Transacciones

Índice

1. Transacciones

La gestión de transacciones es fundamental en cualquier aplicación que trabaja con de datos. Idealmente, una transacción define una *unidad de trabajo* que debe cumplir los criterios ACID: Atomicidad, Consistencia, Aislamiento (la "I" viene del inglés *Isolation*) y Durabilidad. La consistencia la debe proporcionar el programador, realizando operaciones que lleven los datos de un estado consistente a otro. La durabilidad la proporciona el hecho de que usemos un sistema de bases de datos. Las dos características restantes, la *atomicidad* y el *aislamiento* son las más interesantes, y las que vienen determinadas por el sistema de gestión de persistencia que usemos. Veámoslas con más detalle.

Una transacción debe ser atómica. Esto es, todas las operaciones de una transacción deben terminar con exito o la transacción debe abortar completamente, dejando el sistema en el mismo estado que antes de comenzar la transacción. En el primer caso se dice que la transacción ha realizado un commit y en el segundo que ha efectuado un rollback. Por esta propiedad, una transacción se debe tratar como una unidad de computación.

Para garantizar la atomicidad en JDBC se llama al método `setAutoCommit(false)` de la conexión para demarcar el comienzo de la transacción y a `commit()` o `rollback()` al final de la transacción para confirmar los cambios o deshacerlos. Veremos que JPA que utiliza una demarcación explícita de las transacciones, marcando su comienzo con una llamada a un método `begin()` y su final también con llamadas a `commit()` o `rollback()`. Siempre debemos tener muy presente que JPA (cuando no utilizamos un servidor de aplicaciones) se basará completamente en la implementación de JDBC para tratar con la base de datos.

Un transacción también debe ejecutarse de forma aislada. Esto es, no se deben exponer a otras transacciones concurrentes datos que todavía no se han consolidado con un commit. La concurrencia de acceso a los recursos afectados en una transacción hace muy complicado mantener esta propiedad. Veremos que JPA proporciona un enfoque moderno basado en *bloqueos optimistas* (*optimistic locking*) para tratar la concurrencia entre transacciones.

1.1. Atomicidad

JPA define la interfaz `EntityTransaction` para gestionar las transacciones. Esta interfaz intenta imitar el API de Java para gestión de transacciones distribuidas: JTA. Pero tengamos siempre en cuenta que para su implementación se utiliza el propio sistema de transacciones de la base de datos sobre la que trabaja JPA, utilizando los métodos de transacciones de la interfaz `Connection` de JDBC. Estas transacciones son locales (no distribuidas) y no se pueden anidar ni extender.

Para definir una transacción hay que obtener un `EntityTransaction` a partir del entity

manager. El método del entity manager que se utiliza para ello es `getTransaction()`. Una vez obtenida la transacción, podemos utilizar uno de los métodos de su interfaz:

```
public interface EntityTransaction {
   public void begin();
   public void commit();
   public void rollback();
   public void setRollbackOnly();
   public boolean getRollbackOnly();
   public boolean isActive();}
```

Sólo hay seis métodos en la interfaz `EntityTransaction`. El método `begin()` comienza una nueva transacción en el recurso. Si la transacción está activa, `isActive()` devolverá `true`. Si se intenta comenzar una nueva transacción cuando ya hay una activa se genera una excepción `IllegalStateException`. Una vez activa, la transacción puede finalizarse invocando a `commit()` o deshacerse invocando a `rollback()`. Ambas operaciones fallan con una `IllegalStateException` si no hay ninguna transacción activa. El método `setRollbackOnly()` marca una transacción para que su único final posible sea un `rollback()`.

Se lanzará una `PersistenceException` si ocurre un error durante el rollback y se lanzará una `RollbackException` (un subtipo de `PersistenceException`) si falla el commit. Tanto `PersistenceException` como `IllegalException` son excepciones de tipo `RuntimeException`.

El método `setRollbackOnly()` se utiliza para marcar la transacción actual como inválida y obligar a realizar un rollback cuando se ejecuta JPA con transacciones gestionadas por el contenedor de EJB (CMT: *Containter Managed Transactions*). En ese caso la aplicación no define explícitamente las transacciones, sino que es el propio componente EJB el que abre y cierra una transacción en cada método.

Hemos comentado que en JPA todas las excepciones son de tipo `RunTimeException`. Esto es debido a que son fatales y casi nunca se puede hacer nada para recuperarlas. En muchas ocasiones ni siquiera se capturan en el fragmento de código en el que se originan, sino en el único lugar de la aplicación en el que se capturan las excepciones de este tipo.

La forma habitual de definir las transacciones en JPA es la definida en el siguiente código:

```
EntityManager em = emf.createEntityManager();
EntityTransaction tx = em.getTransaction();
   try {
      tx.begin();
      // Operacion sobre entidad 1
      // Operacion sobre entidad 2
      tx.commit();
   } catch (RunTimeException ex) {
      tx.rollback();
   } finally {
      em.close();
}
```

Primero se obtiene la transacción y se llama al método `begin()`. Después se realizan todas las operaciones dentro de la transacción y se realiza un `commit()`. Todo ello se engloba en un bloque de `try/catch`. Si alguna operación falla lanza una excepción que se captura y se ejecuta el `rollback()`. En cualquier caso se cierra el entity manager.

Cuando se deshace una transacción en la base de datos todos los cambios realizados durante la transacción se deshacen también. La base de datos vuelve al estado previo al comienzo de la transacción. Sin embargo, el modelo de memoria de Java no es transaccional. No hay forma de obtener una instantánea de un objeto y revertir su estado a ese momento si algo va mal. Una de las cuestiones más complicadas de un mapeo entidad-relación es que mientras que podemos utilizar una semántica transaccional para decidir qué cambios se realizan en la base de datos, no podemos aplicar las mismas técnicas en el contexto de persistencia en el que viven las instancias de entidades.

Siempre que tenemos cambios que deben sincronizarse en una base de datos, estamos trabajado con un contexto de persistencia sincronizado con una transacción. En un momento dado durante la vida de la transacción, normalmente justo antes de que se realice un commit, esos cambios se traducirán en sentencias SQL y se enviarán a la base de datos.

Si la transacción hace un rollback pasarán entonces dos cosas. Lo primero es que la transacción en la base de datos será deshecha. Lo siguiente que sucederá será que el contexto de persistencia se limpiará (*clear*), desconectando todas las entidades que se gestionaban. Tendremos entonces un montón de entidades desconectadas de la base de datos con un estado no sincronizado con la base de datos.

1.2. Concurrencia y niveles de aislamiento

La gestión de la concurrencia en transacciones es un problema complejo. Por ejemplo, supongamos un sistema de reservas de vuelos. Sería fatal que se vendiera el mismo asiento de un vuelo a dos personas distintas por el hecho de que se han realizado dos accesos concurrentes al siguiente método:

```
public void reservaAsiento(Pasajero pasajero, int numAsiento, long
numVuelo) {
   EntityManager em = emf.createEntityManager();
   AsientoKey key = new AsientoKey(numAsiento, numVuelo);
   em.getTransaction().begin();
   Asiento asiento = em.find(Asiento.class, key);
   if (! asiento.getOcupado()) {
      asiento.setOcupado(true);
      asiento.setPasajero(pasajero);
      pasajero.setAsiento(asiento);
   }
   em.getTransaction().commit();
   em.close();
}
```

Si no se controlara el acceso concurrente de las transacciones a la tabla de asientos, podría

suceder que dos transacciones concurrentes accedieran al estado del asiento a la misma vez (antes de haberse realizado el UPDATE de su atributo `ocupado`), lo vieran libre y se lo asignaran a un pasajero y después a otro. Uno de los pasajeros se quedaría sin billete.

Se han propuesto múltiples soluciones para resolver el acceso concurrente a los datos. El estándar SQL define los llamados *niveles de aislamiento* (isolation levels) que tratan este problema. Los niveles más bajos solucionan los problemas más comunes y permiten al mismo tiempo que la aplicación responda sin generar bloqueos. El nivel más alto alto garantiza que todas las transacciones son serilizables, pero obliga a que se definan un número excesivo de bloqueos.

El estándar SQL define cuatro posibles problemas que pueden generarse cuando dos transacciones concurrentes realizan SELECTS y UPDATES en la base de datos:

- **Actualizaciones perdidas** (*lost updates*): este problema ocurre cuando dos transacciones hacen un UPDATE sobre el mismo dato y una de ellas aborta, perdiéndose los cambios de ambas transacciones. Un ejemplo: (1) un dato tiene un valor V0; (2) la transacción T1 comienza; (3) la transacción T2 comienza y actualiza el dato a V2; (4) la transacción T1 lo actualiza a V1; (5) la transacción T2 hace un commit, guardando V2; (6) por último, la transacción T1 se aborta realizando un rollback y devolviendo el dato a su estado inicial de V0.
- **Lecturas de datos sucios** (*dirty readings*): sucede cuando una transacción hace un SELECT y obtiene un dato modificado por un UPDATE de otra transacción que no ha hecho commit. El dato es *sucio* porque todavía no ha sido confirmado y puede cambiar. Por ejemplo: (1) un dato tiene un valor V0; (2) la transacción T1 lo actualiza a V1; (3) la transacción T2 lee el valor V1 del dato; (4) la transacción T1 hace un rollback, volviendo el dato al valor V0, y quedando sucio el dato contenido en la transacción T2.
- **Lecturas no repetibles** (*urepeatable read*): sucede cuando una transacción lee un registro dos veces y obtiene valores distintos por haber sido modificado por otro UPDATE confirmado. Por ejemplo: (1) una transacción T1 lee un dato; (2) comienza otra transacción T2 que lo actualiza; (3) T2 hace un commit; (4) T1 vuelve a leer el dato y obtiene un valor distinto al primero.
- **Lecturas fantasmas** (*phantom read*): una transacción ejecuta dos consultas y en la segunda aparecen resultados que no son compatibles con la primera (registros que se han borrado o que han aparecido). El mismo ejemplo anterior, cambiando la lectura de T1 por una consulta.

Las bases de datos SQL definen cuatro posibles niveles de aislamiento que corresponden a modos de funcionamiento de la base de datos en el que se garantiza que no sucede alguno de los problemas anteriores. Tradicionalmente se han utilizado bloqueos para asegurar estos niveles. Vamos a describirlos, indicando la estrategia de bloqueo utilizada en cada caso. De menor a mayor nivel de seguridad son los siguientes:

- **READ_UNCOMMITTED**: garantiza que no ocurren actualizaciones perdidas. Si la base de datos utiliza bloqueos, un UPDATE de un dato lo bloqueara para escritura. De

esta forma, otras transacciones podrán leerlo (y se podrán producir cualquiera de los otros problemas) pero no escribirlo. Cuando se realiza un commit se libera el bloqueo.
- **READ_COMMITTED**: garantiza que no existen las lecturas sucias. Utilizando bloqueos, se podría resolver haciendo que un UPDATE de un dato lo bloqueara para lectura y escritura. Cualquier otra transacción que intente leer el dato quedará en espera hasta que se confirme la transacción.
- **REPEATABLE_READ**: garantiza que otra transacción no modifica un dato leído. Para asegurar este nivel utilizando bloqueos, un SELECT sobre un dato lo bloquea frente a otras actualizaciones.
- **SERIALIZABLE**: garantiza que no se producen lecturas fantasmas. Es el nivel máximo de seguridad y para garantizarlo utilizando bloqueos se deben bloquear tablas enteras, no solo registros.

Como hemos dicho, los niveles de aislamiento se han garantizado tradicionalmente utilizando distintos tipos de bloqueos en los accesos a los datos. Recientemente, sin embargo, bases de datos como Oracle, Postgress o MySQL con InnoDB proporcionan alternativas distintas a los bloqueos para garantizar estos niveles. En concreto, permiten usar el llamado *Multiversion Concurrency Control* (MVCC) en el que se utilizan versiones de los datos para garantizar las condiciones de aislamiento. Con esta estrategia, en el nivel READ_COMMITTED cuando una transacción lee de un dato que no ha sido confirmado se lee la versión anterior del dato (la última que ha sido confirmada). Incluso en el modo REPEATABLE_READ el primer SELECT y el segundo obtendrían el mismo valor anterior a la modificación a la modificación de la transacción 2. En general, con MVCC las operaciones de lectura no bloquean, sino que obtienen la última versión confirmada del dato.

¿Cuál es el nivel de aislamiento recomendable para una aplicación? El nivel READ_UNCOMMITTED es demasiado permisivo, ya que permite que una transacción lea datos que no han sido confirmados por otra transacción. El SERIALIZABLE es demasiado restrictivo y hace que la aplicación no escale correctamente. Se producirían demasiados bloqueos para resolver un problema que no sucede demasiado a menudo (lecturas fantasmas). Esto nos deja con los niveles READ_COMMITTED y REPEATABLE_READ. La mayoría de aplicaciones en producción usan alguno de estos niveles.

Una solución muy frecuente es utilizar el nivel READ_COMMITTED como nivel por defecto y realizar bloqueos puntuales en aquellas ocasiones peligrosas en las que puede suceder un problema como el de la butaca de cine. En SQL se puede utilizar la instrucción

```
SELECT ... FOR UPDATE
```

para bloquear un determinado registro explícitamente para lectura y escritura hasta realizar un commit.

1.3. Gestión concurrencia con JPA

Los niveles de aislamiento vistos en el apartado anterior se gestionan por la propia base de datos y afectan a JPA en el momento en que el entity manager hace un flush y se generan las sentencias SELECT y UPDATE. En ese momento el sistema de base de datos toma el control de la concurrencia utilizando el nivel de seguridad definido por defecto.

En la versión estándar de JPA no es posible definir el nivel de aislamiento de la base de datos subyacente. Esta opción es dependiente de la implementación de JPA. En la versión de Hibernate, se puede definir en el fichero `peristence.xml`, con la propiedad `hibernate.connection.isolation`. Los posibles valores son:

- 1: READ_UNCOMMITTED
- 2: READ_COMMITTED
- 4: REPEATABLE_READ
- 8: SERIALIZABLE

JPA utiliza por defecto un sistema optimista de gestión de la concurrencia. Para que funcione correctamente necesita que el nivel de aislamiento de la base de datos sea READ_COMMITTED. Recordemos que este nivel no protege del problema de las lecturas no repetidas (una transacción lee un dato y otra transacción lo modifica antes de que la primera haga un commit).

El control optimista de la concurrencia asume que no suceden conflictos y no se realizan bloqueos sobre los datos. Sin embargo, si al final de la transacción se ha producido un error se lanza una excepción.

En un control optimista de concurrencia todos los objetos tienen un atributo adicional que guarda su número de versión (una columna adicional en la tabla). Las lecturas leen el número de versión y las escrituras lo incrementan. Cuando una transacción intenta escribir en una versión de un objeto que no corresponde con la que había leído previamente se aborta la actualización y se genera una excepción.

Por ejemplo, supongamos que una transacción T1 realiza una lectura sobre un objeto. Se obtiene automáticamente su número de versión. Supongamos que otra transacción T2 modifica el objeto y realiza un commit. Automáticamente se incrementa su número de versión. Si ahora la transacción T1 intenta modificar el objeto se comprobará que su número de versión es mayor que el que tiene y se generará una excepción.

En este caso el usuario de la aplicación que esté ejecutando la transacción T1 obtendrá un mensaje de error indicando que alguien ha modificado los datos y que no es posible confirmar la operación. Lo deberá intentar de nuevo.

Para que JPA pueda trabajar con versiones es necesario que los objetos tengan un atributo marcado con la anotación `@Version`. Este atributo puede ser del tipo `int`, `Integer`, `short`, `Short`, `long`, `Long` y `java.sql.Timestamp`.

```
@Entity
public class Autor {
   @Id
```

```
   private String nombre;
   @Version
   private int version;
   private String correo;
   ...
}
```

Aunque el estándar no lo permite, en Hibernate es posible definir un comportamiento optimista en entidades que no definen una columna de versión. Para ello basta con definir la entidad de la siguiente forma:

```
@Entity
@org.hibernate.annotations.Entity (
   optimisticLock = OptimisticLockType.ALL,
   dynamicUpdate = true
)
public class Autor {
   @Id
   private String nombre;
   private String correo;
   ...
}
```

De forma complementaria al comportamiento optimista, también es posible en JPA definir bloqueos explícitos sobre objetos. Para ello hay que utilizar el método `lock(objeto,LockModeType)` del entity manager. Este método bloquea un objeto (registro) para lectura y escritura. Hay que pasarle como parámetro el objeto sobre el que se realiza el bloqueo y el tipo de bloqueo. El tipo de bloqueo puede ser `LockModeType.READ` y `LockModeType.WRITE`. La diferencia entre ambos es que el segundo incrementa automáticamente el número de versión del objeto, independientemente de que se haga después una actualización o no. Cualquier intento de escritura de una versión anterior del objeto generará una excepción.

Por ejemplo, si se quisiera evitar el problema del asiento del vuelo bloqueando explícitamente el registro, habría que escribir el siguiente código:

```
public void reservaAsiento(Pasajero pasajero, int numAsiento, long
numVuelo) {
   EntityManager em = emf.createEntityManager();
   AsientoKey key = new AsientoKey(numAsiento, numVuelo);
   em.getTransaction().begin();
   Asiento asiento = em.find(Asiento.class, key);
   em.lock(asiento,LockType.WRITE);
   if (! asiento.getOcupado()) {
      asiento.setOcupado(true);
      asiento.setPasajero(pasajero);
      pasajero.setAsiento(asiento);
   }
   em.getTransaction().commit();
   em.close();
}
```

Una llamada a `lock` genera inmediatamente una instrucción SQL `SELECT ... FOR UPDATE`, sin esperar a que se realice un flush del contexto de persistencia. Esta instrucción

SQL hace que el gestor de base de datos realice un bloqueo del registro.

2. Transacciones y EAO

Veamos cómo incorporar las transacciones en nuestro diseño de un EAO. Una técnica muy habitual en los entity beans y en los managers de Spring que implementan un *session façade* es declarar las transacciones de las operaciones como `REQUIRED_NEW`. Esto significa que la operación comprueba si existe una transacción activa en el momento de ejecutarse. Si es así se ejecuta dentro de esa transacción y si no crea una transacción nueva que se cierra al final de la operación.

Es posible utilizar el método `isActive()` para hacer conseguir este comportamiento. Veámoslo en el siguiente código:

```
public class EmpleadoDAO {

                    // ...
                    public Empleado subeSueldoEmpleado(Empleado emp,
long aumento) {
                       boolean transaccionActiva = false;

                       if (!em.getTransaction().isActive()) {
                          em.getTransaction().begin();
                          transaccionActiva = true;
                       }

                       if (aumento > 0)
                          emp.setSueldo(emp.getSueldo + aumento);
                       else
                          // error

                       if (transaccionActiva)
em.getTransaction().commit();

                       return empleado;
                    }
                 }
```

El método del DAO `subeSueldoEmpleado()` comprueba al comienzo si hay una transacción activa y pone a `true` una variable booleana en ese caso. Al finalizar, comprueba el valor de la variable y cierra la transacción si se ha abierto en el método. En el caso en que el método se haya llamado con una transacción ya comenzadas, la variable `transaccionActiva` seguirá a `false` y no se cerrará la transacción.

Habría que hacer esta comprobación en todos los métodos del DAO. En el proyecto de integración veremos un ejemplo completo.