Java Persistence API

- Sesión 6: Transacciones

Índice

- Introducción
- Atomicidad
- Concurrencia y niveles de aislamiento
- Gestión de la concurrencia con JPA
- Transacciones y EAOs

Características generales

- ACID
  - A = atómicas
  - I = aisladas
- En JDBC
  - autocommit = true por defecto
  - setAutocommit(false) para gestionar explícitamente las transacciones
  - Aislamiento dependiente de la BD: bloqueos y MMVC
- JPA: capa sobre JDBC
  - bloqueos optimistas

Transacciones JPA no son distribuidas

- Las transacciones JPA (gestionadas por la aplicación) se basan directamente en la gestión de transacciones de la BD
- Si necesitamos transacciones distribuidas, deberemos utilizar servidores de aplicaciones y JTA

Interfaz EntityTransaction

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

- Se obtiene un EntityTransaction a partir del entity manager con el método getTransaction()
- Sólo puede haber una transacción activa en un entity manager

Excepciones

- Si se intenta comenzar una transacción cuando otra ya está activa se lanza una `IllegalStateException`
- Si se llama a `commit()` o `rollback()` con una transacción no activa se lanza una `IllegalStateException`
- Si ocurre un error durante el rollback se lanza una `PersistenceException`
- Si ocurre un error durante el commit se lanza una `RollbackException` (hija de `PersistenceException`)
- Todas las excepciones son de tipo `RunTimeException`

Ejemplo de gestión atómica

```
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();
}
```

Cuando se deshace una transacción en la base de datos todos los cambios realizados durante la transacción se deshacen también. 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.

Concurrencia

- Problema complejo que impacta directamente en el rendimiento de la aplicación
- Ejemplos variados: reserva de asiento, lectura/escritura de valores

```
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();
}
```

Ejemplo 2

- La concurrencia añade un elemento de complejidad adicional a la gestión de transacciones
- Supongamos las siguientes operaciones. Si se ejecutan concurrentemente 2 clientes es posible que X quede en un estado inconsistente.

Cliente 1

1. **Leer el dato X**
2. **Sumar 10 a X**
3. **Escribir el dato X**

Cliente 2

1. **Leer el dato X**
2. **Sumar 10 a X**
3. **Escribir el dato X**

Two-phase locking

- Antes de usar un dato, éste se bloquea.
- Dos tipos de bloqueos: de lectura y de escritura.
- Un bloqueo de lectura entra en conflicto con otro de escritura. Un bloqueo de escritura entra en conflicto con otro de escritura.
- Una transacción puede obtener un dato si no se produce ningún conflicto. En caso de no poder obtener el dato, queda en espera hasta que el dato se libera.

Ejemplo

Cliente 1	Cliente 2
1. Bloqueo de escritura sobre X	1. Bloqueo de escritura sobre X
2. Leer el dato X	2. Leer el dato X
3. Sumar 10 a X	3. Sumar 10 a X
4. Escribir el dato X	4. Escribir el dato X
5. Liberar X	5. Liberar X

Control optimista

Cliente 1

1. **Clonar X, guardando el número de versión**
2. **Sumar 10 a X**
3. **Si el número de versión de X es el mismo que el del clon: escribirlo**
4. **Sino: abortar la transacción**

Cliente 2

1. **Clonar X, guardando el número de versión**
2. **Sumar 10 a X**
3. **Si el número de versión de X es el mismo que el del clon: escribirlo**
4. **Sino: abortar la transacción**

Problemas de aislamiento (1)

- **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.

Problemas de aislamiento (2)

- **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.

Niveles de aislamiento SQL

- **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.

¿Cuál es el mejor nivel de aislamiento?

- READ_UNCOMMITTED: demasiado permisivo
- SERIALIZABLE: demasiado estricto
- Solución frecuente en JDBC:
  - READ_COMMITTED por defecto
  - Bloqueos explícitos con SELECT ... FOR UPDATE

Solución en JPA

- Nivel por defecto de la BD: READ_COMMITTED
- Bloqueos optimistas, utilizando versiones:

```
@Entity
public class Autor {
   @Id
   private String nombre;
   @Version
   private int version;
   private String correo;
   ...
}
```

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

Bloqueos optimistas

- Una transacción T1 realiza una lectura sobre un objeto. Se obtiene automáticamente su número de versión.
- 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.

Bloqueos explícitos

```
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();
}
```

Transacciones y EAO

```
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;
   }
   ...
}
```

¿Preguntas?