Javaの理論と実践: JTSを理解する -- トランザクションについて

トランザクションは信頼性の高いアプリケーションのビルディング・ブロック

Brian Goetz
Principal Consultant
Quiotix

2002年 3月 01日

Java Transaction Serviceは、J2EEアーキテクチャーの重要な要素です。Java Transaction APIと併用することによって、あらゆる種類のシステム障害やネットワーク障害に対して堅固な分散アプリケーションの構築が可能となります。トランザクションは、信頼性の高いアプリケーションの基本となるビルディング・ブロック（構成要素）であり、トランザクションのサポートがなければ、信頼性の高い分散アプリケーションの作成はきわめて困難なものとなるでしょう。幸いJTSでは、そうした作業のほとんどをプログラマーに意識させずに行うことができ、J2EEコンテナーによって、トランザクション境界の設定とリソースの使用を表に出ないやり方で行います。3回シリーズの第1回である今回の記事では、トランザクションとは何か、また、信頼性の高い分散アプリケーションを構築する上でなぜそれが重要になるのか、という点について基本となるポイントを説明します。

このシリーズの他の記事を見る

J2EEに関する入門用の記事や書籍を見ると、Java Transaction Service (JTS) やJava Transaction API (JTA) に関する記事はほんのわずかしかないことが分かります。これは、JTSの重要性が低いとか、それがJ2EEのオプションに過ぎない、というような理由によるものではありません。実際は、全く逆です。JTSによってアプリケーションに提供されるサービスは、ほとんど意識せずに使用できるため、EJBテクノロジーのように注目されることがないというのがその理由です。開発者の多くは、アプリケーションの中でトランザクションがどこで開始され、どこで終了するのか、気付きもしていません。JTSの存在がそれほど強く認識されていないということは、ある意味では、JTSの成功の証ともいえます。つまり、トランザクション管理がどのように行われるのか、その内容がきわめて巧妙に隠されてしまうため、それ自体について多く語られることや聞かれることがなくなってしまうということです。しかし、皆さんはおそらく、トランザクションが皆さんのために陰でどのようなことをしているのかを知りたいと思うでしょう。

トランザクションがなければ、高い信頼性を持つ分散アプリケーションの作成はほとんど不可能である、と言っても過言ではありません。トランザクションによって、アプリケーションの永

続的なステート（状態）に対して統制のとれた方法で変更を行うことができ、その結果、アプリケーションを、システム・クラッシュ、ネットワーク障害、電源障害、さらには自然災害を含め、あらゆる種類のシステム障害に対して堅固なものとすることができます。トランザクションは、耐障害性、高信頼性、高可用性を備えたアプリケーションの構築に必要な、基本となるビルディング・ブロックの1つです。

なぜトランザクションが必要なのか

銀行口座の振替を例として考えてみましょう。まず、それぞれの口座の残高が、データベース・テーブルの行によって表されます。口座Aから口座Bにお金の振替をする場合、おそらく次のようなSQLコードを実行することになるでしょう。

```
SELECT accountBalance INTO aBalance
    FROM Accounts WHERE accountId=aId;
IF (aBalance >= transferAmount) THEN
    UPDATE Accounts
        SET accountBalance = accountBalance - transferAmount
        WHERE accountId = aId;
    UPDATE Accounts
        SET accountBalance = accountBalance + transferAmount
        WHERE accountId = bId;
    INSERT INTO AccountJournal (accountId, amount)
        VALUES (aId, -transferAmount);
    INSERT INTO AccountJournal (accountId, amount)
        VALUES (bId, transferAmount);
ELSE
    FAIL "Insufficient funds in account";
END IF
```

ここまでは、かなり分かりやすいコードです。Aの口座に十分な残高があれば、その金額がAの口座から引かれ、Bの口座に移されます。しかし、電源障害やシステム・クラッシュが発生した場合にはどうなるでしょうか。口座Aと口座Bの行が同じディスク・ブロックに格納されているということはおそらくないでしょう。つまり、この振替には複数のディスクI/Oが必要になるということです。最初のI/Oの書き込みと、次のI/Oの書き込みの間にシステム障害が発生した場合、一体どうなるでしょうか。そのような場合、その金額がAの口座から差し引かれているにもかかわらずBの口座に加算されないということも起こり得ます (A、Bのいずれにとっても望ましくない)、あるいは、金額がBの口座に加算されているにもかかわらず、Aの口座から差し引かれていないということもあり得ます (銀行にとって望ましくない)。あるいは、口座の更新が適切に行われても、アカウント・ジャーナルの更新が行われなかった場合にはどうなるでしょうか。そのような場合は、AとBの銀行取引明細書に記される取引額が、それぞれの口座残高と一致しなくなってしまいます。

複数のデータ・ブロックを同時にディスクに書き込むことは不可能であり、また、一部の変更に際してディスクへのすべてのデータ・ブロックの書き換えを行うこともシステム・パフォーマンスに悪影響を及ぼすでしょう。ディスクへの書き込みをより適切な時点まで延期することで、アプリケーションのスループットを大幅に向上させることは可能ですが、その場合、データの整合性を危うくすることのない方法でそのようにする必要があります。

システム障害が発生しないとしても、上記のコードでは、検討すべきもうひとつのリスクがあります。それは並行性の問題です。Aの口座に100ドルあり、Aの口座から別の2つの口座へ100ドルずつの振替を全く同時に行った場合、どのようなことが起きるでしょうか。タイミングが悪く、

また、適切なロック・メカニズムがない場合、両方の振替が成立してしまい、Aの口座の残高はマイナスとなります。

このような例は実際に起こりうるものであり、このような状況に対応できるエンタープライズ・データ・システムを視野に入れることは必要でしょう。銀行はたとえ火災、洪水、停電、ディスク障害、システム障害などに遭っても正しいアカウント・レコードを維持してくれるものと、人々は思っています。フォールト・トレランスは、ディスク、コンピューター、さらにはデータ・センターの二重化といった冗長性によって確保することができますが、耐障害性ソフトウェア・アプリケーションの構築を現実のものとしているのは、ほかならぬトランザクションなのです。トランザクションは、システムやコンポーネントの障害に際しての、データの一貫性と整合性を強化するためのフレームワークを提供します。

トランザクションとは

では、トランザクションとは一体どのようなものなのでしょうか。この用語を定義する前に、まずアプリケーションのステートという概念を定義しておきましょう。アプリケーションのステートは、アプリケーションのオペレーションに影響を及ぼすメモリー内およびディスク上のあらゆるデータ項目、つまりアプリケーションが「認識している」すべてを包含します。アプリケーションのステートは、メモリー、ファイル、あるいはデータベースに格納されます。たとえば、アプリケーション、ネットワーク、またはコンピューター・システムの破壊といったシステム障害が発生した場合には、システムの再始動時にアプリケーションのステートが確実に復元できる必要があります。

ではここで、トランザクションを、アプリケーションのステートにおけるオペレーションの関連づけられた集合とし、原子性 (atomicity)、一貫性 (consistency)、独立性 (isolation)、持続性 (durability)という特性を備えるものと定義することにします。これらの特性は、総称してACID特性と呼ばれます。

原子性とは、トランザクションのオペレーションのすべてがアプリケーションのステートに適用されるか、あるいはオペレーションが一切適用されないかのどちらかであるということを意味します。つまり、トランザクションがこれ以上分割できない作業単位であるということです。

一貫性とは、トランザクションがアプリケーションのステートの正しいトランスフォーメーションを表すということを意味します。つまり、アプリケーション面での整合性制約がトランザクションによって侵害されないということです。実際問題として、一貫性の概念は、アプリケーション毎に固有のものです。たとえば、会計アプリケーションでは、一貫性には、すべての資産勘定の合計がすべての負債勘定の合計と等しいというインバリアントが含まれます。この要件については、このシリーズの第3回でトランザクション境界の説明をする際にもう一度扱うことにします。

独立性とは、1つのトランザクションの処理結果が、並行して実行される他のトランザクションに影響を与えないということを意味します。トランザクションの観点では、トランザクションが並列ではなく順次に実行されているように見えます。データベース・システムでは、独立性は通常、ロック・メカニズムによって実装されます。時として、特定のトランザクションに対する独立性の要件を緩和することにより、より優れたアプリケーション・パフォーマンスが得られる場合もあります。

持続性とは、一旦トランザクションが正常に完了すると、アプリケーションのステートへの変更が障害に耐えて存続するということを意味します。

「障害に耐えて存続する」とは、どのようなことでしょうか。耐えることのできる障害とは、どのようなものなのでしょうか。これはシステムによって異なり、優れた設計のシステムでは、リカバリーが可能な障害が明示的に示されています。たとえば、私がデスクトップ・ワークステーションで使用しているトランザクション・データベースは、システム・クラッシュや電源障害に対しては堅固ですが、オフィス・ビルが火事で焼失してしまった場合はその限りではありません。また銀行では、データ・センターにおけるディスク、ネットワーク、システムの二重化だけでなく、二重化された通信リンクによって接続されたデータ・センターを別のサイトに持ち、自然災害のような重大な障害に際してのリカバリーを行えるようにしている場合もあります。また、軍事用のデータ・システムにはより厳しいフォールト・トレランスの要件が求められる場合もあります。

トランザクションの構造分析

典型的なトランザクションには、アプリケーション、トランザクション処理モニター (TPM)、1つまたは複数のリソース・マネージャー (RM) がかかわります。RMは、アプリケーションのステートを格納し、多くの場合はデータベースを指しますが、メッセージ・キュー・サーバー (J2EEアプリケーションではJMSプロバイダー) や他のトランザクション・リソースを指すこともあります。TPMがRMのアクティビティーを調整し、トランザクションの「オール・オア・ナッシング」という特性を確保します。

新しいトランザクションを開始するようにアプリケーションからコンテナーまたはトランザクション・モニターに指示が出されると、トランザクションが開始されます。アプリケーションが各種のRMにアクセスすると、RMがトランザクション内で使用されます。RMは、アプリケーションのステートに対する変更を、その変更を要求しているトランザクションに関連づけます。

アプリケーションによってトランザクションがコミットされた場合、または、アプリケーションによって、あるいは、RMの1つが失敗したために、トランザクションがロールバックされた場合、のいずれかによってトランザクションは終了します。トランザクションが正常にコミットした場合、そのトランザクションに関連づけられた変更は永続ストレージに書き込まれ、新しいトランザクションによって認識されるようになります。これがロールバックされた場合には、そのトランザクションによって行われたすべての変更は廃棄され、トランザクションが全くなかったかのような状態になります。

トランザクション・ログ - 持続性の鍵

トランザクションRMは、複数のトランザクションの処理結果をひとつのトランザクション・ログに要約することによって、受容できる程度のパフォーマンスを維持しながら持続性を達成します。トランザクション・ログは、順次ディスク・ファイルとして (時としてパーティションに直接) 格納され、ロールバックやリカバリーの場合を除いては、通常、書き込み専用であり、読み取りは行われません。先の銀行口座の例を用いれば、口座Aと口座Bの残高はメモリー内で更新され、新しい残高とその前の残高がトランザクション・ログに書き込まれます。更新レコードをトランザクション・ログに書き込む場合には、(書き込みが必要なのは、ディスク・ブロック全体ではなく、変更されたデータのみであるため) ディスクに書き込まれるデータ総量も少なくてすみ、また、(すべての変更がログ内の連続したディスク・ブロックに含まれるため) ディスク・シーク

も少なくてすみます。さらに、複数の並行トランザクションに関連する変更は、トランザクション・ログに対する1つの書き込みにまとめることができます。つまり、1つのトランザクションに対して複数回のディスク書き込みを必要とするのではなく、複数のトランザクションを1回のディスク書き込みで処理できるということです。その後、変更されたデータに対応する実際のディスク・ブロックの更新がRMによって行われます。

再始動時のリカバリー

システムに障害が発生した場合、再始動時に最初に行われることは、ログにあるコミット済みトランザクションのうち、まだデータ・ブロックが更新されていないものの処理結果を再度反映することです。このようにして、ログによって、さまざまな障害に対する持続性が保証され、また、行うべきディスクI/Oオペレーションの数を軽減したり、あるいは、少なくともシステム・パフォーマンスへの影響がより小さくてすむようなタイミングにまで遅らせることができるようになります。

2フェーズ・コミット

多くのトランザクションにおいては、単一のRM (通常、データベース) しか関与しません。そのような場合、RMは通常、トランザクションをコミットするか、またはロールバックする作業のほとんどを行います。(ほとんどすべてのトランザクションRMには独自のトランザクション・マネージャーが組み込まれており、これがローカル・トランザクション、つまりそのRMにのみ関与するトランザクションを処理することができます。)しかし、2つの別々のデータベース、データベースとJMSキュー、あるいは2つの別々のJMSプロバイダーなど、トランザクションに複数のRMが関与する場合には、「オール・オア・ナッシング」のセマンティクスがRM内だけでなく、トランザクション内のすべてのRMに適用されるようにすべきです。この場合、TPMが2フェーズ・コミット処理を指揮します。2フェーズ・コミットでは、まずTPMが「Prepare」メッセージを各RMに送信して、トランザクションをコミットする準備ができているかどうかを確認します。すべてのRMからオーケーの応答を受け取ると、トランザクション・ログでそのトランザクションにコミット済みとしてマークを付け、すべてのRMにそのトランザクションをコミットするよう指示を出します。失敗したRMがあった場合、再始動の際に、障害発生時に保留になっていたトランザクションの状態についてTPMに確認し、これをコミット、あるいは、ロールバックします。

2フェーズ・コミットは、社会的な例で考えると結婚式に例えることができます。まず、牧師あるいは判事がふたりにそれぞれ「あなたはこの男性/女性を夫/妻としますか」と尋ねます。両者共に「はい」と答えれば、結婚が成立したことが宣言されます。それ以外の場合は、結婚は成立しません。どちらが先に「はい」と答えるかにかかわりなく、一方の結婚が成立していながら、もう一方の結婚が成立していないというようなことは決してありません。

トランザクションは例外処理のメカニズム

皆さんは、同期化（synchronized）ブロックがメモリー内データに対して行っているのと同じ機能の多くを、トランザクションがアプリケーション・データに対して提供しているということにお気付きかもしれません。それは「原子性の保証」「変更の可視性」「明確な順序付け」です。しかし、同期化が並行性制御のメカニズムであるのに対して、トランザクションは主として例外処理のメカニズムです。ディスク障害の発生も、システムとソフトウェアのクラッシュもなく、電源も100%の信頼性が保証されているような環境であれば、トランザクションというものは必要ありません。トランザクションは、実社会における契約法のような役割をエンタープライズ・アプリケーションにおいて果たしています。つまり、当事者の一方が契約を果たすことができなかっ

た場合に、どのようなプロセスで責務を解除するのかを規定しているのです。通常、私たちが契約書にサインする際には、その契約書を引き合いに出すような事態が起きないことを私たちは願うものであり、また幸い、実際には、そのような事態はそう頻繁には起こりません。

よりシンプルなJavaプログラムに例えれば、`catch`や`finally`ブロックがメソッド・レベルで行うのと同じ便宜の一部を、トランザクションは、アプリケーション・レベルで提供します。つまり、たくさんのエラー・リカバリー・コードを記述せずに、信頼できるエラー・リカバリーが実行できるようになります。ファイル・コピーのための次のメソッドについて見てみましょう。

```
public boolean copyFile(String inFile, String outFile) {
  InputStream is = null;
  OutputStream os = null;
  byte[] buffer;
  boolean success = true;

  try {
    is = new FileInputStream(inFile);
    os = new FileOutputStream(outFile);
    buffer = new byte[is.available()];
    is.read(buffer);
    os.write(buffer);
  }
  catch {IOException e) {
    success = false;
  }
  catch (OutOfMemoryError e) {
    success = false;
  }
  finally {
    if (is != null)
      is.close();
    if (os != null)
      os.close();
  }

  return success;
}
```

ファイル全体に単一のバッファーを割り振っているという問題は無視することとして、このメソッドのどのような点に問題があるのでしょうか。問題はたくさんあります。まず、入力ファイルが存在しないかもしれません。あるいは、ユーザーに読み取りの権限がないかもしれません。また、ユーザーに出力ファイルへの書き込みの権限がない可能性や出力ファイルが別のユーザーによってロックされている可能性もあります。さらにまた、ファイルの書き込み操作を完了するのに十分なディスク・スペースがない可能性、使用可能なメモリーが十分でなくバッファーの割り振りに失敗する可能性もあります。幸い、`copyFile()` によって使用されるリソースをすべて解放する`finally` 文節によって、こうしたことはすべて処理されます。

このメソッドをあの古き悪しきCの時代に記述していたとすれば、オペレーションごと (open input、open output、malloc、read、write) にいちいちリターン状況をテストしなければならず、オペレーションが失敗すれば、前に正常に行われたオペレーションのすべてを取り消して、適切な状況コードを戻さなければならなくなります。これらのエラー処理コードによってコードの量が増え、読み取りにますます手間がかかるようになります。また、リソースの解放に失敗したり、リソースの解放を2回行ってしまったり、まだ割り振られていないリソースを解放してしまったりなどして、エラー処理コード (これは図らずもテストが最も難しい部分なのですが) の中でエ

ラーを発生させてしまうことも簡単に起こり得ます。そして、2つのファイルと1つのバッファー程度では収まらない、多くのリソースがかかわってくるような、より重要なメソッドでは、一層複雑さが増します。これらのエラー・リカバリー・コードによって、実際のプログラム・ロジックが分かりづらいものになってしまう場合もあります。

ではここで、複数のデータベースで複数の行を挿入または更新する複雑なオペレーションを実行中であり、オペレーションの1つが整合性制約に違反して失敗した場合を想定してみましょう。エラー・リカバリーの管理には、どのオペレーションをすでに実行済みであるか、また、後続のオペレーションが失敗した場合にそれぞれをどのように取り消すかを追跡することが必要です。作業単位が複数のメソッドやコンポーネントにわたって分散している場合には、より一層事態は複雑になります。アプリケーションをトランザクションで構造化することによって、この記帳作業をすべてデータベースに任せることができます。また、ROLLBACKという指示だけで、トランザクションの開始以来行ったことをすべて取り消すことができます。

結論

アプリケーションをトランザクションで構造化することによって、アプリケーションのステートの一連の正しい変換を定義し、たとえシステムやコンポーネントに障害が発生した後でも、アプリケーションを常に適正なステートに保つことができます。トランザクションによって、私たちは、例外処理とリカバリーの多くの要素をTPMやRMに任せることができるため、コードを簡略化し、アプリケーションのロジックに集中できるようになります。

このシリーズの第2回では、このことがJ2EEアプリケーションにどのような意味があるのかについてさらに検討します。J2EEがいかにしてトランザクション・セマンティクスをJ2EEコンポーネント (EJBコンポーネント、サーブレット、JSPページ) に伝えられるようにするか、いかにしてリソースのリスト追加を (bean対応のトランザクションであっても) アプリケーションに全く意識させずに行えるのか、また、単一のトランザクションがEJBコンポーネント間で、あるいはサーブレットからEJBコンポーネントへ、あるいは複数のシステムにおいて、いかにして透過的に制御の流れに従うことができるのか、という点について見ていきます。

たとえJ2EEが比較的透過的にオブジェクト・トランザクション・サービスを提供したとしても、アプリケーション設計者は依然として、どこをトランザクション境界にするか、また、トランザクション・リソースをアプリケーションでどのように使うかという点について慎重に検討する必要があります。誤ったトランザクション境界は、アプリケーションの一貫性を損ねるおそれがあり、また、トランザクション・リソースの不適正な使用は、パフォーマンスの重大な問題を招くおそれがあります。このシリーズの第3回ではこうした問題を取り上げ、トランザクションを構造化する方法に関するいくつかのアドバイスを示します。

著者について

Brian Goetz

Brian Goetz は18 年間以上に渡って、専門的ソフトウェア開発者として働いています。彼はカリフォルニア州ロスアルトスにあるソフトウェア開発コンサルティング会社、Quiotixの主席コンサルタントであり、またいくつかのJCP Expert Groupの一員でもあります。2005年の末にはAddison-Wesleyから、Brianによる著、Java Concurrency In Practiceが出版される予定です。Brian著による有力業界紙に掲載済みおよび掲載予定の記事のリストを参照してください。