensión `.db` o bien `.db4o`

Si el archivo en la ruta especificada no existe, db4o crea un nuevo archivo de base de datos empleando la ruta. Si el archivo existe, db4o accede a la base de datos.

En caso de que el archivo exista y haya algun problema con el mismo se lanza la excepción `Db4oException`

4.1.- Cierre de conexiones

A fin de conservar recursos y evitar problemas de funcionamiento por agotamiento de los mismos es necesario el cerrar una conexión cuando ya no se necesite más.

Esto se debe hacer de forma manual empleando el método `close()` de la clase `ObjectContainer` ya que esta clase **NO IMPLEMENTA** el interfaz `AutoCloseable` y no se puede emplear por tanto en un try.. catch con recursos.

Se abre y se cierra
O un método que cierre cada cosa.

5.- Operaciones

Una vez que tenemos establecida una conexión, la interacción con el SGBDR se realiza mediante *operaciones*. Al no existir un lenguaje estandarizado de manipulación y consulta como SQL las operaciones se realizan directamente mediante mensajes, en este caso sobre objetos de la clase `ObjectContainer`.

5.1.- Almacenamiento

En db4o sólo existe una operación de almacenamiento en lugar de dos operaciones separadas (inserción y actualización - `INSERT` / `UPDATE`).

Para almacenar un objeto en la base de datos se emplea el método `store` de la forma

```
ObjectContainer db;
..... Adquiere la conexión
......Se hacen otras tareas
Persona persona = new Persona("Juan Gomez", 23);
// Almacena la persona
try {
  db.store(persona);
} catch (Db4oException e) {
  // Error almacenando el objeto
}
```

Al hacer store db4o almacena el objeto en la base de datos, si no existía previamente, o lo actualiza, si ya existía. Asimismo actualiza otros objetos que puedan ser atributos del objeto que se almacena, así como atributos de estos, etc.

5.2.- Eliminación

La eliminación se realiza mediante el método `delete` de la forma:

```
ObjectContainer db;
..... Adquiere la conexión
......Se hacen otras tareas
Persona persona = new Persona("Juan Gomez", 23);
db.store(persona);
.... Mas tareas
db.delete(persona);
```

`delete` recibe el objeto a eliminar de la base de datos y procede a la eliminación.

Un detalle importante es que los objetos que sean atributos del objeto eliminado **no se eliminan de la base de datos sino que permanecen**. Esto es importante porque si se quiere eliminar un objeto y aquellos que dependen de él hay que hacer los borrados desde dentro hacia afuera, esto es, desde los objetos miembros de otros objetos hacia los objetos que los contienen. De no hacerse asi permanecerían objetos aislados almacenados en la base de datos.

Otra observación importante es que `delete` elimina objetos aislados. Si se quieren eliminar varios objetos hay que hacer repetidas llamadas a `delete`, una por cada objeto a eliminar.

6.- Consultas

db4o ofrece varias formas de consultar la información almacenada en la base de datos. En concreto proporciona las siguientes:

Consulta usando ejemplo, seria un filtro muy basico, es solo con unos campos y si estos tienen un valor Es para buscar valores exactos

Todos los objetos sin filtro

- Query by Class (QBC). Usando este método se recuperan todos los objetos almacenados pertenecientes a una clase dada.
- Query by Example (QBE). Usando este método se proporciona a la base de datos un "prototipo" del objeto u objetos que se quieren recuperar y la base de datos devuelve todos los objetos almacenados que "cumplen" con el prototipo.
- Native Queries. Usando este método debemos proporcionar un "filtro" en Java. La base de datos obtiene todos los objetos y los va pasando uno a uno al filtro que determina si el objeto entra o no en el resultado. Dado que se utiliza nativamente el mismo lenguaje (Java en nuestro caso) se llaman consultas nativas

Se puede especificar de forma general un filtro, empleando condiciones de java

- Consultas SODA. Son consultas que emplean las mismas librerías de db4o para localizar los objetos. No vamos a tratar estas consultas pues pueden ser bastante complejas y ya tenemos métodos suficientes para todas las posibles situaciones que se nos pruedan presentar.

En las siguientes secciones veremos los distintos métodos, la forma de emplearlos y sus ventajas e inconvenientes.

6.1.- Proceso de los resultados

Sea cual sea el tipo de consulta db4o devuelve un objeto que implementa el interfaz `ObjectSet<T>`, donde `T` es la clase de los objetos que estamos consultando (que tienen que ser todos de la misma clase). `ObjectSet` ofrece métodos para navegar por el resultado y obtener los objetos que componen el mismo.

Estos métodos son:

- `int size()`. Devuelve el número de elementos en el resultado. Puede ser cero si la consulta no obtuvo ningún elemento.
- `boolean hasNext()`. Devuelve true si quedan aún objetos por procesar en el resultado. false si ya no quedan objetos sin procesar.
- `T next()`. Devuelve el siguiente objeto a procesar
- `void reset()`. Reinicia el proceso de los objetos al inicio del conjunto (permite volver a procesar los resultados sin necesidad de hacer otra consulta).

Una forma de procesar los resultados podría ser:

```
ObjectSet<Persona>personas;
.... Hace una consulta y guarda el resultado en personas
// Mientras haya resultados
```

```
while (personas.hasNext()) {
  // Obtiene la persona
  Persona persona = personas.next();
  // Procesa la persona
  ......
}
```

Se emplea un ciclo `while` porque existe la posibilidad que la consulta no devuelva ninguna Persona por lo que no hay que hacer ninguna iteración.

Al inicio del bucle se comprueba `hasNext()` para determinar si hay o no más personas a procesar. Si las hay (devuelve true) se accede a la persona que toca (mediante `next()`) y se procesa.

No es necesario "cerrar" el resultado ya que la información se descartará automáticamente.

Aunque este método es sencillo, existen formas alternativas de procesar los resultados. De ellas, la más sencilla consiste en explotar el hecho de que `ObjectSet` implementa el interfaz Iterable. Por lo tanto se puede emplear en un ciclo tipo forEach de la forma:

```
ObjectSet<Persona>personas;
.... Hace una consulta y guarda el resultado en personas
// Para cada persona en el resultado
for (Persona persona: personas) {
  // Procesa la persona (en la variable persona)
  ......
}
```

Este método es ciertamente más simple y menos propenso a errores por lo que es el recomendado.

6.2.- Consultas Query By Class

Las consultas tipo Query By Class (Consulta por Clase) permiten consultar TODOS los objetos de una clase determinada que estén almacenados en la base de datos. Para ello se emplea el método `query` de `ObjectContainer` con la forma:

Introspección java.

```
ObjectSet<C> query(Class<C> clazz);
```

Este método recibe como parámetro la *clase* de los objetos que queremos consultar en la base de datos y responde con un `ObjectSet` con *todos* los objetos de dicha clase que se encuentren en la base de datos (si hay alguno).

Ejemplo:

```
ObjectContainer db = null;
try {
  // Accedemos a la base de datos
  db = Db4o.openFile("personas.db");

  // Consultamos a todas las personas (empleamos Persona.class para acceder
a la clase de Persona)
  ObjectSet<Persona> personas = db.query(Persona.class);

  // Para cada persona
  for(Persona persona: personas) {
```

```
    // Imprime la persona
    System.out.println(persona);
  }
} catch (Exception e) {
  // En caso de excepcion muestra un mensaje por la salida de error
  System.err.println("Error accediendo a la base de datos");
} finally {
  // En cualquier caso hay que cerrar la conexión si está abierta
  // Se intenta cerrar y se ignora cualquier excepción que se pudiera
producir
  try {
    db.close();
  } catch (Exception e) {}
}
```

Este método es apropiado cuando lo que se desea es acceder a TODOS los objetos de la misma clase almacenados en la base de datos.

Un detalle MUY importante es que al consultar objetos de una clase determinada se obtiene los objetos pertenecientes a dicha clase y todos los objetos de clases que hereden de ella.

Por ejemplo, supongamos que tenemos las clases `Persona`, `Alumno` y `Profesor`. Tanto `Alumno` como `Profesor` heredan de `Persona`. Si consultamos empleando `Persona.class` obtendremos tanto las `Persona`s que no son ni `Alumno` ni `Profesor` como todos los `Alumno`s (que también son `Persona`) y los `Profesor`es.

Si consultamos empleando `Profesor` o `Alumno` sólo obtendremos objetos de dichas clases (suponiendo que no hay otras que heredan de éstas) pero no `Persona`.

probar crear un visitador con esto. Deberia ser un objeto que implemente esto

6.3.- Consultas Query By Example

Las consultas Query By Example (Consulta Mediante Ejemplo) es un tipo de consulta en el cual se proporciona a db4o un "prototipo" de lo que estamos buscando y el motor nos contesta con los objetos que cumplen el prototipo.

La cuestión importante es: ¿Qué se considera como un prototipo?

Para db4o un prototipo es un objeto cualquiera. Cuando se emplea un objeto como prototipo, db4o intenta localizar objetos en la base de datos que:

- Son de la misma clase que el prototipo o su clase hereda de aquella, es decir, busca objetos de la misma clase que el prototipo o subclases.
- Los objetos tienen los mismos valores (según `equals`) para aquellos campos que en el prototipo tienen valores distintos a los valores por defecto en Java (0 para los numéricos, `false` para los booleanos, caracter `\0` para los caracteres y `null` para los objetos.

Por ejemplo, supongamos el objeto:

```
Persona persona = new Persona(null, 23);
```

Si lo empleamos como prototipo obtendríamos todas las personas cuya edad sea 23.

Esto es asi porque el objeto tiene dos campos (`nombre`, `edad`) de los cuales uno (`nombre`) tiene el valor por defecto (`null`) y el otro no (23). Por lo tanto `nombre` no es emplea y sólo se emplea la `edad` y el valor 23.

Si empleáramos en cambio:

```
Persona persona = new Persona("Paco", 0);
```

La búsqueda se realizaría por nombre (que debe ser exactamente igual a "Paco") y no por edad,ya que ésta vale 0.

Para realizar la búsqueda se emplea el método queryByExample(), al cual se le proporciona el objeto plantilla y devuelve un objeto ObjectSet con los objetos que cumplen con el prototipo.

Ejemplo:

```
ObjectContainer db = null;
try {
  // Accedemos a la base de datos
  db = Db4o.openFile("personas.db");

  // Consultamos a todas las personas de 20 años
  Persona personaProto = new Persona(null, 20);
  ObjectSet<Persona> personas = db.queryByExample(personaProto);

  // Para cada persona
  for(Persona persona: personas) {
    // Imprime la persona
    System.out.println(persona);
  }
} catch (Exception e) {
  // En caso de excepcion muestra un mensaje por la salida de error
  System.err.println("Error accediendo a la base de datos");
} finally {
  // En cualquier caso hay que cerrar la conexión si está abierta
  // Se intenta cerrar y se ignora cualquier excepción que se pudiera
producir
  try {
    db.close();
  } catch (Exception e) {}
}
```

6.4.- Consultas Native Queries

Las consultas Native Queries (Consultas Nativas) emplean el mismo lenguaje de programación (en nuestro caso Java) para determinar si un objeto debe formar o no parte del resultado.

Para ello las expresiones empleadas deben devolver true si un objeto debe formar parte del resultado o false si no. db4o intentará optimizar las consultas nativas de forma que se empleen indices y sean lo más rápidas y eficientes que sea posible.

Por lo tanto, a la hora de hacer una consulta la primera cuestión a resolver es: ¿Qué criterio debe serguir un objeto para entrar en el resultado de la consulta? Esto depende fuertemente de lo que se esté haciendo y para qué se quiere el resultado.

Siguiendo nuestro ejemplo, supongamos que queremos obtener todas las Personas que son mayores de edad, es decir, cuya edad sea igual o superior a 18. Este tipo de consultas no lo podíamos hacer con los mecanismos vistos hasta ahora pero si lo podemos hacer con consultas nativas.

Una vez determinado el criterio, la siguiente cuesión es: ¿Cómo se implementa este criterio?

La respuesta es sencilla: Hay que crear una nueva clase, por ejemplo `CriterioMayorDeEdad`, que herede de la clase base `com.db4o.query.Predicate`

Esta clase es abstracta y es genérica con un parámetro de tipo `ExtentType` que sería el tipo de los objetos que se quieren tratar en la consulta. Por lo tanto, en nuestro caso haríamos:

```
class CriterioMayorDeEdad extends Predicate<Persona> {
.....
}
```

Como puedes ver se parametriza la clase `Predicate` empleando la clase concreta de objetos que vamos a filtrar, en nuestro caso `Persona`. Si el filtro fuera para objetos de otra clase se debería emplear esa clase.

Para hacer la consulta crearemos una instancia de este criterio y se usaría en la consulta. Ésta se realiza con el método query y se pasa el criterio de la forma:

```
CriterioMayorDeEdad criterio = new CriterioMayorDeEdad();
ObjectSet<Persona> personas = db.query(criterio);
```

Si has seguido la discusión hasta ahora seguro que te estarás preguntando: ¿Y donde está el criterio? Esto es, ¿en qué sitio se indica que la edad de la persona debe ser mayor o igual a 18?

La respuesta simple es que esto no lo hemos hecho... aun.

El problema es que la clase `CriterioMayorDeEdad` no está vacía como se ve en el ejemplo anterior sino que debe implementar un método, llamado `match()` de la forma:

```
boolean match(ExtentType candidato);
```

Como se hereda de una clase genérica (Predicate) hay que sustituir el tipo ExtentType por un tipo real, en nuestro caso Persona. Por lo tanto, en nuestro ejemplo el método sería:

```
boolean match(Persona candidato);
```

Mejor pero aún nos falta dar un cuerpo al método match(). Dentro de este método se debe analizar cada Persona y devolver true si consideramos que una Persona debe entrar en el resultado o false si consideramos que NO debe entrar. En nuestro caso concreto lo que nosotros queremos son las personas mayores de edad (con 18 o más años). Por lo tanto, nuestro método match quedaría (vamos a hacer la clase CriterioMayorDeEdad completa):

```
class CriterioMayorDeEdad extends Predicate<Persona> {
  @Override
  public boolean match(Persona candidato) {
```

```
    return candidato.getEdad() >= 18;
}
```

Todo esto se puede hacer de forma más compacta empleando clases anónimas de la forma:

```
ObjectSet<Persona> personas = db.query(new Predicate<Persona>() {
  @Override
  public boolean match(Persona candidato) {
    return candidato.getEdad() >= 18;
  }
});
```

En este caso indicamos el criterio directamente en lugar de crear una nueva clase. Este es el método recomendado a no ser que se desee reutilizar el mismo criterio en varios sitios de la aplicación, lo cual no es muy común.

6.4.1.- Ordenación de los resultados

El objeto `ObjectSet` con los resultados de una consulta no mantiene un orden definido entre los objetos que contiene. La razón de esto es evidente:

- Los objetos se almacenan en la base de datos empleando métodos opacos para el programador por lo que no podemos preveer el orden en que se obtienen desde almacenamiento.
- Excepto en los casos básicos y triviales no se puede suponer cual es el orden entre objetos de una clase ya que el orden es un concepto arbitrario y más en el caso de que haya varios atributos

Por lo tanto, si queremos que los resultados de una consulta estén ordenados debemos proporcionar un criterio de ordenación, de la misma forma que proporcionamos un criterio de selección (empleando `Predicate`). En este caso hay que proporcionar un objeto de una clase que implemente el interfaz `java.util.Comparator<T>`.

Este es un interfaz funcional (sólo define un método) y genérico por el tipo `T` que especifica la clase de los objetos a comparar. El único método que se debe definir es:

```
int compare(T o1, T o2)
```

Este método define el criterio de ordenación que queremos usar al comparar dos objetos del tipo que sea (el genérico `T`) y devolver un entero negativo si el primer objeto debe ir antes que el segundo, 0 si los dos objetos son "iguales", esto es, deberían ocupar la misma posición y no podemos decidir, y positivo si el segundo debe ir antes del primero.

Siguiendo con nuestro ejemplo, supongamos que queremos ordenar las `Persona`s por edad en orden inverso (las mayores primero).

Esto lo podemos hacer, al igual que con `Predicate` creando una nueva clase:

```
public class OrdenaPorEdadMayoresAntes implements Comparator<Persona> {

  @Override
  public int compare(Persona o1, Persona o2) {
```

```
    return o2.getEdad() - o1.getEdad();
  }

}
```

Y luego haciendo lo siguiente a la hora de realizar la consulta

```
CriterioMayorDeEdad criterio = new CriterioMayorDeEdad();
OrdenaPorEdadMayoresAntes comparador = new OrdenaPorEdadMayoresAntes();
ObjectSet<Persona> personas = db.query(criterio, comparador);
```

En este caso pasamos a `query` dos parámetros. El primero es el criterio de selección y el segundo es el comparador.

También podemos emplear clases anónimas, como en el caso de emplear sólo un selector:

```
      ObjectSet<Persona> personas = db.query(new Predicate<Persona>() {
        @Override
        public boolean match(Persona candidato) {
          return candidato.getEdad() >= 18;
        }
      }, new Comparator<Persona>() {
        @Override
        public int compare(Persona o1, Persona o2) {
          return o2.getEdad() - o1.getEdad();
        }
      });
```

se puede emplear landas.
(o1, o2) -> resultado.

Es como una coleccion
Ver si usa metodos de coleccion

7.- Referencias