トランザクション・ストラテジー: モデルとストラテジーの概要

3 つのトランザクション・モデル、そしてこの 3 つのモデルを使用するトランザクション・ストラテジーについて理解する

Mark Richards
Director and Sr. Technical Architect
Collaborative Consulting, LLC

2009年 3月 03日

トランザクション・モデルをトランザクション・ストラテジーと混同するのは、よくある過ちです。連載「トランザクション・ストラテジー」の第 2 回目では、Java™ プラットフォームがサポートする 3 つのトランザクション・モデルの概要を説明し、これらのモデルを使用する 4 つの主要なトランザクション・ストラテジーを紹介します。今回、Mark Richards が Spring Framework と EJB (Enterprise JavaBeans) 3.0 仕様の例を引用しながら、それぞれのトランザクション・モデルがどのように機能するのか、そして基本的なトランザクション処理から高速トランザクション処理システムに至るまでのトランザクション・ストラテジーを作成する土台として、これらのモデルがどのような役目を果たすのかを説明します。

このシリーズの他の記事を見る

トランザクション・モデルをトランザクション・ストラテジーと混同している開発者、設計者、そしてアーキテクトはあまりにも多くいます。私はよく、顧客の下で作業を行っているアーキテクトや技術リーダーに対し、それぞれのプロジェクトのトランザクション・ストラテジーについて尋ねますが、その答えは大抵、3 通りに分けられます。1 つは、「このアプリケーションでは実際にトランザクションを使っているとは言えません」という控えめな答え、そしてもう 1 つは「その質問が何を意味するのか、はっきり理解できません」という戸惑った答えです。その一方、大抵返ってくるのは「宣言型トランザクションを使っています」という自信に満ちた答えです。しかし、この記事を読むとわかるように、宣言型トランザクションという言葉はトランザクション・モデルを意味するのであって、トランザクション・ストラテジーとはまったく違うものです。

この連載について

トランザクションはデータの品質、完全性、整合性を向上させてアプリケーションをより堅牢にします。しかし、Java アプリケーションに効果的なトランザクション処理を実装するのは容易な作業ではありません。トランザクション処理の実装には、設計がコーディングと同

じく重要になってくるからです。新しく始まったこの連載では、Mark Richards が読者の案内役となり、単純なアプリケーションからハイパフォーマンスのトランザクション処理に至るまでの様々な使用状況で功を奏するトランザクション・ストラテジーを設計する方法を説明します。

Java プラットフォームでサポートされるトランザクション・モデルには、以下の 3 つがあります。

- ローカル・トランザクション・モデル
- プログラム型トランザクション・モデル
- 宣言型トランザクション・モデル

上記のモデルは、Java プラットフォームでのトランザクションの振る舞い、そしてトランザクションの実装方法について表すものですが、トランザクション処理の規則とセマンティクス以外には何も規定していません。トランザクション・モデルをどのように適用するかは、完全に開発者次第です。例えば、`REQUIRED` と `MANDATORY` のどちらのトランザクション属性を使用するか、トランザクション・ロールバックを指示するディレクティブをいつ、どこで指定するか、プログラム型トランザクション・モデルと宣言型トランザクション・モデルの比較検討をいつするべきか、ハイパフォーマンス・システム向けにトランザクションをどのように最適化するかについてなどは、トランザクション・モデル自体が決めるわけではありません。これらの問題に対する答えを出すには、独自のトランザクション・ストラテジーを作成するか、あるいはこの記事で紹介する 4 つの基本トランザクション・ストラテジーのいずれかを導入するしかありません。

連載第 1 回目の記事でおわかりのように、トランザクションにありがちな多くの落とし穴はトランザクションの振る舞いに影響し、それによってデータの完全性と整合性が損なわれることがあります。同様に、トランザクション・ストラテジーが効果的でなかったり、トランザクション・ストラテジー自体が欠如していたりすると、データの完全性、整合性に悪影響を及ぼします。この記事で説明するトランザクション・モデルは、効果的なトランザクション・ストラテジーを作成するための基本となる要素です。そのため、それぞれのモデルの違い、そしてモデルがどのように機能するかを理解することこそが、そのモデルを使用するトランザクション・ストラテジーを理解する上で最も重要な点となります。そこで、この記事ではまず 3 つのトランザクション・モデルについて説明し、それから単純な Web アプリケーションから大規模な高速トランザクション処理システムに至るまでの広範なビジネス・アプリケーションに適用できる 4 つのトランザクション・ストラテジーを紹介したいと思います。それぞれのトランザクション・ストラテジーについての詳細は、連載「トランザクション・ストラテジー」の今後の記事で説明します。

ローカル・トランザクション・モデル

ローカル・トランザクション・モデルという名前は、トランザクションを管理するのが、アプリケーションが実行されるコンテナーやフレームワークではなく、データベース・リソース・マネージャーであるところに由来しています。このモデルでは、開発者はトランザクションではなく、接続を管理することになります。「トランザクションの落とし穴を理解する」で学んだように、Hibernate や TopLink といったオブジェクト・リレーショナル・マッピング・フレームワーク、あるいは JPA (Java Persistence API) 使ってデータベースを更新するとしたら、ローカル・トランザクション・モデルを使用することはできません。しかし、データ・アクセス・オブジェクト (DAO) を使用したり、JDBC ベースのフレームワークとデータベース・ストアード・プロシージャーを使用したりする場合にはこのモデルを適用することができます。

ローカル・トランザクション・モデルを使用するには、2 通りの方法があります。それは、データベースに接続を管理させるか、あるいは接続をプログラムによって管理するかのいずれかです。データベースに接続を管理させるには、JDBC の `Connection` オブジェクトで `autoCommit` プロパティーを `true` (デフォルト値) に設定します。この設定によって、データベース管理システム (DBMS) には、挿入、更新、削除が完了した後にトランザクションをコミットすること、あるいはこれらの操作が失敗した場合には作業をロールバックすることが指示されます。リスト 1 に、この手法を説明します。このコードは、株式取引の注文を `TRADE` テーブルに挿入します。

リスト 1. 更新が 1 回行われる場合のローカル・トランザクション

```
public class TradingServiceImpl {
   public void processTrade(TradeData trade) throws Exception {
      Connection dbConnection = null;
      try {
         DataSource ds = (DataSource)
              (new InitialContext()).lookup("jdbc/MasterDS");
         dbConnection = ds.getConnection();
         dbConnection.setAutoCommit(true);
  Statement sql = dbConnection.createStatement();
  String stmt = "insert into TRADE ...";
  sql.executeUpdate(stmt1);
      } finally {
         if (dbConnection != null)
            dbConnection.close();
      }
   }
}
```

リスト 1 では `autoCommit` の値を `true` に設定し、DBMS にデータベースを操作する各ステートメントが完了した後にローカル・トランザクションをコミットするように指示しています。この手法は、論理作業単位 (LUW) のなかで単一のデータベース保守操作を行う場合には十分有効に機能します。その一方、リスト 1 の `processTrade()` メソッドが株式取引の注文を反映して `ACCT` テーブルの口座残高も更新するとしたらどうなるでしょう。この場合、2 つのデータベース・アクションがそれぞれ単独で行われ、`ACCT` テーブルを更新する前に、`TRADE` テーブルへの挿入がデータベースにコミットされます。これでは `ACCT` テーブルの更新が失敗したとしても、`TRADE` テーブルへの挿入をロールバックするメカニズムがないため、データベース内のデータに矛盾が生じることになります。

このシナリオから、2 番目の手法であるプログラムによる接続の管理へと話がつながります。この手法では、`Connection` オブジェクトの `autoCommit` プロパティーを `false` に設定し、開発者が接続を手動でコミットまたはロールバックします。リスト 2 に、この手法を示します。

リスト 2. 更新が複数回行われる場合のローカル・トランザクション

```
public class TradingServiceImpl {
   public void processTrade(TradeData trade) throws Exception {
      Connection dbConnection = null;
      try {
         DataSource ds = (DataSource)
             (new InitialContext()).lookup("jdbc/MasterDS");
         dbConnection = ds.getConnection();
         dbConnection.setAutoCommit(false);
  Statement sql = dbConnection.createStatement();
  String stmt1 = "insert into TRADE ...";
  sql.executeUpdate(stmt1);
```

```
  String stmt2 = "update ACCT set balance...";
  sql.executeUpdate(stmt2);
  dbConnection.commit();
      } catch (Exception up) {
         dbConnection.rollback();
         throw up;
      } finally {
         if (dbConnection != null)
            dbConnection.close();
      }
   }
}
```

リスト 2 では `autoCommit` プロパティーを `false` に設定することで、基礎となる DBMS に対し、接続はデータベースでなくコード内で管理されることを通知しています。この場合、すべてが上手くいったら `Connection` オブジェクトで `commit()` メソッドを呼び出し、例外が発生したら `rollback()` メソッドを呼び出すことになります。このようにして、2 つのデータベース操作を同じ作業単位内で連携させることができます。

ローカル・トランザクション・モデルは今となっては多少、時代遅れのモデルのように思えるかもしれませんが、この記事の終わりのほうで紹介する主要なトランザクション・ストラテジーの 1 つにとっては、重要な要素となります。

プログラム型トランザクション・モデル

プログラム型トランザクション・モデルの名前の由来となっているのは、開発者がトランザクションの管理を行うという点です。ローカル・トランザクション・モデルとは異なり、プログラム型トランザクション・モデルでは開発者がトランザクションを管理するため、データベース接続とは切り離されます。

このモデルではリスト 2 の例と同様に、トランザクション・マネージャーからトランザクションを取得するのも、トランザクションを開始、コミットするのも、そして例外が発生した場合にトランザクションをロールバックするのも、すべて開発者の役目です。そのためご想像の通り、このモデルにはエラーを起こしがちなコードが溢れることになり、アプリケーションのビジネス・ロジックが妨害されがちです。とは言うものの、トランザクション・ストラテジーのなかには、このプログラム型トランザクション・モデルを使用しなければならないものもあります。

Spring Framework と EJB 3.0 仕様とでは、概念は同じとは言え、それぞれに異なるプログラム型トランザクション・モデルを実装します。このモデルの実装について、まずは EJB 3.0 を使用して説明し、次に同じデータベース更新を Spring Framework を使用して行う場合を説明します。

EJB 3.0 でのプログラム型トランザクション

EJB 3.0 でトランザクション・マネージャー (つまり、コンテナー) からトランザクションを取得するには、`javax.transaction.UserTransaction` で JNDI (Java Naming and Directory Interface) ルックアップを行います。`UserTransaction` を取得した後は、トランザクションを開始する `begin()` メソッド、トランザクションをコミットする `commit()` メソッド、そしてエラーが発生した場合にトランザクションをロールバックする `rollback()` メソッドを呼び出すことができるようになります。このモデルではコンテナーがトランザクションを自動的にコミットまたはロールバックすることはないため、データベースを更新する Java メソッドにこの振る舞いをプログラムするかどう

かは開発者次第です。リスト 3 に、JPA を使用した EJB 3.0 でのプログラム型トランザクション・モデルの一例を示します。

リスト 3. EJB 3.0 を使用したプログラム型トランザクション

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class TradingServiceImpl implements TradingService {
   @PersistenceContext(unitName="trading") EntityManager em;

   public void processTrade(TradeData trade) throws Exception {
      InitialContext ctx = new InitialContext();
      UserTransaction txn = (UserTransaction)ctx.lookup("UserTransaction");
      try {
         txn.begin();
  em.persist(trade);
  AcctData acct = em.find(AcctData.class, trade.getAcctId());
  double tradeValue = trade.getPrice() * trade.getShares();
  double currentBalance = acct.getBalance();
  if (trade.getAction().equals("BUY")) {
     acct.setBalance(currentBalance - tradeValue);
         } else {
            acct.setBalance(currentBalance + tradeValue);
         }
         txn.commit();
      } catch (Exception up) {
         txn.rollback();
         throw up;
      }
   }
}
```

ステートレス・セッション Bean を使用する Java EE (Java Platform, Enterprise Edition) コンテナー環境でプログラム型トランザクション・モデルを適用する場合には、コンテナーに、プログラム型トランザクションを使用していることを伝える必要があります。それには `@TransactionManagement` アノテーションを使ってトランザクション・タイプを `BEAN` に設定します。このアノテーションを使用しない場合は、コンテナーは EJB 3.0 のデフォルト・トランザクション・タイプである宣言型トランザクション管理 (`CONTAINER`) が使用されるという前提の下で処理を行います。ステートレス・セッション Bean のコンテキストには含まれないクライアント層でプログラム型トランザクションを使用するときには、トランザクション・タイプを設定する必要はありません。

Spring でのプログラム型トランザクション

Spring Framework では、2 通りの方法でプログラム型トランザクション・モデルを実装することができます。1 つは Spring `TransactionTemplate` によって実装する方法、そしてもう 1 つは Spring のプラットフォーム・トランザクション・マネージャーを直接使用するという方法です。匿名内部クラスや難解なコードをあまり好まない私としては、2 番目の手法を用いて Spring でのプログラム型トランザクション・モデルを説明することにします。

Spring には少なくとも 9 つのプラットフォーム・トランザクション・マネージャーがあります。なかでもとりわけよく使用することになるのは、`DataSourceTransactionManager`、`HibernateTransactionManager`、`JpaTransactionManager`、`JtaTrans`す。この記事のサンプル・コードでは JPA を使用しているので、`JpaTransactionManager` を構成する場合を説明します。

Spring で `JpaTransactionManager` を構成するには、ただ単に、アプリケーション・コンテキストの XML ファイルに `org.springframework.orm.jpa.JpaTransactionManager` クラスを使って Bean を定義し、JPA の Entity Manager Factory Bean への参照を追加するだけです。Spring がアプリケーション・ロジックを含むクラスを管理する場合には、この Bean にトランザクション・マネージャーを注入します (リスト 4 を参照)。

リスト 4. Spring JPA トランザクション・マネージャーの定義

```
<bean id="transactionManager"
         class="org.springframework.orm.jpa.JpaTransactionManager">
   <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>

<bean id="tradingService" class="com.trading.service.TradingServiceImpl">
   <property name="txnManager" ref="transactionManager"/>
</bean>
```

Spring がアプリケーション・クラスを管理しないのであれば、Spring コンテキストで `getBean()` メソッドを使用することによって、メソッドのなかでトランザクション・マネージャーへの参照を取得することができます。

これで、ソース・コードでプラットフォーム・マネージャーを使ってトランザクションを取得できるようになりました。すべての更新が実行されたら、`commit()` メソッドを呼び出してトランザクションをコミットするか、または `rollback()` メソッドを呼び出してトランザクションをロールバックすることができます。リスト 5 に、この手法を示します。

リスト 5. Spring JPA トランザクション・マネージャーの使用

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

   JpaTransactionManager txnManager = null;
   public void setTxnManager(JpaTransactionManager mgr) {
      txnManager = mgr;
   }

   public void processTrade(TradeData trade) throws Exception {
      TransactionStatus status =
         txnManager.getTransaction(new DefaultTransactionDefinition());
      try {
         em.persist(trade);
  AcctData acct = em.find(AcctData.class, trade.getAcctId());
  double tradeValue = trade.getPrice() * trade.getShares();
  double currentBalance = acct.getBalance();
  if (trade.getAction().equals("BUY")) {
     acct.setBalance(currentBalance - tradeValue);
  } else {
     acct.setBalance(currentBalance + tradeValue);
  }
         txnManager.commit(status);
      } catch (Exception up) {
         txnManager.rollback(status);
         throw up;
      }
   }
}
```

リスト 5 のコードでは、Spring Framework と EJB 3.0 との違いに注目してください。Spring の場合、プラットフォーム・トランザクション・マネージャーで `getTransaction()` メソッドを呼び出すことによって、トランザクションを取得します (そして結果的に、トランザクションを開始します)。匿名 `DefaultTransactionDefinition` クラスには、トランザクションの名前、分離レベル、伝播モード (トランザクション属性)、そしてトランザクションのタイムアウト (該当する場合) を含め、トランザクションとその振る舞いについての詳細が含まれます。そのため上記では単に、名前のデフォルト値として空のストリング、DBMS にはデフォルト分離レベル (通常は `READ_COMMITTED`)、トランザクション属性には `PROPAGATION_REQUIRED`、そして DBMS のデフォルト・タイムアウトを使用しています。さらに、EJB での場合のようにトランザクションを呼び出すのではなく、プラットフォーム・トランザクション・マネージャーを使用して `commit()` メソッドと `rollback()` メソッドを呼び出していることにも注目してください。

宣言型トランザクション・モデル

CMT (Container Managed Transactions) としても知られる宣言型トランザクション・モデルは、Java プラットフォームでは最もよく使用されるトランザクション・モデルです。このモデルでは、コンテナー環境がトランザクションの開始、コミット、ロールバックを処理するため、開発者の役目はトランザクションの振る舞いを指定することだけとなります。この連載の最初の記事で説明したトランザクションの落とし穴の多くは、この宣言型トランザクション・モデルに関連するものです。

Spring Framework と EJB 3.0 ではどちらもアノテーションを利用して、トランザクションの振る舞いを指定します。Spring では使用するのは @Transactional アノテーション、EJB 3.0 で使用するのは `@TransactionAttribute` アノテーションです。宣言型トランザクション・モデルを使用する場合、コンテナーがチェック例外の時点で自動的にトランザクションをロールバックすることはありません。そのため開発者が、チェック例外の発生時にトランザクションをロールバックする場所とタイミングを指定する必要があります。Spring Framework でこれを指定するには、`@Transactional` アノテーションで `rollbackFor` プロパティーを使用します。一方、EJBでは `SessionContext` で `setRollbackOnly()` メソッドを呼び出して指定します。

リスト 6 に、EJB での宣言型トランザクション・モデルの使用方法を示します。

リスト 6. EJB 3.0 を使用した宣言型トランザクション

```
@Stateless
public class TradingServiceImpl implements TradingService {
   @PersistenceContext(unitName="trading") EntityManager em;
   @Resource SessionContext ctx;

   @TransactionAttribute(TransactionAttributeType.REQUIRED)
   public void processTrade(TradeData trade) throws Exception {
      try {
         em.persist(trade);
  AcctData acct = em.find(AcctData.class, trade.getAcctId());
  double tradeValue = trade.getPrice() * trade.getShares();
  double currentBalance = acct.getBalance();
  if (trade.getAction().equals("BUY")) {
     acct.setBalance(currentBalance - tradeValue);
  } else {
     acct.setBalance(currentBalance + tradeValue);
  }
      } catch (Exception up) {
```

```
            ctx.setRollbackOnly();
            throw up;
         }
      }
   }
```

リスト 7 は、Spring Framework での宣言型トランザクション・モデルの使用方法です。

リスト 7. Spring を使用した宣言型トランザクション

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

   @Transactional(propagation=Propagation.REQUIRED,
                  rollbackFor=Exception.class)
   public void processTrade(TradeData trade) throws Exception {
      em.persist(trade);
      AcctData acct = em.find(AcctData.class, trade.getAcctId());
      double tradeValue = trade.getPrice() * trade.getShares();
      double currentBalance = acct.getBalance();
      if (trade.getAction().equals("BUY")) {
         acct.setBalance(currentBalance - tradeValue);
      } else {
         acct.setBalance(currentBalance + tradeValue);
      }
   }
}
```

トランザクション属性

ロールバックを指示するディレクティブに加え、トランザクションの振る舞いを定義するトランザクション属性も指定する必要があります。Java プラットフォームでは、EJB と Spring Framework のどちらを使用しているかに関わらず、以下の 6 つのタイプのトランザクション属性をサポートします。

- `Required`
- `Mandatory`
- `RequiresNew`
- `Supports`
- `NotSupported`
- `Never`

上記のトランザクション属性を `methodA()` という架空のメソッドに適用して、これらのトランザクション属性それぞれの特徴を説明します。

`methodA()` に `Required` トランザクション属性が指定されている場合、既存のトランザクションのスコープで `methodA()` が呼び出されると、このメソッドには既存のトランザクションのスコープが使用されます。トランザクション・コンテキストが存在しなければ、`methodA()` が新しいトランザクションを開始します。`methodA()` によって開始されたトランザクションは、同じく `methodA()` によって終了 (コミットまたはロールバック) されなければなりません。これは最もよく使用されるトランザクション属性で、EJB 3.0 と Spring 両方でのデフォルトになっていますが、残念ながら正しく使用されていない場合がほとんどです。そのため、データの完全性と整合性に問題をもたらす結果となっています。このトランザクション属性については、今後の記事で説明するトランザクション・ストラテジーのそれぞれで、その使用方法を詳しく説明します。

methodA() に Mandatory トランザクション属性が指定されている場合、既存のトランザクションのスコープで methodA() が呼び出されると、このメソッドには既存のトランザクションのスコープが使用されます。ここまでは上記のトランザクションと同じですが、methodA() が呼び出されたときにトランザクション・コンテキストが存在しなければ、TransactionRequiredException がスローされます。これは、methodA() を呼び出す前にトランザクションが存在していなければならないことを示す例外です。このトランザクション属性は、次のセクションで説明するクライアント・オーケストレーションというトランザクション・ストラテジーで使用されます。

RequiresNew トランザクション属性は興味深い属性で、ほとんどの場合、この属性は誤って使用されていたり、誤解されていたりします。methodA() にこの RequiresNew トランザクション属性が指定されている場合、methodA() が呼び出されると、トランザクション・コンテキストの有無に関わらず methodA() が常に新しいトランザクションを開始 (そして終了) します。これはつまり、methodA() が別のトランザクション (ここでは Transaction1 と呼びます) のコンテキスト内で呼び出されると、Transaction1 は中断され、新しいトランザクション (ここでは Transaction2 と呼びます) が開始されるということです。methodA() が終了して Transaction2 がコミットまたはロールバックされると、Transaction1 が再開します。これは明らかに、トランザクションが持つ ACID (原子性 (アトミック性)、一貫性 (整合性)、独立性、耐久性) 特性 (特にアトミック性) に反します。言い換えると、すべてのデータベース更新がもはや、単一の作業単位に含まれなくなるということです。Transaction1 がロールバックされたとしても、Transaction2 によってコミットされた変更はコミットされたままだというのに、それでもこのトランザクション属性を使う利点はあるのでしょうか。連載の最初の記事で指摘したように、このトランザクション属性を使用するのは、トランザクション (この例の場合は Transaction1) とは独立したデータベース操作 (監査やロギングなど) に限らなければなりません。

Supports トランザクション属性も、ほとんどの開発者が十分に理解または評価していない属性だと思います。Supports トランザクション属性を指定した methodA() が既存のトランザクションのスコープで呼び出されると、methodA() はそのトランザクションのスコープで実行されます。その一方、トランザクション・コンテキストが存在しないなかで methodA() が呼び出された場合には、トランザクションは 1 つも開始されません。この属性は主に、データベースに対する読み取り専用操作で使用されますが、それではなぜ、この属性の代わりに、トランザクションが存在しなくてもメソッドが実行されることを保証する NotSupported トランザクション属性 (次のパラグラフで説明) を指定しないのでしょうか。その答えは簡単です。既存のトランザクションのコンテキストでクエリー操作を呼び出すと、データベースのトランザクション・ログからデータ (すなわち、更新後のデータ) が読み出されることになりますが、トランザクション・スコープを使用せずにクエリーを実行すると、テーブルから変更されていないままのデータが読み出されるからです。例えば、新しい株式取引の注文を TRADE テーブルに挿入してから、(同じトランザクションで) すべての株式取引の注文の一覧を取得すると、コミットされていないその株式取引も一覧に表示されます。その一方、NotSupported トランザクション属性のような属性を使用した場合には、データベース・クエリーがトランザクション・ログではなくテーブルから読み取ることになります。したがって、まだコミットされてない株式取引は一覧には表示されません。これが必ずしも問題になるということではなく、要は使用状況とビジネス・ロジック次第ということです。

NotSupported トランザクション属性は、トランザクションの有無に関わらず、呼び出し対象のメソッドがトランザクションを使用することも、開始することもないように指定します。methodA() に NotSupported トランザクション属性が指定されている場合、トランザクション・コンテキス

トが存在するなかで `methodA()` が呼び出されると、そのトランザクションは `methodA()` が終了するまで中断されます。`methodA()` が終了すると、元のトランザクションが再開されます。このトランザクション属性を使用する場合は限られていますが、これが関係するのは主に、データベース・ストアード・プロシージャーです。例えば、既存のトランザクション・コンテキストのスコープ内でデータベース・ストアード・プロシージャーを呼び出すとします。この場合、そのデータベース・ストアード・プロシージャーに `BEGIN TRANS` が含まれていたり、(Sybase の場合に) データベース・ストアード・プロシージャーが非連鎖モードで実行されたりすると、既存のトランザクションがあるために新しいトランザクションを開始できなかったことを示す例外がスローされます (つまり、ネストされたトランザクションはサポートされないということです)。ほとんどすべてのコンテナーは、JTA のデフォルト・トランザクション実装として JTS (Java Transaction Service) を使用します。ネストされたトランザクションをサポートしないのは、Java プラットフォーム自体ではなく、この JTS です。データベース・ストアード・プロシージャーを変更できないとしたら、この致命的例外を避けるには `NotSupported` 属性を使用して既存のトランザクション・コンテキストを中断するという手段があります。ただしその影響として、同じ LUW でデータベースに対するアトミックな更新は使用できなくなります。こうしたトレード・オフがあるものの、このトランザクション属性が困難な状況を簡単に抜け出す手段となります。

`Never` トランザクション属性は、今まで説明したなかで最も興味深い属性でしょう。振る舞いは `NotSupported` トランザクション属性と同じですが、重要な違いが 1 つあります。`Never` トランザクション属性を使用してメソッドを呼び出したときに、既存のトランザクション・コンテキストがあると、そのメソッドの呼び出し時にはトランザクションが存在していてはいけないことを示す例外がスローされます。今まで、このトランザクション属性に考え付いた唯一の用途はテストです。このトランザクション属性を使用すれば、特定のメソッドを呼び出すときにトランザクションが存在するかどうかを素早く簡単に確認することができます。`Never` トランザクション属性を使用して対象のメソッドを呼び出したときに例外を受け取ったとしたら、トランザクションが存在していたことになります。メソッドが実行できた場合には、トランザクションが存在してなかったことがわかります。これは、トランザクション・ストラテジーが確かなものであることを保証するのに絶好の方法です。

トランザクション・ストラテジー

この記事でこれまでに説明したトランザクション・モデルは、これから紹介するトランザクション・ストラテジーの基礎となります。トランザクション・ストラテジーの作成に取り掛かる前に、肝心なこととして、モデルそれぞれの違いについて、そして各モデルがどのように機能するのかを十分に理解してください。以下の基本的なトランザクション・ストラテジーは、ほとんどのビジネス・アプリケーション・シナリオで使用することができるものです。

- クライアント・オーケストレーション・トランザクション・ストラテジー
- API 層トランザクション・ストラテジー
- 高並行性トランザクション・ストラテジー
- 高速処理トランザクション・ストラテジー

今回は上記のストラテジーのそれぞれを要約し、詳細については連載の今後の記事で説明します。

クライアント・オーケストレーション・トランザクション・ストラテジーは、クライアント層からのサーバーまたはモデルをベースとした複数の呼び出しが単一の作業単位になる場合に使用し

ます。その際、クライアント層は Web フレームワーク、ポータル・アプリケーション、デスクトップ・システム、あるいは場合によってはワークフロー製品やビジネス・プロセス管理 (BPM) のコンポーネントからの呼び出しを参照することができます。特定のリクエストを完了するために必要な処理フローと「ステップ」を所有するのは、基本的にクライアント層です。例えば株式取引の注文をするために、その取引をデータベースに挿入し、それから顧客の口座残高を更新してその取引の値を反映する必要があるとします。アプリケーションの API 層があまりにも細分化されている場合には、クライアント層から挿入と更新両方のメソッドを呼び出す必要がありますが、このようなシナリオでアトミックな作業単位を確実にするためには、トランザクションの作業単位がクライアント層になければなりません。

API 層トランザクション・ストラテジーは、メソッドの細分度が低く、メソッドがバックエンド機能への 1 次エントリー・ポイントとして機能する場合に使用します (これらのメソッドは、サービスと呼ぶこともできます)。このシナリオでは、クライアント (Web ベース、Web サービス・ベース、またはメッセージ・ベースのクライアント、あるいはデスクトップの場合もあります) がバックエンドに対して単一の呼び出しを行って特定のリクエストを実行します。前のパラグラフで取り上げた株式取引を注文するシナリオを引用すると、このストラテジーでは、クライアント層が呼び出すのは単一のエントリー・ポイント・メソッド (例えば、`processTrade()` というメソッド) となるため、この単一のメソッドに、株式取引の注文を挿入し、口座を更新するために必要なオーケストレーションが含まれることになります。私がこのストラテジーにこの名前を付けた理由は、バックエンドの処理機能はほとんどの場合、インターフェース、つまり API によってクライアント・アプリケーションに公開されるためです。これは、最もよく使われるトランザクション・ストラテジーの 1 つとして数えられます。

高並行性トランザクション・ストラテジーは API 層トランザクション・ストラテジーの変形で、これは、長時間実行されるトランザクションを (通常は、パフォーマンスまたはスケーラビリティーが必要になるため) API 層からサポートすることができないアプリケーションに使用します。その名前が示唆するように、高並行性トランザクション・ストラテジーは主に、ユーザーの観点から高度な並行性をサポートするアプリケーションで使用されます。トランザクションは、Java プラットフォームではかなりコストがかかるものです。使用しているデータベースによっては、トランザクションがデータベース内でのロックの原因となる場合や、リソースを占有してしまう場合、スループットの点からアプリケーションの速度を低下させる場合もあります。さらに、データベースでデッドロックを発生させることにもなりかねません。このトランザクション・ストラテジーの中心となる考えは、トランザクション・スコープを狭めることによって、データベース内でのロックを最小限にすると同時に、あらゆるクライアントのリクエストに対してアトミックな作業単位を維持することです。場合によっては、このトランザクション・ストラテジーをサポートするためにアプリケーション・ロジックをリファクタリングしなければならないこともあります。

高速処理トランザクション・ストラテジーはおそらく最も極端なトランザクション・ストラテジーでしょう。このトランザクション・ストラテジーを使用するのは、アプリケーションの処理時間を最大限に高速化 (したがってスループットも高速化) するとともに、トランザクションの処理において、ある程度のアトミック性を維持する必要がある場合です。このストラテジーはデータの完全性および整合性の点で多少のリスクをもたらしますが、正しく実装すれば、Java プラットフォームで考えられる最も高速なトランザクション・ストラテジーとなります。また、ここで

紹介した 4 つのトランザクション・ストラテジーのなかで、実装するのが最も困難で厄介なのも、おそらくこの高速処理トランザクション・ストラテジーです。

まとめ

以上の概要からおわかりのように、効果的なトランザクション・ストラテジーの作成は、必ずしも簡単なタスクではありません。データの完全性、整合性の問題を解決するためには、さまざまな検討事項、オプション、モデル、フレームワーク、構成、そして手法が関係してきます。私が長年アプリケーションとトランザクションに取り組んできてわかったことは、モデル、オプション、設定、構成の組み合わせは気の遠くなるほど数が多いように思えても、ほとんどの使用状況で理にかなったオプションと設定の組み合わせは、実際にはほんの少数しかないということです。連載の今後の記事では、私が作成した 4 つのトランザクション・ストラテジーについて詳しく説明していきますが、この 4 つのトランザクション・ストラテジーが Java プラットフォームで考えられるビジネス・アプリケーション開発のシナリオのほとんどをカバーするはずです。ただし 1 つだけ注意しておきますが、これらのストラテジーは「特効薬」としてそのまま採用すればよいだけのソリューションではありません。場合によっては、ソリューションとして実装するためにソース・コードのリファクタリングやアプリケーションの再設計が必要になる可能性があります。そのような状況になったら、単に「このデータの完全性と整合性はどれだけ重要なのか」を考えてみてください。ほとんどの場合、不良なデータによるリスクとコストに比べれば、リファクタリングの苦労も厭わないと思うはずです。

著者について

Mark Richards

Mark Richards は、Collaborative Consulting, LLC のディレクター兼シニア・テクニカル・アーキテクトです。彼は『Java Message Service』(O'Reilly、2009年) の第 2 版、そして『Java Transaction Design Strategies』(C4Media Publishing、2006年) の第 2 版の著者であり、寄稿者としても『97 Things Every Software Architect Should Know』(O'Reilly、2009年)、『NFJS Anthology Volume 1』(Pragmatic Bookshelf、2006年)、『NFJS Anthology Volume 2』(Pragmatic Bookshelf、2007年) などの本に貢献しています。IBM、Sun、The Open Group、そして BEA の認定アーキテクトおよび技術者である彼は No Fluff Just Stuff Symposium Series ではお馴染みの講演者で、世界各地のその他のカンファレンスやユーザー・グループでも講演を行っています。