トランザクション・ストラテジー: トランザクションの落とし穴を理解する

Java プラットフォームにトランザクションを実装するときにありがちな過ちに用心してください

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

2009年 2月 03日

トランザクション処理では、データの完全性と整合性が高いレベルで実現されなければなりません。この記事では、Java プラットフォームでの効果的なトランザクション・ストラテジーの開発に関する連載の第 1 回目として、データの完全性と整合性を高いレベルで実現することの妨げとなる、よくありがちなトランザクションの落とし穴を紹介します。連載の著者、Mark Richards が、Spring Framework および EJB (Enterprise JavaBeans) 3.0 仕様のサンプル・コードを使用して、よくありがちな過ちについて説明します。

このシリーズの他の記事を見る

アプリケーションでトランザクションを使用する最も一般的な理由は、データの完全性と整合性を高いレベルで維持するためです。データの品質に無関心であれば、トランザクションに関心を持つ必要もありません。結局のところ、Java プラットフォームでのトランザクション・サポートはパフォーマンスを犠牲にし、ロックの問題やデータベースの並行性に関する問題を引き起こし、アプリケーションの複雑さを増す結果となる可能性があります。

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。

しかしトランザクションに関心を持たない開発者は、同時にそれ相当の危険も覚悟しなければなりません。ビジネス関連のほとんどすべてのアプリケーションには、高いレベルのデータ品質が要求されます。金融資産投資業界だけを例にとっても、何百億ドルの損失を出している取引失敗の原因として第 2 位に挙げられるのは、不良データです (「参考文献」を参照)。トランザクション・サポートの欠如は不良データにつながる要因の 1 つでしかありませんが (ただし、主な要因

です)、金融資産投資業界での数百億ドルの損失は、トランザクションがサポートされていないこと、あるいはトランザクション・サポートが十分ではないことの結果だと推測するのが妥当です。

トランザクション・サポートに関する知識がないことも、同じく問題の原因となっています。「アプリケーションに障害が発生することはあり得ないので、トランザクション・サポートは必要ない」という主張はあまりにもよく耳にします。実際、その主張通り、めったに例外をスローしないアプリケーションや、まったく例外をスローしないアプリケーションを見たこともあります。これらのアプリケーションでは、トランザクション処理に伴うパフォーマンス上の犠牲や複雑さを避けるために、十分に検討して作成されたコードと検証ルーチン、そして完全なテストとコード・カバレッジのサポートを頼りしているわけですが、このような考え方で問題となるのは、トランザクション・サポートの 1 つの特性、つまりアトミック性しか考慮していないという点です。アトミック性は、すべての更新がまとめて 1 つの単位として処理されるように、すべての更新がコミットされるか、あるいはすべてロールバックされるかのいずれかの結果になるようにします。しかし、更新をロールバックまたは統合することだけが、トランザクション・サポートの側面ではありません。1 つの作業単位が確実に他の作業単位から分離されるようにする、分離という別の側面もあります。トランザクションの分離が適切に行われていなければ、進行中の作業単位がまだ完了してないとしても、他の作業単位がこの作業単位で行われた更新にアクセスできてしまいます。そうなると、部分的なデータをベースにしてビジネス決定が行われる可能性があり、それが原因で取引が失敗するなどの良くない (あるいはコストの高い) 結果に至る恐れがあります。

たとえ遅くても、何もしないよりはましです

私がトランザクション処理に伴う問題の大きさを理解し始めたのは、2000年の初めです。その当時、クライアントのサイトで作業していた私は、プロジェクト計画のシステム・テスト・タスクの直前に 1 つの項目があることに気付きました。その項目は、トランザクション・サポートの実装と書かれていました。主要なアプリケーションのシステム・テストを実行する準備がほぼ整った時点で、そのアプリケーションにトランザクション・サポートを追加するのはもちろん簡単ですが、残念なことに、あまりにもこの手法が使われ過ぎています。それでも多くのプロジェクトとは違い、少なくともこのプロジェクトでは開発サイクルの終わりだったとは言え、トランザクション・サポートを実装していました。

トランザクションが重要 (かつ必要) であるという基本知識に基づき、不良データによってもたらされる高いコストと悪影響を考えると、トランザクションを使用し、それによって発生し得る問題に対処する方法を学ばなければならないという必要に迫られます。そこで余儀なくトランザクション・サポートをアプリケーションに追加することになりますが、多くの場合、問題はここから始まります。Java プラットフォームでは、トランザクションが必ずしもその見込み通りに有効に機能するとは限らないからです。この記事ではその理由を探るため、サンプル・コードの助けを借りて、よくあるトランザクションの落とし穴をいくつか取り上げて説明します。この記事で説明する落とし穴は、私が現場でよく目にしたり、経験したりしたもので、そのほとんどは本番環境に潜んでいます。

この記事に記載するサンプル・コードのほとんどでは Spring Framework (バージョン 2.5) を使用していますが、トランザクションの概念は EJB 3.0 仕様の場合でも変わりません。大抵は、Spring Framework の `@Transactional` アノテーションが、EJB 3.0 仕様では `@TransactionAttribute` アノテーションに置き換わるだけの話です。この 2 つのフレームワークが概念と手法の点で異なる場合には、Spring Framework と EJB 3.0 両方のサンプル・コードを記載しています。

ローカル・トランザクションの落とし穴

最初は、最もわかりやすいシナリオである、ローカル・トランザクションを使用する例から始めるのがよいでしょう。ローカル・トランザクションは別名、データベース・トランザクションとも呼ばれています。初期のデータベース・パーシスタンス (JDBC など) では、トランザクション処理をデータベースに委任するのが一般的でした。結局のところ、それがデータベースに意図されている役割なのではないでしょうか。ローカル・トランザクションは、単独で insert 文、update 文、delete 文を実行する論理作業単位 (LUW) には有効に機能します。例えば、リスト 1 の単純な JDBC コードを見てください。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. JDBC による単純なデータベース挿入操作

```
@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();
    }
  }
}
```

リスト 1 の JDBC コードにはトランザクション・ロジックが含まれていないものの、データベース内の `TRADE` テーブルに取引注文を保存します。この場合、データベースがトランザクション・ロジックを処理します。

LUW の単一のデータベース保守アクションに対しては、これでまったく問題ありません。けれども取引注文をデータベースに挿入すると同時に、口座残高も更新しなければならないとしたらどうでしょう。リスト 2 はその一例です。

リスト 2. 同じ 1 つのメソッドで何度かテーブルを更新する場合

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

上記の場合、insertTrade() メソッドと updateAcct() メソッドはトランザクションを使用せずに標準 JDBC コードを使用します。insertTrade() メソッドが実行されると、データベースには取引注文が保存された (そしてコミットされた) 状態になっています。この後、updateAcct() メソッドが何らかの理由で失敗した場合には、placeTrade() メソッドの終了時に取引注文がまだ TRADE テーブルに残ることになるため、それによってデータベース内のデータの整合性がなくなります。一方、placeTrade() メソッドがトランザクションを使用するとしたら、両方のアクティビティーが 1 つの LUW に組み込まれ、残高更新の失敗によって取引注文はロールバックされます。

Hibernate や TopLink などの Java パーシスタンス・フレームワーク、そして JPA (Java Persistence API) がよく使われるようになっている現在では、単純な JDBC コードを作成することはまれになってきました。最近では作業を楽にするために、新しいオブジェクト・リレーショナル・マッピング (ORM) フレームワークを使って、厄介な JDBC コードのすべてをいくつかの単純なメソッド呼び出しに置き換えるほうが一般的です。リスト 1 の JDBC サンプル・コードを例に取ると、取引注文を挿入するには、Spring Framework を JPA と一緒に使用して TradeData オブジェクトを TRADE テーブルにマッピングし、JDBC コード全体をリスト 3 の JPA コードに置換することになります。

リスト 3. JPA を使用した単純な挿入操作

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

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

リスト 3 では、persist() メソッドを EntityManager で呼び出して、取引注文を挿入していることに注目してください。単純なことのようですが、実はそうでもありません。このコードは期待通りには動作せず、取引注文が TRADE テーブルに挿入されないどころか、例外もスローされません。コードはデータベースを変更せずに、単に取引注文へのキーとして 0 の値を返すことしかしません。これが、トランザクション処理で最初に遭遇する大きな落とし穴の 1 つです。つまり、ORM ベースのフレームワークには、オブジェクト・キャッシュとデータベースとが同期した状態にするためにトランザクションが必要だということです。トランザクションのコミットによって、SQL コードが生成されてデータベースに目的のアクション (つまり、挿入、更新、削除) が作用します。トランザクションが使用されなければ、ORM が SQL コードを生成して変更を保存するトリガーとなるものがないため、メソッドはただ単に終了するに過ぎず、例外もスローされなければ、更新も行われません。ORM ベースのフレームワークを使用する場合には、トランザクション

を必ず使用してください。データベースによる接続の管理や作業のコミットには、もはや依存することはできません。

上記の単純な例から、データの完全性と整合性を維持するためにはトランザクションが必要であることは明らかです。しかしこれは、Java プラットフォームにトランザクションを実装する上で伴う複雑さと落とし穴のほんの一端にしか過ぎません。

Spring Framework の @Transactional アノテーションの落とし穴

リスト 3 のコードをテストすると、persist() メソッドはトランザクションなしでは機能しないことがわかります。そこで、手軽なインターネット検索でいくつかのリンクを調べてみると、Spring Framework の場合には @Transactional アノテーションを使用する必要があるとあります。それに従って、コードにリスト 4 のようにアノテーションを追加したとします。

リスト 4. @Transactional アノテーションの使用

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

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

もう一度コードをテストしてみると、やはりコードは機能しません。問題は、Spring Framework に対して、トランザクション管理にアノテーションを使用していることを伝えなければならないという点です。完全なユニット・テストを行っているのでない限り、この落とし穴はなかなか見つからないものです。開発者がこの点を見落として、アノテーションを介さずに Spring 構成ファイルにそのままトランザクション・ロジックを追加することは珍しくありません。

Spring で @Transactional アノテーションを使用するときには、Spring 構成ファイルに以下の行を追加する必要があります。

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

transaction-manager プロパティーが保持するのは、Spring 構成ファイルに定義されたトランザクション・マネージャー Bean への参照です。このコードは、トランザクション・インターセプターを適用するときに @Transaction アノテーションを使用するように Spring に指示します。この指示がなければ @Transactional アノテーションは無視され、コード内でまったくトランザクションが使用されないという結果になってしまいます。

基本的な @Transactional アノテーションをリスト 4 のコードで機能させるという作業は、まだ序の口に過ぎません。リスト 4 で注意しなければならない点は、追加のアノテーション・パラメーターを何も指定せずに @Transactional アノテーションを使用していることです。私が気付いたところでは、多くの開発者は @Transactional アノテーションの役割を十分に理解せずに、このアノテーションを使用しています。例えば、リスト 4 のように @Transactional アノテーションを単独で使用した場合、トランザクションの伝播モードが何に設定されるかわかりますか？読み取り専用フラグ、トランザクション分離レベルはそれぞれどのように設定されますか？さらに重要な点とし

て、トランザクションが作業をロールバックする条件は何ですか？アプリケーションで適切なレベルのトランザクション・サポートを確実にするには、このアノテーションの使い方を理解することが重要です。上記の質問に対する答えを明らかにすると、@Transactional アノテーションを単独で使用してパラメーターを何も指定しない場合、伝播モードは REQUIRED に設定されます。読み取り専用フラグは false に、トランザクション分離レベルは READ_COMMITTED に設定されます。そして、トランザクションはチェック例外が発生してもロールバックしません。

@Transactional 読み取り専用フラグの落とし穴

作業中によく陥りがちな落とし穴は、Spring の @Transactional アノテーションで、読み取り専用フラグに関して不適切な使い方をすることです。ここで簡単なクイズに答えてみてください。Java パーシスタンスに標準 JDBC コードを使用しているときに、読み取り専用フラグを true に設定し、伝播モードを SUPPORTS に設定すると、リスト 5 の @Transactional アノテーションはどのように作用するでしょうか。

リスト 5. SUPPORTS 伝播モードで読み取り専用を使用する場合 — JDBC

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

リスト 5 の insertTrade() メソッドが実行されたときの結果は、以下のうちのどれですか。

- 読み取り専用接続例外がスローされる
- 取引注文が正しく挿入され、データがコミットされる
- 伝播レベルが SUPPORTS に設定されているため何も行われない

お手上げですか？正解は B です。読み取り専用フラグが true に設定されていて、トランザクション伝播モードが SUPPORTS に設定されているとしても、取引注文は正しくデータベースに挿入されます。その理由を説明すると、SUPPORTS 伝播モードではトランザクションが開始されないため、メソッドは実質上、ローカル (データベース) トランザクションを使用することになるからです。読み取り専用フラグが適用されるのは、トランザクションが開始された場合のみです。上記の例ではトランザクションは開始されなかったため、読み取り専用フラグは無視されます。

では、もしこの説明のとおりであれば、今度は読み取り専用フラグを設定して伝播モードを REQUIRED に設定した場合、リスト 6 の @Transactional アノテーションはどのように作用するでしょうか。

リスト 6. REQUIRED 伝播モードで読み取り専用を使用する場合 — JDBC

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

リスト 6 の insertTrade() メソッドが実行されたときの結果は、以下のうちのどれですか。

- 読み取り専用接続例外がスローされる
- 取引注文が正しく挿入され、データがコミットされる

- 読み取り専用フラグが true に設定されているため何も行われない

この質問には、前の説明を考えれば簡単に答えられるはずです。正解は A で、読み取り専用接続で更新操作を実行しようとしていることを示す例外がスローされます。トランザクションが開始されることから (REQUIRED)、接続は読み取り専用に設定されます。そのため当然、SQL 文を実行しようとすると、読み取り専用の接続であることを通知する例外を受け取るわけです。

読み取り専用フラグの奇妙なところは、トランザクションを開始しなければ、このフラグが使用されないことです。データを読み取るだけの場合に、なぜトランザクションが必要になるのでしょうか。実を言うと、トランザクションは必要ありません。読み取り専用操作を実行するためにトランザクションを開始すると、処理中のスレッドにオーバーヘッドが追加されるため、データベースで共有読み取り操作がロックされる可能性があります (これは、使用しているデータベースのタイプと、分離レベルの設定によります)。要するに、読み取り専用フラグは JDBC ベースの Java パーシスタンスで使用する場合には意味がなく、不要なトランザクションが開始されることによってオーバーヘッド追加の原因になるということです。

では、ORM ベースのフレームワークを使用する場合はどうでしょうか。引き続きクイズ形式で話を進めると、Hibernate で JPA を使って insertTrade() メソッドを呼び出した場合、リスト 7 の @Transactional アノテーションの結果はどうなるでしょうか。

リスト 7. **REQUIRED** 伝播モードで読み取り専用を使用する場合 — JPA

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

リスト 7 の insertTrade() メソッドの実行結果は、以下のうちのどれですか。

- 読み取り専用接続例外がスローされる
- 取引注文が正しく挿入され、データがコミットされる
- 読み取り専用フラグが true に設定されているため何も行わない

正解は B です。エラーが発生することなく、取引注文は正しくデータベースに挿入されます。前の例では、REQUIRED 伝播モードを使用すると読み取り専用接続例外がスローされると説明しましたが、これは JDBC を使用した場合の結果です。ORM ベースのフレームワークを使用する場合、読み取り専用フラグはデータベースに対してはヒントでしかないのですが、ORM ベースのフレームワーク (この例では Hibernate) に対してはオブジェクト・キャッシュのフラッシュ・モードを NEVER に設定する指示になります。フラッシュ・モードが NEVER に設定されると、オブジェクト・キャッシュは現行の作業単位が処理されている間は、データベースとの同期を取りにいきません。ただし、伝播モードが REQUIRED に設定されていると、このすべてが無効になるので、トランザクションが開始され、読み取り専用フラグが設定されていない場合と同じように動作するというわけです。

このことから、話は私がよく遭遇する別の大きな落とし穴につながります。今までの説明をすべて踏まえた上で、@Transactional アノテーションに読み取り専用フラグだけを設定した場合にリスト 8 のコードがどのように作用するかを考えてください。

リスト 8. 読み取り専用を使用する場合 — JPA

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

リスト 8 の getTrade() メソッドの実行結果は、以下のうちのどれですか。

- トランザクションが開始され、取引注文が取得された後、トランザクションがコミットされる
- トランザクションは開始されず、取引注文が取得される

決して駄目とは言いませんが

整合性のために読み取り操作を分離する場合や、読み取り操作のために特定のトランザクション分離レベルを設定する場合など、データベースの読み取り操作でトランザクションを開始したくなることがあるかもしれません。しかしこのような、ビジネス・アプリケーションでは稀な状況に直面しない限りは、データベースの読み取り操作でトランザクションを開始するのは避けるべきです。というのも、データベースの読み取り操作にはトランザクションは不要であり、この不要なトランザクションの影響で、データベースのデッドロックや、パフォーマンスの低下、スループットの低下などが発生する可能性があるからです。

この場合の正解は A です。トランザクションが開始されてコミットされます。@Transactional アノテーションのデフォルト伝播モードは REQUIRED であることを思い出してください。つまり、トランザクションが実際に必要ない場合でも、トランザクションは開始されます。その結果、使用しているデータベースによっては不要な共有ロックが行われ、データベース内でデッドロック状態が発生する可能性があります。さらに、トランザクションの開始時と停止時に不必要な処理のための時間とリソースが消費されることにもなります。結論として、ORM ベースのフレームワークを使用する場合には、読み取り専用フラグはまったく意味がなく、大抵は無視されるということです。それでも読み取り専用フラグを使用すると主張するのであれば、伝播モードを SUPPORTS に設定して (リスト 9 を参照)、トランザクションが開始されないようにしてください。

リスト 9. 選択操作に SUPPORTS 伝播モードで読み取り専用を使用する場合

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

それでもやはり、読み取り操作を行うときには @Transactional アノテーションを使用しないようにするのが得策です (リスト 10 を参照)。

リスト 10. @Transactional アノテーションを削除した選択操作

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

REQUIRES_NEW トランザクション属性の落とし穴

Spring Framework と EJB のどちらを使用しているかに関わらず、REQUIRES_NEW トランザクション属性の使用が思わぬ結果を招き、データが破損したり、データの整合性が失われたりする場合が

あります。REQUIRES_NEW トランザクション属性は、メソッドの開始時に、既存のトランザクションの有無に関わらず常に新しいトランザクションを開始します。トランザクションを確実に開始するには REQUIRES_NEW 属性を使用するのが正しい方法であると思い込んで、この属性を誤って使用している開発者は少なくありません。その一例として、リスト 11 の 2 つのメソッドを見てください。

リスト 11. REQUIRES_NEW トランザクション属性の使用

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

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

リスト 11 の 2 つのメソッドはどちらも public です。つまり、これらのメソッドはそれぞれ単独で呼び出せるということです。REQUIRES_NEW 属性での問題は、サービス間での通信やオーケストレーションを通じて、この属性を使用する複数のメソッドが同じ論理作業単位の中で呼び出されたときに発生します。例えばリスト 11 の場合で考えると、一部の使用場面では updateAcct() メソッドを他のすべてのメソッドから独立して呼び出すことが可能ですが、updateAcct() メソッドが insertTrade() メソッド内で呼び出されるというケースもあります。この場合、updateAcct() メソッド呼び出しの後に例外が発生すると、取引注文はロールバックされる一方、口座の更新はデータベースにコミットされることになります (リスト 12 を参照)。

リスト 12. REQUIRES_NEW トランザクション属性を使用した複数の更新操作

```
@Transactional(propagation=Propagation.REQUIRES_NEW)
public long insertTrade(TradeData trade) throws Exception {
  em.persist(trade);
  updateAcct(trade);
  //exception occurs here! Trade rolled back but account update is not!
  ...
}
```

このような事態が発生する理由は、新しいトランザクションが updateAcct() メソッド内で開始され、メソッドが終了した時点でそのトランザクションがコミットされるからです。REQUIRES_NEW トランザクション属性の使用時には、既にトランザクション・コンテキストが存在する場合、現行のトランザクションが中断されて新しいトランザクションが開始されます。メソッドが終了すると、この新しいトランザクションがコミットされて元のトランザクションが再開します。

このような振る舞いになることから、REQUIRES_NEW トランザクション属性を使用するのは、重複して行われているトランザクションの結果にかかわらず、呼び出し中のメソッド内のデータベース・アクションをデータベースに保存する必要がある場合に限らなければなりません。例えば、試行されたすべての株式取引を監査用のデータベースに記録する必要があるとします。この情報は、検証エラー、資金不足、あるいはその他の理由で取引が失敗したとしても保存されなければなりません。監査メソッドで REQUIRES_NEW 属性を使用しなければ、監査レコードは試行された取引と一緒にロールバックされてしまいます。REQUIRES_NEW 属性を使用することで、初期トランザクションの結果に関わらず、監査データが確実に保存されます。ここでの重要なポイントは、この監査の場合のような理由で REQUIRES_NEW 属性を使用するのでない限り、REQUIRES_NEW 属性ではなく必ず MANDATORY 属性または REQUIRED 属性を使用するということです。

トランザクション・ロールバックの落とし穴

最も多く見られるトランザクションの落とし穴は、最後の説明として残しておきました。残念ながら、大半の実動コードがこの落とし穴に陥っています。まずは Spring Framework での場合を説明してから、EJB 3 での説明に移ります。

これまでは、リスト 13 のような見かけのコードを検討してきました。

リスト 13. ロールバック・サポートがないコード

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

例えば、口座の資金が当該株式を購入するには足りなかったり、株式売買用の口座がまだ開設されていなかったりしたために、チェック例外 (FundsNotAvailableException など) がスローされたとします。この場合、取引注文はデータベースに保存されるでしょうか。それとも論理作業単位全体がロールバックされるでしょうか。その答えは意外なことに、(Spring Framework または EJB のいずれの場合でも) チェック例外がスローされた時点で、トランザクションはまだコミットされていないすべての作業をコミットします。リスト 13 の場合で言うと、updateAcct() メソッドの実行中にチェック例外が発生すると、取引注文は保存される一方、口座は取引を反映するように更新されません。

これはおそらく、トランザクションを使用する際のデータの完全性と整合性における最大の問題です。ランタイム例外 (つまり、非チェック例外) は自動的に論理作業単位全体をロールバックさせますが、チェック例外による強制ロールバックは行われません。そのためトランザクションの観点からすると、リスト 13 のコードは使いものになりません。このコードはトランザクションを使用してアトミック性と整合性を維持しているように見えても、実際にはそうではないからです。

このような振る舞いは奇妙に思えるかもしれませんが、トランザクションのこの振る舞いには正当な理由があります。第一に、すべてのチェック例外が悪いわけではありません。チェック例外はイベント通知に使用されることも、特定の条件に応じて処理をリダイレクトするために使用されることもあるからです。しかしそれよりも肝心なことは、特定のチェック例外では、アプリケーション・コードが是正アクションを行い、それによってトランザクションを完了することも可能であるという点です。例えば、オンライン書籍小売業者のためにコードを作成しているというシナリオを考えてみてください。本の注文を完了するには、注文プロセスの一環として注文確認の E メールを送信しなければなりません。E メール・サーバーがダウンしている場合には、メッセージの送信が不可能であることを示すある種の SMTP チェック例外を送信することになります。その際、チェック例外によって自動ロールバックが行われるとなると、E メール・サーバーがダウンしていたという理由だけで本の注文全体がロールバックされてしまいます。チェック例

外で自動ロールバックが行わなれなければ、例外をキャッチして何らかの是正アクション (メッセージを保留キューに送信するなど) を行い、残りの注文をコミットすることができます。

宣言型トランザクション・モデルを使用するときには (この連載の第 2 回で詳しく説明します)、コンテナーまたはフレームワークがチェック例外をどのように処理するかを指定する必要があります。Spring Framework では、チェック例外の処理方法を指定するには `@Transactional` annotation アノテーション内で `rollbackFor` パラメーターを使用します (リスト 14 を参照)。

リスト 14. トランザクション・ロールバック・サポートの追加 — Spring

```
@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 the error
    throw up;
  }
}
```

`@Transactional` アノテーション内での `rollbackFor` パラメーターの使用方法に注意してください。このパラメーターに指定できるのは、例外クラスまたは例外クラスの配列です。あるいは、`rollbackForClassName` パラメーターを使って例外の名前を Java `String` 型として指定することもできます。また、このプロパティーの否定形バージョン (`noRollbackFor`) を使用することで、特定の例外を除くすべての例外を強制的にロールバックすることができます。ほとんどの開発者は通常、`rollbackFor` の値として `Exception.class` を指定してメソッド内のすべての例外を強制ロールバックするという方法を採っています。

一方、EJB のトランザクション・ロールバックに関する仕組みは、Spring Framework とは多少異なります。EJB 3.0 仕様にある `@TransactionAttribute` アノテーションは、ロールバックの振る舞いを指定するディレクティブを組み込みません。そのため、このアノテーションの代わりに `SessionContext.setRollbackOnly()` メソッドを使用して、ロールバックするトランザクションにマークを付ける必要があります。リスト 15 に、その方法を示します。

リスト 15. トランザクション・ロールバック・サポートの追加 — EJB

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

いったん `setRollbackOnly()` メソッドが呼び出されたら、もう止めることはできません。唯一考えられる結果は、トランザクションを開始したメソッドが完了した時点で、トランザクションがロール

バックされることです。トランザクション・ストラテジーについて説明するこのシリーズの今後の記事では、いつどこでロールバック・ディレクティブを使用すればよいのか、また REQUIRED トランザクション属性と MANDATORY トランザクション属性はどのように使い分けるのかを説明します。

まとめ

Java プラットフォームにトランザクションを実装するためのコードはそれほど複雑というわけではありませんが、実装コードを使用する方法、構成する方法に関しては多少複雑になり得ます。Java プラットフォームでのトランザクション・サポートの実装には、多くの落とし穴が伴うからです (これには、この記事で説明していない、それほど一般的ではない落とし穴も含まれます)。ほとんどのトランザクション実装で最大の問題となるのは、コンパイラーの警告やランタイム・エラーとしては、トランザクション実装が不適切であることが知らされることはないということです。その上、この記事の最初に紹介した「たとえ遅くても、何もしないよりはましです」の逸話の内容に反し、トランザクション・サポートの実装は簡単ではなく、コーディングを行うだけでは済みません。全体的なトランザクション・ストラテジーの開発に、かなりの設計作業が費やされます。連載「トランザクション・ストラテジー」では今後、単純なアプリケーションからハイパフォーマンス・トランザクション処理に至るまでの様々な使用状況に有効なトランザクション・ストラテジーの設計方法について解説していきます。

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。