Transacciones con Spring Framework y EJB 3.0
Escrito por Leonardo De Seta

El procesamiento de transacciones debería lograr un alto grado de integridad y consistencia de datos. En este artículo veremos algunos errores comunes al usar transacciones en Java. Usaremos ejemplos con Spring Framework y Enterprise JavaBeans (EJB) 3.0 para demostrar el uso de una estrategia transaccional efectiva.

El porqué de las transacciones

El uso más común de las transacciones dentro de una aplicación es para mantener un alto grado de integridad y consistencia en los datos. Si no nos importa la calidad de nuestros datos, no tenemos que preocuparnos en usar transacciones. Después de todo, el soporte de transacciones en la plataforma Java puede afectar el rendimiento, introducir problemas de concurrencia en la base de datos, y aumentar la complejidad de la aplicación.

Pero aquellos desarrolladores que no se preocupan por las transacciones lo hacen bajo su propio riesgo. Casi todas las aplicaciones de negocio requieren un alto nivel de calidad de datos. Solo la industria financiera de inversiones gasta decenas de billones de dólares en negocios fallidos, y los datos incorrectos por malas transacciones son la segunda causa. Aunque la falta de soporte transaccional es sólo uno de los factores que llevan a datos incorrectos (aunque uno de los factores más importantes), se puede inferir que billones de dólares se gastan en la industria financiera como resultado de un pobre soporte transaccional.

Otra fuente de problemas es el desconocimiento sobre el soporte de transacciones. Muy a menudo escucho frases como "no necesitamos transacciones en nuestra aplicación porque nunca falla". Si, claro. Fui testigo de aplicaciones que, de hecho, casi nunca lanzaban excepciones. Estas aplicaciones tenian código bien escrito, rutinas de validación bien escritas, pruebas completas y cobertura de código para evitar el costo en rendimiento y la complejidad asociada al procesamiento de transacciones. El problema con esta forma de pensar es que sólo tiene en cuenta una de las características del soporte transaccional: la ***atomicidad***. La atomicidad asegura que todas las actualizaciones son tratadas como una única unidad, y son todas confirmadas (commit) o deshechas (rollback). Pero el soporte transaccional es más que hacer un rollback o coordinar actualizaciones. Otro aspecto, el ***aislamiento***, asegura que una unidad de trabajo está aislada del resto de las unidades de trabajo. Sin un aislamiento transaccional adecuado, otras unidades de trabajo pueden acceder a actualizaciones que están siendo realizadasd por una unidad de trabajo en curso, aunque el proceso de esta unidad de trabajo esté incompleto. Como resultado, se podrían tomar decisiones de negocio en base a datos parciales, lo que podría resultado en operaciones fallidas u otros resultados negativos (o costosos).

Entonces, dado el alto costo y el impacto negativo de los datos incorrectos, y con el conocimiento básico para comprender que las transacciones son importantes (y necesarias), necesitamos utilizar las transacciones y aprender cómo saltear los problemas que pueden surgir. Entonces avanzamos y agregamos transacciones a nuestras aplicaciones. Y aquí es donde empiezan los problemas. Las transacciones no siempre parecen funcionar como se espera en la plataforma Java. A continuación veremos porqué ocurre esto, con ejemplos de código y errores comunes que se comenten en la mayoría de los entornos productivos.

Para los ejemplos de código vamos a usar Spring Framework (versión 2.5), pero los conceptos de transacciones son los mismos para la especificación EJB 3.0. En la mayoría de los casos, es sólo cuestión de reemplazar la anotación de Spring Framework @Transactional con la anotación de EJB 3.0 @TransactionAttribute. En los casos es que haya diferencias entre estos frameworks, vamos a incluir ejemplos para ambos.

Problemas comunes en transacciones locales

Un buen punto de inicio es con el escenario más simple: el uso de *transacciones locales*, también conocidas como *transacciones de base de datos*. En los primeros días de la persistencia en las bases de datos (por ejemplo, con JDBC), se solía delegar el procesamiento de las transacciones a la base de datos. Después de todo, ¿no es lo que las bases de datos hacen? Las transacciones locales trabajan bien para Unidades Lógicas de Trabajo (ULT, o LUW en inglés) que ejecutan una única sentencia insert, update o delete. Por ejemplo, a continuación vemos código usando JDBC que realiza un insert de una orden de compra a la tabla TRADE:

```
@Stateless
public class TradingServiceImpl implements TradingService {
  @Resource SessionContext ctx;
  @Resource(mappedName="java:jdbc/tradingDS") DataSource ds;

  public long insertTrade(TradeData trade) throws Exception {
    Connection dbConnection = ds.getConnection();
    try {
      Statement sql = dbConnection.createStatement();
      String stmt =
        "INSERT INTO TRADE (ACCT_ID, SIDE, SYMBOL, SHARES, PRICE, STATE)"
       + "VALUES ('"
       + trade.getAcct() + "','"
       + trade.getAction() + "','"
       + trade.getSymbol() + "',"
       + trade.getShares() + ","
       + trade.getPrice() + ",'"
       + trade.getState() + "')";
      sql.executeUpdate(stmt, Statement.RETURN_GENERATED_KEYS);
      ResultSet rs = sql.getGeneratedKeys();
      if (rs.next()) {
        return rs.getBigDecimal(1).longValue();
      } else {
        throw new Exception("Trade Order Insert Failed");
      }
    } finally {
      if (dbConnection != null) dbConnection.close();
    }
  }
}
```

Este código JDBC no incluye ninguna lógica de transacción, y persiste una orden de compra de acciones en la tabla TRADE de la base de datos. En este caso, la base de datos se encarga de manejar la lógica transaccional.

Esto funciona bien para una única acción sobre la base de datos dentro de esta ULT. Pero supongamos que necesitamos actualizar el balance de la cuenta en el mismo momento que insertamos la orden de compra en la base de datos. Por ejemplo:

```
public TradeData placeTrade(TradeData trade) throws Exception {
  try {
    insertTrade(trade);
    updateAcct(trade);
    return trade;
  } catch (Exception up) {
    //log del error
    throw up;
  }
}
```

En este caso, los métodos insertTrade() y updateAcct() usan código estándar JDBC sin transacciones. Una vez que el método insertTrade() finaliza, la base de datos persiste (y hace commit) a la orden de compra. Si el método updateAcct() falla por cualquier razón, la orden de compra seguirá en la tabla TRADE al finalizar el método placeTrade(), resultando en datos inconsistentes en la base de datos. Si el método placeTrade() hubiera usado transacciones, ambas actividades se hubieran incluido en la misma ULT, y se hubiera realizado un rollback de la orden de compra si la actualización de la cuenta fallaba.

Con la popularidad de los frameworks de persistencia Java como Hibernate, TopLink y Java Persistence API (JPA), es bastante raro escribir código JDBC directamente. Generalmente usamos los frameworks de mapeo objeto-relacional (ORM) para hacernos la vida más facil y reemplazar todo el código JDBC molesto por unas simples invocaciones. Por ejemplo, para insertar una orden de compra del ejemplo anterior en JDBC, usando Spring Framework con JPA, mapeamos el objeto TradeData a la tabla TRADE y reemplazamos todo el código JDBC con el código JPA:

```
public class TradingServiceImpl {
   @PersistenceContext(unitName="trading") EntityManager em;

   public long insertTrade(TradeData trade) throws Exception {
      em.persist(trade);
      return trade.getTradeId();
   }
}
```

Notemos que en este ejemplo invocamos al método persist() del EntityManager para insertar una orden de compra. Simple, ¿no?. No tanto. Este código no va a insertar la orden de compra en la tabla TRADE como esperaríamos, ni tampoco lanza una excepción. Simplemente va a retornar el valor 0 como la clave de la orden de compra sin cambiar la base de datos. Este es uno de los primeros errores al procesar transacciones:**los frameworks ORM necesitan de transacciones para disparar la sincronización entre el caché de objetos y la base de datos**. Es a través del commit de la transacción que el código SQL se genera y se afecta a la base de datos con la acción (es decir, con el insert, update, delete). Sin la transacción no hay un disparador al ORM para que genere el código SQL y persista los cambios, y así el método termina - sin excepciones, sin actualizaciones. Si se usa un framework ORM, debemos usar transacciones. Ya no podemos confiar en que la base de datos va a manejar las conexiones y hacer el commit del trabajo.

Estos ejemplos simples deberían bastar para dejar en claro que las transacciones son necesarias para mantener integridad y consistencia de datos. Pero apenas son el inicio de la complejidad y los problemas asociados con la implementación de transacciones en Java.

Problemas comunes con la anotación @Transactional de Spring Framework

Entonces, probamos el código anterior y descubrimos que el método persist() no funciona sin transacciones. Buscamos en Internet, investigamos algunos links y nos encontramos con que Spring Framework tiene una anotación @Transactional. Así que la agregamos a nuestro código, como vemos a continuación:

```
public class TradingServiceImpl {
  @PersistenceContext(unitName="trading") EntityManager em;

  @Transactional
  public long insertTrade(TradeData trade) throws Exception {
    em.persist(trade);
    return trade.getTradeId();
  }
}
```

Volvemos a probar el código, y sigue sin funcionar. El problema es que hay que decirle a Spring que estamos uando anotaciones para gestionar las transacciones.A menos que estemos haciendo una prueba unitaria completa, este error suele ser dificil de descubrir, y hace que los desarrolladores terminen añadiendo la lógica de transacciones en los archivos de configuración de Spring en vez de usar anotaciones.

Cuando se usa la anotación @Transactional de Spring, debemos agregar la línea siguiente a nuestro archivo de configuración de Spring:

```
<tx:annotation-driven transaction-manager="transactionManager" />
```

La propiedad transaction-manager tiene una referencia al bean que gestiona las transacciones, definido en el archivo de configuración de Spring. Este código le dice a Spring que use la anotación @Transactional cuando aplique el interceptor de transacciones. Sin esta línea, se ignora la anotación @Transactional, y en consecuencia no se usan transacciones en el código.

Este es el uso básico de la anotación @Transactional, y es sólo el comienzo. En el ejemplo usamos la anotación @Transaction sin especificar ningún parámetro adicional a al anotación. Muchos desarrolladores usan la anotación @Transactional sin tomarse el tiempo de comprender lo que hace. Por ejemplo, cuando se usa la anotación @Transactional por si sola, como en el ejemplo anterior, ¿cuál es el modo de propagación? ¿En qué valor está el flag de sólo-lectura? ¿Cuál es el nivel de aislamiento? Más importante, ¿cuándo se realiza un rollback? Es importante comprender cómo funciona esta anotación para asegurarnos de usar el soporte correcto de transacciones en nuestra aplicación. Y entonces, respondiendo a todas estas preguntas: cuando se usa la anotación @Transactional sin ningún parámetro, el modo de propagación es REQUIRED, el flag de sólo-lectura es false, el nivel de aislamiento de la transacción es READ_COMMITTED, y la transacción no va a realizar un rollback sobre una excepción no-chequeada (o sea, que no sea de java.lang.RuntimeException).

Problemas comunes con el flag de sólo-lectura de @Transactional

Un problema común es el uso incorrecto del flag de sólo-lectura de la anotación @Transactional de Spring. Veamos algunos ejemplos.

Uso de sólo-lectura con propagación SUPPORTS en JDBC

Una pregunta rápida: si usamos código JDBC estándard para la persistencia, ¿qué hace la anotación cuando el flag de sólo-lectura está en true y el modo de propagación en SUPPORTS? Veamos el ejemplo:

```
@Transactional(readOnly = true, propagation=Propagation.SUPPORTS)
public long insertTrade(TradeData trade) throws Exception {
  //código JDBC acá...
}
```

¿Sabés qué ocurre? A ver, vamos con unas opciones. Cuando se ejecuta el método del ejemplo anterior:

A) Se lanza una excepción indicando que es una conexión de sólo-lectura
B) Se inserta correctamente la orden de compra y se hace el commit
C) No se hace nada porque el nivel de propagación está en SUPPORTS

¿Te rendís? La opción correcta es B. La orden de compra se inserta correctamente en la base de datos, incluso aunque el flag de sólo-lectura estaba en true y el nivel de propagación en SUPPORTS. ¿Cómo puede ser? No se inició ningúna transacción porque el modo de propagación era SUPPORTS, así que el método usó el modo de transacciones locales (de base de datos). Y el flag de sólo-lectura sólo aplica si se inicia una transacción. En este caso, no se inició ninguna transacción, así que se ignoró el flag.

Uso de sólo-lectura con propagación REQUIRED en JDBC

Vamos con otra oportunidad. ¿Qué hace la anotación @Transactional cuando el flag de sólo-lectura está activo, y el modo de propagación en REQUIRED? Como en este ejemplo:

```
@Transactional(readOnly = true, propagation=Propagation.REQUIRED)
public long insertTrade(TradeData trade) throws Exception {
  //código JDBC acá...
}
```

Cuando se ejecuta el método insertTrade() lo que ocurre es:

A) Se lanza una excepción indicando que es una conexión de sólo-lectura
B) Se inserta correctamente la orden de compra y se hace el commit
C) No se hace nada porque el flag de sólo-lectura está en true

Con la explicación anterior, esta vez debería ser más facil de responder. La respuesta correcta es A. O al menos es lo que debería ocurrir. Se lanza una excepción, indicando que se está intentado realizar una operación de actualización sobre una conexión de sólo-lectura. Como se inició la transacción (REQUIRED), la conexión se estableción en sólo-lectura. Así que cuando se ejecuta la sentencia SQL, se obitene una excepción diciendo que la conexión es de sólo-lectura.

Sin embargo, dependiendo de la base de datos que utilicen, puede que ocurra B: es decir, que el flag de sólo lectura sea ignorado completamente, y la conexión se comporte normalmente.

Lo extraño del flag de sólo-lectura es que se necesita iniciar una transacción para usarlo. ¿Por qué se necesita una transacción si sólo se van a leer datos? La respuesta es que, en realidad, no se necesita. Iniciar una transacción para realizar operaciones de sólo-lectura agregar procesamiento extra, y puede ocasionar bloqueos

de lectura compartida en la base de datos (dependiendo del tipo de base de datos y del nivel de aislamiento que se use). En resumen, el flag de sólo lectura no tiene mucho sentido cuando se usa persistencia usando JDBC, y ocasiona procesamiento extra cuando se inicia una transacción innecesaria.

Uso de sólo-lectura con propagación REQUIRED en JPA

¿Y qué pasa cuando usamos un framework ORM? Siguiendo con este formato de preguntas, ¿podrías adivinar cual es el resultado de la anotación @Transactional si se usara en el método insertTrade() que utiliza JPA con Hibernate?

```
@Transactional(readOnly = true, propagation=Propagation.REQUIRED)
public long insertTrade(TradeData trade) throws Exception {
  em.persist(trade);
  return trade.getTradeId();
}
```

Cuando se ejecuta el método insertTrade():

A) Se lanza una excepción indicando que es una conexión de sólo-lectura
B) Se inserta correctamente la orden de compra y se hace el commit
C) No se hace nada porque el flag de sólo-lectura está en true

La opción correcta es la B. La orden de compra se inserta en la base de datos sin ningún error. A ver un minuto el ejemplo anterior nos mostró que se lanza una excepción por conexión de sólo lectura cuando el modo de propagación era REQUIRED. Esto es cierto cuando se usa JCBC. Cuando se usa un framework ORM, el flag de sólo-lectura es sólo un "consejo" para la base de datos y una directiva para el ORM (en este caso, Hibernate) para establecer el modo de flush del caché de objetos a NEVER (Nunca), indicando que el caché de objetos no debe sincronizarse con la base de datos durante esta unidad de trabajo. Sin embargo, el modo de propagación REQUIRED tiene precedencia sobre todo esto, permitiendo a la transacción iniciar y funcionar como si no estuviera el flag de sólo-lectura.

Uso de sólo-lectura en JPA

Lo que nos lleva a otro error frecuente. Sabiendo todo esto, ¿cuál suponés que será el resultado de ejecutar el siguiente código, sólo asignando el valor del flag de sólo lectura?

```
@Transactional(readOnly = true)
public TradeData getTrade(long tradeId) throws Exception {
  return em.find(TradeData.class, tradeId);
}
```

El método getTrade():

A) Inicia una transacción, obtiene la orden de compra, y luego hace commit de la transacción
B) Obtiene la orden de compra sin iniciar una transacción

La respuesta correcta es la A. Se inicia una transacción y se hace un commit. No nos olvidemos que el modo de propagación por defecto de la anotación @Transactional es REQUIRED. Esto significa que se inicia una transacción cuando, en realidad, no se necesitaba de una. Dependiendo de la base de datos que usemos, esto puede ocasionar bloqueos compartidos innecesarios, lo que puede resultar en situaciones de deadlock en la base de datos. Además, se consume tiempo de procesamiento y recursos innecesarios para iniciar y detener la transacción.

En resumen, cuando se usa un framework ORM, el flag de sólo-lectura es practicamente inutil, y en la mayoría de los casos se ignora. Pero si vamos a insitir con su uso, siempre tenemos que establecer el modo de

propagación a SUPPORTS, de manera que no se inicie una transacción, como en el ejemplo siguiente:

```
@Transactional(readOnly = true, propagation=Propagation.SUPPORTS)
public TradeData getTrade(long tradeId) throws Exception {
  return em.find(TradeData.class, tradeId);
}
```

Mejor aún, simplemente evitamos usar la anotación @Transactional cuando hacemos operaciones de sólo lectura:

```
public TradeData getTrade(long tradeId) throws Exception {
  return em.find(TradeData.class, tradeId);
}
```

Problemas comunes con el atributo REQUIRES_NEW

Cuando usamos Spring Framework o EJB, el mal uso del atributo de transacción REQUIRES_NEW puede tener resultados negativos y llevar a corrupción e inconsistencia de datos. El atributo de transacción REQUIRES_NEW siempre inicia una transacción nueva cuando comienza el método, exista o no una transacción en curso. Muchos desarrolladores usan al atributo REQUIRES_NEW de manera incorrecta, asumiendo que es la forma correcta de asegurar el inicio de transacciones. Consideremos los siguientes dos métodos:

```
@Transactional(propagation=Propagation.REQUIRES_NEW)
public long insertTrade(TradeData trade) throws Exception {...}

@Transactional(propagation=Propagation.REQUIRES_NEW)
public void updateAcct(TradeData trade) throws Exception {...}
```

Ambos métodos son públicos, por lo que pueden ser invocados de forma independiente uno del otro. Los problemas surgen con el atributo REQUIRES_NEW cuando hay métodos que los usan dentro de la misma Unidad Lógica de Trabajo a través de comunicación entre servicios, o usando orquestación. Por ejemplo, supongamos que invocamos al método updateAcct() de forma independiente de cualquier otro método en algunos casos de uso, pero también hay un caso donde el método se invoca dentro del método insertTrade(). Ahora bien, si ocurre una excepción luego de la invocación a updateAcct(), la orden de compra va a tener un rollback, pero se va a realizar un commit de la actualización a la cuenta. Por ejemplo:

```
@Transactional(propagation=Propagation.REQUIRES_NEW)
public long insertTrade(TradeData trade) throws Exception {
  em.persist(trade);
  updateAcct(trade);
  //Una excepción ocurre acá! La orden de compra tiene un rollback,
  // pero la actualización de la cuenta no!
}
```

Esto ocurre porque se inició una transacción nueva en el método updateAcct(), de manera que ocurre un commit cuando termina el método updateAcct(). Cuando se usa el atributo de transacción REQUIRES_NEW, si hay un contexto transaccional presente, la transacción en curso se suspende y se inicia una nueva transacción. Una vez que termina el método, se realiza el commit de la nueva transacción y se continua con la transacción original.

Debido a este comportamiento, el atributo de transacción REQUIRES_NEW sólo debe usarse si la acción sobre la base de datos necesita guardarse sin importar el resultado de la transacción subyacente. Por ejemplo, supongamos que se necesita guardar una auditoria de cada intento de orden de compra que se realice. Se necesita persistir esta información sin importar si hubo problemas de validación, fondos insuficientes, o

cualquier otra razón. Si no usamos el atributo REQUIRES_NEW en el método de auditoria, el registro de auditoria sufriría un rollback junto con la orden de compra fallida. El uso del atributo REQUIRES_NEW garantiza que los datos de auditoria se guarden sin importar el resultado de la transacción original.

En resumen, siempre hay que usar los atributos MANDATORY o REQUIRED en lugar de REQUIRES_NEW a menos que tengamos razones para usarlo, por motivos similares al del ejemplo de auditoria.

Problemas comunes con el rollback de transacciones

Por último vamos a ver uno de los problemas más comunes con las transacciones. Desafortunadamente vi este tipo de problemas en código productivo más de una vez. Vamos a comenzar con Spring Framework, y luego seguimos con EJB3. Hasta ahora nuestro código se veía algo así:

```
@Transactional(propagation=Propagation.REQUIRED)
public TradeData placeTrade(TradeData trade) throws Exception {
  try {
    insertTrade(trade);
    updateAcct(trade);
    return trade;
  } catch (Exception up) {
    //log del error
    throw up;
  }
}
```

Supongamos que la cuenta no tiene fondos suficientes para realizar la operación en cuestión, o no está habilitada todavía, y lanza una excepción chequeada (por ejemplo, un FundsNotAvailableException). ¿Se persiste la orden de compras en la base de datos, o se se hace un rollback de toda la unidad de trabajo? La respuesta, sorprendentemente, es que **sobre una excepción chequeada (tanto en Spring Framework como en EJB), se hace commit sobre la transacción**. Esto significa que si ocurre una excepción chequeada durante la ejecución del méotdo updateAcct(), la orden de compra igual será persistida, pero la cuenta no va a estar actualizada reflejando la operación.

Quizás esta sea la principal causa de los problemas de integridad y consistencia de datos. Las excepciones de Runtime (es decir, las no-chequeadas) fuerzan un rollback automático sobre toda la unidad de trabajo, pero las excepciones chequeadas no. Es por esto que, en el código anterior, la transacción es inutil; a pesar de que parece usar una transacción para mantener atomicidad y consistencia, en la realidad no lo hace.

Aunque este comportamiento pueda parecer extraño, hay varios buenos motivos para que las transacciones funcionen así. Primero, no todas las excepciones chequeadas son "malas"; pueden usarse para identificación de eventos o para redirigir el procesamiento basándose en ciertas condiciones. Pero además, el código de la aplicación podría tomar medidas correctivas en algunas excepciones chequeadas, y permitir que la transacción se complete. Por ejemplo, consideremos el escenario en donde escribimos código para un sitio de venta de libros. Para completar la orden de compra de libros, necesitamos enviar un e-mail de confirmación como parte del proceso. Si el servidor de mail está caido, se lanzará algún tipo de excepción chequeada indicando que el mensaje no pudo enviarse. Si las excepciones chequeadas causaran un rollback automático, toda la orden de compra tendría un rollback porque el servidor de mail está caido. Al no realizar un rollback automático sobre excepciones chequeadas, podemos atrapar la excepción y realizar alguna acción correctiva (como enviar el mensaje a una cola de pendientes), y realizar un commit del resto de la orden.

Cuando se usa un modelo transaccional Declarativo, debemos indicar cómo el contenedor o el framework va a manejar a las excepciones. En Spring Framework especificamos el parámetro rollbackFor en la anotación @Transactional, como vemos a continuación:

```
@Transactional(propagation=Propagation.REQUIRED, rollbackFor=Exception.class)
public TradeData placeTrade(TradeData trade) throws Exception {
```

```
  try {
    insertTrade(trade);
    updateAcct(trade);
    return trade;
  } catch (Exception up) {
    //log del error
    throw up;
  }
}
```

Veamos el uso del parámetro rollbackFor para la anotación @Transactional. Este parámetro acepta tanto una única excepción como un array de excepciones, o podemos usar el parámetro rollbackForClassName para especificar los nombres de las excepciones como String. También podemos usar la versión "negativa" de esta propiedad (noRollbackFor) para indicar que todas las excepciones fuerzan un rollback menos algunas específicas. Usualmente, la mayoría de los desarrolladores utilizan el valor Exception.class, indicando que todas las excepciones del método deben forzar un rolklback.

Los EJB funcionan algo diferente a Spring en cuanto al rollback de transacciones. La anotación @TransactionAttribute de EJB3.0 no incluye directivas para especificar el comportamiento del rollback. En cambio, se debe usar el método SessionContext.setRollbackOnly() para marcar a la transacción para hacer rollback, como vemos a continuación:

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public TradeData placeTrade(TradeData trade) throws Exception {
  try {
    insertTrade(trade);
    updateAcct(trade);
    return trade;
  } catch (Exception up) {
    //log del error
    sessionCtx.setRollbackOnly();
    throw up;
  }
}
```

Un vez que se invoca al método setRollbackOnly() no podemos cambiar de parecer; el único resultado posible es el rollback de la transacción luego de que se complete la ejecución del método que la inició.

Conclusión

El código que se usa para implementar transacciones es Java no es muy complejo; sin embargo, puede resultar complejo su utilización. Hay varios errores comunes asociados con la implementación del soporte transaccional en Java. El mayor problema es que no hay ningún aviso del compilador o en tiempo de ejecución de que la implementación de la transacción es incorrecta. Más aún, implementar el soporte de transacciones no es sólo un ejercicio de codificación. Se debe dedicar un esfuerzo importante a establecer la estrategia transaccional general.

Basado en Transaction strategies: Understanding transaction pitfalls