Spring Web Flow 2 でのフロー・マネージド・パーシスタンス

トランザクション Web フローのためのパーシスタンス・ストラテジー

Xinyu Liu
Senior App Dev Consultant
Magellan Health Services

2010年 4月 13日

Spring Web Flow 2 の JPA/Hibernate パーシスタンス・アーキテクチャーは、フロー・マネージド・パーシスタンスの概念に基づいていますが、この概念については、これまで簡単に説明されたドキュメントしかありませんでした。フロー・マネージド・パーシスタンスに関して掘り下げるこの記事では、Xinyu Liu がフロー・マネージド・パーシスタンスの概念上の構成要素、そしてフロー・スコープド・パーシスタンス・コンテキストについて説明し、実際の複雑な開発シナリオでアトミック Web フローを処理する場合、および非アトミック Web フローを処理する場合のトランザクション・ストラテジーについて具体的に解説します。

Spring Web Flow は、Spring MVC 技術を拡充する革新的な Java™ Web フレームワークです。Spring Web Flow でのアプリケーション開発は、Web フローとして定義された使用ケースを中心に構成されています。開発ワークスペースを Web フローの観点から構成することで、より意味のある、コンテキストに沿った開発エクスペリエンスが実現します。さらに、Spring Web Flow の JPA/Hibernate パーシスタンスのサポートは、このフレームワークが最も大きくサーバー・サイドに貢献しているものの 1 つに挙げられます。

Spring Web Flow については SpringSource と Spring Web Flow プロジェクト・チームによって詳細なドキュメントが作成されていますが、このフレームワークのパーシスタンス・サポート、特にフロー・マネージド・パーシスタンス (flow-managed persistence) のメカニズムについてはほとんど説明されていません。この記事では、フロー・マネージド・パーシスタンスとその必須構成要素であるフロー・スコープド・パーシスタンス・コンテキスト (flow-scoped persistence context) に重点を置き、Spring Web Flow 2 での Java パーシスタンス・プログラミングについて掘り下げて説明します。

Spring Web Flow でのパーシスタンスの概念を概説するのに続き、読み取り専用トランザクションと読み取り/書き込みトランザクションを、アトミック Web フローおよび非アトミック Web フローのそれぞれで処理する場合のストラテジーについて、使用ケースを用いて説明します。いずれの場合も、優先して使われるトランザクション処理ストラテジーの概念的な基礎を説明すると

ともに、その欠点を明らかにします。そして最後に、Spring Web Flow 2 でトランザクションを効率的かつ安全に管理するために私が独自に定めている指針を紹介して記事を締めくくります。

この記事は、Spring Web Flow 2 とその継続 (continuation) ベースのアーキテクチャーを十分に理解している経験豊かな Java 開発者を対象としています。この記事で説明する使用ケースとサンプル・アプリケーション・コードは、Spring Web Flow アプリケーションで既に JPA/Hibernate を使用している開発者には特に役立つはずです。

JPA/Hibernate におけるパーシスタンスの問題

典型的な Web アプリケーションにおけるユーザー・リクエストの処理は、主にアクションの処理、ビューのレンダリングという 2 段階で行われます。アプリケーションの主要なビジネス・ロジックは、この 2 段階の処理のうちアクションの処理の段階に属しています。そして、その後に行われるビューのレンダリングで、データをビュー・テンプレートに流し込んで表示を生成します。

JPA/Hibernate では、データ (具体的にはエンティティー間の関係) が即座にロードされることもあれば、プロキシー・オブジェクトとして遅延ロードされることもあります。ビューのレンダリング・フェーズでパーシスタンス・コンテキストのオブジェクト (JPA の `EntityManager` または Hibernate の `Session`) がすでにクローズされている場合、エンティティーは分離された状態になります。このような分離された状態のエンティティー間のアンロードされた関係へのアクセスが試行されると、`LazyInitializationException` が発生する結果となります。

Open Session in View (「参考文献」を参照) は、この `LazyInitializationException` が発生する問題を解決しようとするパターンです。Open Session in View がフィルターまたはインターセプターとして実装されていれば、ビューのレンダリング中、パーシスタンス・コンテキストのオブジェクトはオープン状態を維持します。永続エンティティーのアンロードされた関係にナビゲートすると、別のデータベース・クエリーがトリガーされて、オンデマンドでその関係がフェッチされることになります。

Open Session in View パターンのマイナス面は、パーシスタンス・コンテキストのオブジェクトのスコープが実質的にユーザー・リクエストに設定されることです。そのため、現行リクエスト以外のサーブレットのスコープで保管されるエンティティーは、必ず分離されてしまいます。分離されたエンティティーを現行のパーシスタンス・コンテキストに関連付けるには、マージ/再アタッチ/再ロード操作を行わなければなりません。

Spring Web Flow が採る手法はこれとは異なり、エンティティーが分離されてしまうという問題をフロー・マネージド・パーシスタンスによって解決します。具体的には、フロー・スコープド・パーシスタンス・コンテキストのオブジェクトを使用するという手法です。

フロー・マネージド・パーシスタンス

Spring Web Flow でのアプリケーション開発は、Web フローの概念に基づいて行われます。Web フローは通常、1 つの使用ケースを表します。多くの場合、Web フロー全体でのデータの変更は、アトミックでなければなりません。つまり、フローのさまざまな段階で行われた変更は、まとめて 1 つの単位としてバックエンド・データベースに保存するか、あるいはデータベースに変更の痕跡を一切残さずに、完全に取り消さなければならないということです。

Spring Web Flow がトランザクション・アトミック Web フローでの JPA/Hibernate プログラミングを容易にするために使用している手段は、フロー・マネージド・パーシスタンスのメカニズムです。フロー・マネージド・パーシスタンスは、概念的には Hibernate/Seam 対話 (「参考文献」を参照) と同じで、Web フロー (Seamで言う「ページ・フロー」) の中で行われたデータの変更は、同じフロー・スコープド・パーシスタンス・コンテキストのオブジェクトにダーティー・エンティティーとしてキャッシュされます。SQL の挿入/更新/削除文は、フローの終了時に変更がまとめて一度にフラッシュされてデータベースにコミットされるまでは、実行されません (「フラッシュ」と「コミット」は別々の概念であることに注意してください。前者は SQL の一連の挿入/更新/削除文を実行して、ダーティー・エンティティーを対応するデータベース値と同期させます。後者は、単にデータベース・トランザクションをコミットするだけです)。

フロー・マネージド・パーシスタンスにおける OptimisticLockingFailureException

オプティミスティック・ロックは、データベースに物理的なロックを設定することなくデータ保全性を保証する、極めて効果的な並行性制御方式です。強制まではしませんが、フロー・マネージド・パーシスタンスではオプティミスティック・ロックを使用することを是非ともお勧めします。

パーシスタンス・コンテキストはフラッシュ時にエンティティーのバージョンをチェックし、エンティティーに対する同時変更を検出した場合には `OptimisticLockingFailureException` (Hibernate での `StaleObjectException` に相当) をスローします。メモリー内でのエンティティーの存続時間が長ければ長いほど、エンティティーに対応するデータベース値が他のプロセスによって変更される可能性が高くなります。

Open Session in View パターンでは、前述のとおりエンティティーの永続状態はユーザー・リクエストに依存します。エンティティーが分離されると、その後のユーザー・リクエストでは一般に、エンティティーの永続状態をリストアするためのマージ/再アタッチ/再ロード操作が必要となります。この操作によって、エンティティーとそれに対応するデータベース値が同期されるからです。

フロー・マネージド・パーシスタンスでは、エンティティーは複数のユーザー・リクエストにわたって、その永続状態を維持します。ユーザー・リクエストが行われてから、次のユーザー・リクエストが行われるまでの間にデータベースの同期は強要されません。そのため、`OptimisticLockingFailureException` が発生する可能性が高くなります。これに対する対策は、他のあらゆるビジネス・チェック例外の場合と同じく、`OptimisticLockingFailureException` をグレースフルに処理することです (`OptimisticLockingFailureException` はデータベース・トランザクションをロールバックするランタイム例外ですが、それでもこの対策が該当します)。一般的なストラテジーとしては、ユーザーに変更をマージするかどうかを選択するためのオプションを表示するか、または失効していないデータでフローを再開する方法が挙げられます。

フロー・スコープド・パーシスタンス・コンテキスト

Web フローは、XML 形式のフロー定義ファイルとして宣言されます。`<persistence-context/>` タグを使用して Web フローが開始されると、新しいパーシスタンス・コンテキストのオブジェクトが作成されて、フロー・スコープにバインドされます。このオブジェクトはユーザー・リクエストを待機している間は下位の JDBC 接続からは切り離され、ユーザー・リクエストに対応する時

点で再び接続されます。つまり、フロー全体を通して同じパーシスタンス・コンテキストのオブジェクトが再利用されることから、エンティティーが分離された状態になることも、それによって `LazyInitializationException` が発生することもありません。

パーシスタンス・コンテキストは、現行のリクエスト・スレッドにもバインドされ、開発者に 2 つの異なる形で公開されます。1 つは暗黙的変数 `persistenceContext` という形、もう 1 つは JPA `@PersistenceContext` アノテーションによって Spring Bean に注入された形です。

暗黙的変数は、以下のようにフロー定義 XML ファイルで直接使用することができます。

```
<evaluate expression="persistenceContext.persist(transientEntityInstance)"/>
```

注入された JPA エンティティー・マネージャーは、DAO やサービス Bean、または Web 層 Bean など、Spring コンポーネント内のどこからでも参照することができます。

パーシスタンス・コンテキストのタイプ: トランザクションまたは拡張

`@PersistenceContext` アノテーションには、`type` というオプション属性があります。この属性は、デフォルトで `PersistenceContextType.TRANSACTION` に設定されます (この設定値は、トランザクションにバインドされたパーシスタンス・コンテキストであることを意味します)。フロー・スコープド・パーシスタンス・コンテキストを使用してプログラミングする際には、このデフォルト設定値を使用しなければなりません。その場合、注入されたトランザクション・バインド・パーシスタンス・コンテキストのオブジェクトは、フロー・スコープの実際のスレッドにバインドされたパーシスタンス・コンテキストに透過的に委任する、単なる共有プロキシーとなります。

属性のもう 1 つの選択肢 `PersistenceContextType.EXTENDED` を選択すると、いわゆる「拡張エンティティー・マネージャー」となります。拡張エンティティー・マネージャーはスレッド・セーフではないため、シングルトン Spring Bean のような同時にアクセスされるコンポーネントでは使用できません。拡張エンティティー・マネージャーをフロー・スコープド・パーシスタンス・コンテキストとして使用すると、アプリケーションでのデータベース/トランザクションが予想外の振る舞いを見せることがあるので、拡張エンティティー・マネージャーは使用しないでください。

面白いことに、Seam 対話は一般に、ステートフル・セッション Bean (EJB) に注入された拡張エンティティー・マネージャーを使って実装されます。この点が、Spring Web Flow のフロー・マネージド・パーシスタンスと Seam 対話との顕著な違いです。

フロー・スコープド・パーシスタンス・コンテキストのオブジェクトとアノテーション `@Transactional` を併せて使用することで、フローのパーシスタンス特性を微調整することができます。

トランザクション・セマンティクス

アノテーションが付けられたクラスまたはメソッドのトランザクション・セマンティクスは、Spring Core パッケージに含まれる `@Transactional` アノテーションによって指定されます。Spring 開発チームによると、`@Transactional` は、インターフェースより、具体的なクラスに

適用したほうが有効です。以下に、デフォルトのトランザクション・セマンティクスを記載します。

```
@Transactional(readOnly=false,propagation=PROPAGATION_REQUIRED,
  isolation=ISOLATION_DEFAULT,timeout=TIMEOUT_DEFAULT)
```

readOnly: `@Transactional(readOnly=false)` を指定して読み取り/書き込みトランザクションを設定すると、パーシスタンス・コンテキストの `FlushMode` が `AUTO` に設定されます。`@Transactional(readOnly=true)` を適用した場合には、そのベースで確立されている Hibernate セッションの `FlushMode` が `MANUAL`に設定されます。

JPA 1.0 では、`MANUAL` フラッシュまたは読み取り専用トランザクションのいずれもサポートしていません。そのため、`@Transactional(readOnly=true)` の設定が意味を持つのは、Hibernate などのベースとなる JPA プロバイダーが読み取り専用データベース・トランザクションをサポートしている場合だけです。さらに Hibernate は、この設定を特定のデータベース・タイプを知る手掛かりとして使用し、クエリーのパフォーマンスを最適化します。

propagation: `propagation` 属性は、現行のメソッドが継承されたトランザクションで実行されているのか、親トランザクションを中断/再開することによって新しく開始されたトランザクションで実行されているのか、あるいはトランザクションでは実行されていないのかを決定します。

isolation: JPA 1.0 ではカスタム分離レベルをサポートしていないため、開発者がデータベース側で、デフォルトのトランザクション分離レベルを指定する必要があります。オプティミスティック・ロックを機能させるために必要な最小レベルは `Read-Committed` です。

timeout: `timeout` 属性は、トランザクションがタイムアウトになるまでの実行時間を指定します (タイムアウトになったトランザクションは、ベースとなるトランザクション・インフラストラクチャーによって自動的にロールバックされます)。

rollbackFor、rollbackForClassname、noRollbackFor、noRollbackForClassname: 原則として、トランザクションはシステム・エラーを示す `RuntimeException` が発生した場合は常にロールバックし、ビジネスの意味が事前に定義されたチェック例外の発生時には常にコミットします。デフォルトのセマンティクスは、この 4 つのロールバック属性を使用してカスタマイズすることが可能です。

Spring Core の堅牢なトランザクション・インフラストラクチャーによって、実際の開発シナリオのほとんどで、容易にトランザクションを管理できるようになります。以降のセクションでは、Spring Web Flow がさまざまな Web フローでのパーシスタンス・プログラミングに対処するために、この Spring トランザクション・インフラストラクチャーとフロー・スコープド・パーシスタンス・コンテキストのオブジェクトをどのように利用するかを説明します。さらに、フロー・マネージド・パーシスタンスの制約を説明する使用ケースもいくつか取り上げます。

アトミック Web フロー

フロー・マネージド・パーシスタンスは、トランザクションの観点からアトミックと見なされる Spring Web Flow の使用ケースに対処することを目的としています。例えば、オンライン・バンキング・システムで、ユーザーが当座預金口座の預金を、普通預金口座や定期預金口座として開設

する口座に移すとします。このようなトランザクションは、複数のステップで行わなければなりません。

1. ユーザーが振り込み元の当座預金口座を選択します。
2. システムが口座残高を表示します。
3. ユーザーが振り込み金額を入力します。
4. ユーザーが振り込み先として普通預金口座または定期預金口座を選択します。
5. システムが確認用にトランザクションの要約内容を表示します。
6. ユーザーがトラザクションをコミットするか、取り消すかを決定します。

このトランザクションには明らかに並行性が必要であることから、まずはエンティティー・クラスでオプティミスティック・ロックを有効に設定することになります。それには、JPA の `@Version` アノテーションか、Hibernate 固有の`OptimisticLockType.ALL` 属性を使用することができます。続いて使用ケース全体を、Spring Web Flow の `<persistence-context/>` タグを使用した単一の Web フローにマッピングします。

Web フローでのトランザクションを使用しないデータ・アクセス

Spring Web Flow では、すべてのデータ・アクセスはデフォルトでトラザクションを使用しないもの (非トランザクション・データ・アクセス) となります。非トランザクション・データ・アクセスの場合、Hibernate はベースとなるデータベースの `auto-commit` モードを `true` に設定します。これによって、各 SQL 文がそれぞれに独自の「短いトランザクション」で即時に実行され、コミットまたはロールバックされます。アプリケーションの観点からすると、データベースでの短いトランザクションは、トランザクションがまったく行われないことと同じです。さらに重要な点として、Hibernate は非トランザクション操作に対してはデフォルトの `FlushMode.AUTO` を無効にします。つまり実質的には、`FlushMode.MANUAL` として機能するということです。

`FlushMode.AUTO` を無効にすることは、フロー・マネージド・パーシスタンスには不可欠です。ビューのレンダリング・フェーズでのエンティティー遅延読み取りも同じく非トランザクション・モードで行われますが、さまざまなビューのレンダリング中にフラッシュが発生すると、フロー終了時に遅延フラッシュを実行する機会がなくなってしまいます。`auto-commit` モードでの非トランザクション読み取りは、実質的に `Read-Committed` 分離レベルのトランザクション内での読み取り操作に相当します。同じように、非トランザクション書き込み操作でも決してフラッシュは行われません。

上記の使用ケースで、各ユーザー・アクションがデータベース・トランザクションの範囲外で実行されるようにする方法は、`@Transactional` アノテーションや XML で構成したトランザクション・アドバイザーを指定しないことです。フロー・スコープド・パーシスタンス・コンテキストのオブジェクトは、フローの期間中にロードされたデータを永続エンティティーとして管理し、データの変更をエンティティーのダーティー状態としてキャッシュします。

フローの最後で、ユーザーが `<end-state commit="true"/>` によって振込トランザクションを確認すると、Spring Web Flow ランタイムはデータベースの読み取り/書き込みトランザクション内で暗黙的に `entityManager.flush()` を呼び出します。続いてトランザクションをコミットし、パーシスタンス・コンテキストのバインドを解除してクローズします。ユーザーが `<end-state commit="false"/>` によってトランザクションを取り消した場合は、フロー・スコープド・パーシ

スタンス・コンテキストがクローズされた時点で、キャッシュされたすべてのデータ変更がメモリーから破棄されます。

このフロー・マネージド・パーシスタンスの手法は、JPA 1.0 が解釈する対話の処理とまったく同じで、これを可能にするのが、ベースとなる Spring Web Flow のコンポーネント、`JpaFlowExecutionListener` クラスです。このような非トランザクション・データ・アクセスによるフロー・マネージド・パーシスタンスの手法とは別に、読み取り専用トランザクションを使用するという方法もあります。

Web フローでの読み取り専用トランザクション

非トランザクション・データ・アクセスよりも、読み取り専用トランザクションを使用したほうがよい場合もあります。その一例として、Spring Web Flow リリースに含まれる「Hotel Booking (ホテル予約)」サンプル・アプリケーション (「参考文献」を参照) を調べてみてください。「booking (予約)」の Web フロー中、操作が読み取り/挿入/更新/削除のどれであろうと、すべてのデータ・アクセスには例外なく `@Transactional(readOnly=true)` が使用されていることに気付くはずです。

読み取り専用トランザクションは JPA 1.0 仕様でサポートされていないため、この設定は特定の JPA プロバイダーでしか使えません。Hibernate の JPA 実装では、ベースとなる Hibernate セッションの `FlushMode` が `MANUAL` に設定され、`auto-commit` モードが `false` に設定されます。

事実上、フロー管理パーシスタンスに対する読み取り専用トランザクションの動作は、アトミック Web フローの終了時にのみ、変更されたエンティティーを `<end-state commit="true"/>` によってフラッシュするという点で、非トランザクション・データ・アクセスと同じです。

`<end-state/>` の前にフラッシュが行われるようにするには、`@Transactional` アノテーションを設定した Spring Bean メソッドのいずれかで `entityManager.flush()` を呼び出す必要があります。

Web フローから直接 `<evaluate expression="persistenceContext.flush()"/>` を呼び出しても、フラッシュは行われません。`<end-state commit="true"/>` 以外の Spring Web Flow タグには、トランザクションがバインドされていないためです。直接呼び出した場合には、以下のエラー・メッセージを受け取ることになります。

```
"javax.persistence.TransactionRequiredException: no transaction is in progress"
```

「Hotel Booking (ホテル予約)」サンプル・アプリケーションについては、あとでフロー・スコープド・パーシスタンス・コンテキストを使用しない場合のパーシスタンス・プログラミングに伴う問題を調べるときに再び取り上げます。

トランザクション伝播についての追加情報

前に、`propagation` 属性の値に基づいてトランザクションが伝播される仕組みについて触れましたが、その際、ある特定の使用ケースを説明から省いていました。`@Transactional(readOnly=true, propagation=Propagation.REQUIRED)` というアノテーションを付けられたメソッドが、`@Transactional(readOnly=false, propagation=Propagation.REQUIRED)` というアノテーションが付けられた別のメソッドを呼び出す場合、あるいはその逆の場合には、トランザクションの伝播はどうなるのでしょうか。

Spring Web Flow はこのような使用ケースを単純ながらも洗練された方法で処理します。それは、2 番目のメソッドで `readOnly` 属性値を無視するという方法です。簡単に言うと、読み取り専用として開始されたトランザクションは終了するまで読み取り専用トランザクションとして処理され、読み取り/書き込みトランザクションとして開始されたトランザクションは終了するまで読み取り/書き込みトランザクションとして処理されます。

このことは、フロー・マネージド・パーシスタンスでトランザクションを使用しないようにするべきか、それとも読み取り専用トランザクションを使用するべきか、という問題に対する興味深い答えとなっています。

読み取り専用トランザクションの使用ケース

アプリケーションのサービス層にある Spring Bean は、いくつかの JAX-WS/JAX-RS アノテーションによって、再利用可能な SOAP/REST Web サービスとして公開することができます。これらの `@Service` Bean やそのメソッドに `@Transactional` を適用すると、Web サービスの呼び出しがデータベース・トランザクションにバインドされます (アプリケーションの階層アーキテクチャーに問題があり、他にトランザクション・プロパティーを指定する場所がないという場合を除き、DAO の `@Repository` Bean で `@Transactional` を使用する正当な理由はありません)。

ここでもう一度、Spring Web Flow での非トランザクション・データ・アクセスによるフロー・マネージド・パーシスタンスの手法について考えてください。`@Transactional` を Web サービス対応 `@Service` Bean に適用すると、非トランザクション・コンテキストが無効にされる可能性があります。その場合、メソッド呼び出しチェーンの中でサービス層に指定された読み取り/書き込みトランザクションが行われると、フロー・スコープド・パーシスタンス・コンテキストで保留されているすべてのデータ変更がフラッシュされ、いわゆる「早期フラッシュ」という結果に至ります。

早期フラッシュを回避すること

挿入操作 (つまり、ID 列) でエンティティー ID が生成される場合、手動フラッシュに設定しているとしても、`entityManager.persist()` または `entityManager.merge()` メソッドの呼び出し後にフラッシュが行われます。これらのメソッド呼び出し後にフラッシュが必要となるのは、パーシスタンス・コンテキストの管理対象 (永続) エンティティーごとに ID を割り当てなければならないためです。このような早期フラッシュが行われないようにするには、ID 生成ストラテジーを `sequence` に設定する必要があります。

その一方、ビュー層の Spring Bean に `@Transactional(readOnly=true)` を指定すると、これらのサービス Bean での読み取り/書き込みトランザクションの設定が無効にされます。したがって、トランザクションが読み取り専用のままとなり、早期フラッシュは回避されます。SOAP/REST Web サービス通信で Web 層全体が迂回される場合には、サービス Bean に適用された `@Transactional` アノテーションによって、Web サービス呼び出しが確実にデータベース・トランザクション内で実行されるようになります。

これが、フロー・マネージド・パーシスタンスで非トランザクション・データ・アクセスではなく、読み取り専用トランザクションを使用する場合の大きな利点です。

前述のとおり、フロー・マネージド・パーシスタンスは、アトミック Web フローを伴う使用ケースを解決します。記事の残りでは、今度は非アトミック Web フローを必要とする使用ケースに焦点を当てます。非アトミック Web フローにフロー・マネージド・パーシスタンスは適用されませ

んが、使用ケースによっては、フロー・スコープド・パーシスタンス・コンテキストのオブジェクトを使用できることに注目してください。

非アトミック Web フロー

ビジネス・プロセス管理 (BPM) の観点から見ると、長期にわたって実行されるプロセスは、標準的な Web セッションの存続期間よりも長く存続します。このような長期実行プロセスにヒューマン・タスクが関わっているとしたら、ユーザーが任意の時間枠でプロセスを処理し、その数時間後、数日後、さらには数ヶ月後にプロセスの実行を再開するという場合が考えられます。当然、このようなプロセスはサーバー・クラッシュにも持ちこたえられなければなりません。

こうした長期実行プロセスに特有のあらゆる要素が示すように、長期実行プロセスでは、プロセスの段階が進むごとに、その状態をバックエンド・データベースに永続化する必要があります。確実な技術的ソリューションの 1 つは、長期実行プロセスのヒューマン・アクティビティーを Web フローとして実装することです。このフローは異なる Web セッションで繰り返し実行され、長期実行プロセスのライフサイクルをエミュレートします。

上記のシナリオとは別に、一部のアプリケーションは、ユーザーが任意にページ間をナビゲートできる、コンテキストのない Web ページで構成されています。このような Web ページは、論理的に連続したフローがなかったり、あるいはフローに開始状態や終了状態がなかったりするかもしれませんが、そうした場合でもビジネス機能に応じて複数のフローにグループ化するという方法が考えられます。そして、すべてのユーザー・リクエストの実行中に行われたデータの変更は保存しなければなりません。こうしたアプリケーションでのパーシスタンス・プログラミングは、トランザクションのアトミック性が一連のユーザー・アクション (Web フロー) ではなく各ユーザー・アクションを対象範囲とするという点では、前述の長期実行プロセスと変わりありません。

非アトミック Web フローの使用ケース

医療業界では、医療サービス・プロバイダーが定期的に慢性疾患患者と連絡を取って健康状態と潜在するリスクを評価し、その結果に応じて、医療および生活面での健康上のアドバイスを行います。これは、ケース・マネージメントと呼ばれています。

ケース・マネージメント・システムが重点を置いているのは、一連の連絡タスクです。典型的なタスクでは、ケース・マネージャーが患者への電話連絡や、患者の状態を判断するための質問、その答えに応じた適切なアドバイス、医療照会用の情報の取りまとめ、連絡結果の記録、そしてフォローアップ・タスクの準備を行います。

厄介な問題は数多くあります。例えば、患者の状態を判断するための質問リストが長すぎたり、さまざまな理由で電話での会話が中断されたり、医療照会を行うための情報が記録されていないために特定の業務を完了できなかったりするなどです。並行して行われる作業や非同期の作業が含まれる連絡タスクは長期実行プロセスであり、各段階での進捗をデータベースに保存しなければなりません。このような連絡タスクは 1 つの Web フローとしてエミュレートすることができ、長期実行プロセスの過程でフローを繰り返し開始して実行することができます。

この非アトミック Web フローのシナリオは、Spring Web Flow のドキュメントには記載されていません。それでも、この使用ケースでフロー・スコープド・パーシスタンス・コンテキストのオブジェクトを利用できるのでしょうか。その答えは、イエスです。

トランザクションのスコープの指定

すでに説明したように、フロー定義ファイルの `<persistence-context/>` タグによって、パーシスタンス・コンテキストにスレッドをバインドし、`flowScoped` パーシスタンス・コンテキストにすれば、エンティティーが分離されることも、`LazyInitializationException` が発生することもありません。したがって、このタグはそのまま使用することにします。アトミック・フローでのフロー・マネージド・パーシスタンスと最も顕著に異なる点は、トランザクションのスコープです。アトミック性はフロー全体ではなく、プロセスの各ステップに適用されます。ほとんどの場合、プロセスでのアトミック単位は、Web フロー定義の `<transition>` タグによって表されるユーザー・アクションとなります。

残念なことに、Spring Web Flow は `<transition>` や `<evaluate>` をはじめとする、いずれのタグでもトランザクション境界をサポートしません。このため、開発者にとっての次の選択肢は、Spring Bean のメソッドに `@Transactional` アノテーションを付け、そのメソッドを `<evaluate>` タグから呼び出してデータベース・トランザクションを開始することです (`<transition>` タグはメソッド呼び出しをサポートしません)。

基本的に、このトランザクションのスコープは、フロー内の `<evaluate>` タグに設定されます。`@Transactional(readyOnly=false)` を適用すると JPA/Hibernate の `FlushMode` が `AUTO` に設定されるため、Hibernate が、同じトランザクションのコンテキスト内でデータ変更をフラッシュするタイミングを決定します。ここで説明しているような使用ケースでは、プログラミングの便宜上、そして SQL を最適化するという理由から、手動でのフラッシュよりも自動フラッシュが推奨されます。注意する点として、1 つの `<transition>` には複数の `<evaluate>` タグを含められるため、ユーザー・アクションごとに複数のデータベース・トランザクションが行われる結果となります。

ユーザー・アクション/リクエストのそれぞれがアトミックである場合 (通常はアトミックです)、すべてのデータベース書き込み操作を 1 つの Spring Bean の 1 つの `@Transactional` メソッド内にまとめ、これらの操作が同じトランザクション・コンテキストにバインドされて、同じ `<evaluate>` タグによって呼び出されるようにする必要があります。そのため、アトミック・リクエストにはリスト 1 のようにトランザクション・コンテキストを指定します。

リスト 1. アトミック・ユーザー・アクションに対して指定したトランザクション・コンテキスト

```
<transition>
  <evaluate expression="beanA.readAlpha()"/>
  <evaluate expression="beanA.readBeta()"/>
  <evaluate expression="beanB.readGamma()"/>
  <evaluate expression="beanA.writeAll()"/> <!-- a single read/write transaction -->
  <evaluate expression="beanB.readEta()"/>
</transition>
```

リスト 2 に示す典型的な例では、複数の読み取り/書き込みトランザクション (それぞれ独自にコミットまたはロールバックするトランザクション) が同じユーザー・リクエストで呼び出されるた

め、その結果、このユーザー・リクエストは非アトミックとなります。これは、大部分の開発シナリオでは壊滅的な事態です。

リスト 2. 非アトミック・ユーザー・アクションに対して指定したトランザクション・コンテキスト

```
<transition>
  <evaluate expression="beanA.readAlpha()"/>
  <evaluate expression="beanA.readBeta()"/>
  <evaluate expression="beanB.readGamma()"/>
  <evaluate expression="beanA.writeDelta()"/> <!-- read/write transaction -->
  <evaluate expression="beanA.writeEpsilon()"/> <!-- read/write transaction -->
  <evaluate expression="beanB.writeZeta()"/> <!-- read/write transaction -->
  <evaluate expression="beanB.readEta()"/>
</transition>
```

このように同じ `<transition>` に含まれる他の `<evaluate>` タグによって参照される読み取り専用操作を処理するにはどうすればよいのでしょうか。それには、3 つの方法があります。

1. 前に説明したように、データベース・トラザクションを使わずに読み取り専用操作を実行する。
2. これらの操作に `@Transactional(readOnly=false)` というアノテーションを付け、SQL クエリーがデータベースの読み取り/書き込みトランザクションで実行されるようにする。この場合、フロー・スコープド・パーシスタンス・コンテキストの `FlushMode` は常に `AUTO` になります。
3. これらの操作に `@Transactional(readOnly=true)` というアノテーションを付ける。この場合、`FlushMode` はこれらの読み取り専用トランザクションでは `MANUAL` に設定され、読み取り/書き込みトランザクションに達すると `AUTO` に変更されます。

JPA/Hibernate は、読み取り/書き込みトランザクションがコミットされる前に、パーシスタンス・コンテキストで保留中の変更を自動的にフラッシュします。簡単のため、Hibernate チームはこのような場合には、すべてのデータ操作に一貫して読み取り/書き込みトランザクションを適用するよう開発者に推奨しています。したがって、`@Transactional` を適用する場所にはすべて、`readOnly=false` を設定してください。

リクエストごとに 1 つのトランザクションにした場合の問題

皆さんは、どうして各ユーザー・リクエストを 1 つの大きなトランザクションでカプセル化しないのか疑問に思われるかもしれません。問題は、リクエスト単位でのトランザクションでは、ビューのレンダリングが完了するまで関連するデータベースのロックが開いたままになってしまうことです。残念ながら、ビューのレンダリングには、ベースとなるデータベースのパフォーマンスやスケーラビリティーに影響を及ぼすほど時間のかかるブロッキング I/O 操作が伴います。

予期せぬ OptimisticLockingFailureException

非アトミック Web フローでフロー・スコープド・パーシスタンス・コンテキストを使用すると、予期せぬ `OptimisticLockingFailureException` が発生することがあります。

各ユーザー・アクションのデータ保全性を守るためには、非アトミック Web フローにもオプティミスティック・ロックを使用することを是非ともお勧めします。エンティティーの `@Version` フィールドに、データベースによって生成された整数またはタイムスタンプが設定されてい

る場合、更新操作の後にはエンティティーに対して明示的にクエリーを実行して、パーシスタンス・コンテキストでのエンティティーの状態を更新する必要があります。これを行わないと、`@Version` フィールドには古い値が設定されたままになり、このエンティティーに対して別のトランザクションで更新が行われると、`OptimisticLockingFailureException` が発生する結果となります。皮肉なことに、この例外は複数のユーザーに対して同時に発生しません。逆に、アトミック・フローでは、この更新後のクエリー操作を行わないでください。アトミック・フローで行うと、早期フラッシュが発生することになるからです。結局のところ、アトミック Web フローではエンティティー・オブジェクトがメモリー内で何度更新されようと、フローの終了時に行われる SQL フラッシュで、ようやくエンティティー・インスタンスの最終状態がわかるようになります。

アトミック Web フローでも、非アトミック Web フローでも、フロー・スコープド・パーシスタンス・コンテキストがパーシスタンス・プログラミングを円滑かつ単純にすることは明らかです。Web フローでフロー・スコープド・パーシスタンス・コンテキストのオブジェクトを使わずにパーシスタンスをプログラミングすることもできますが、その場合、さまざまな困難と落とし穴があります。

フロー・スコープド・パーシスタンス・コンテキストを使用しないパーシスタンス・プロブラミング

「Hotel Booking (ホテル予約)」サンプル・アプリケーションが実証しているように、場合によっては `<persistence-context/>` タグを使わずに Web フローを実装することも可能ですが、その場合の影響は、アトミック Web フローで最も顕著に現れます。それは、フロー・スコープド・パーシスタンス・コンテキストのオブジェクトを省略すると、アトミック Web フローを実現できなくなることです。以降のセクションでは、これ以外の不都合な影響について説明します。

スコープをデータベース・トランザクションに設定したパーシスタンス・コンテキスト

フロー・スコープド・パーシスタンス・コンテキストを使用しない場合、`@PersistenceContext` アノテーションによって注入されたパーシスタンス・コンテキストのスコープは、デフォルトでデータベース・トランザクションに設定されます。このことが問題になる理由を理解するには、「Hotel Booking (ホテル予約)」サンプル・アプリケーションから抜粋した以下のコード・スニペットを見てください。

リスト 3. 「Hotel Booking (ホテル予約)」での「メイン・フロー」定義の一部

```
<view-state id="enterSearchCriteria">
  <on-render>
    <evaluate expression="bookingService.findBookings(currentUser.name)"
      result="viewScope.bookings" result-type="dataModel" />
  </on-render>
  <transition on="cancelBooking">
    <evaluate expression="bookingService.cancelBooking(bookings.selectedRow)" />
    <render fragments="bookingsFragment" />
  </transition>
</view-state>
```

リスト 3 で参照されている `cancelBooking` メソッドが、例えば以下のように定義されているとします。

リスト 4. cancelBooking メソッド

```
@Service("bookingService")
@Repository
public class JpaBookingService implements BookingService {

//...

@Transactional
public cancelBooking(Booking booking){

  if (booking != null) {
    em.remove(booking);
  }
}
```

このように定義されている場合、このコードを実行すると以下のエラーが発生します。

```
Caused by: java.lang.IllegalArgumentException: Removing a detached instance
```

`<on-render>` タグから返される `booking` エンティティーは、その次に実行されるアクション `<transition on="cancelBooking">` では分離された状態になります。同じ `bookingService` Bean の `findBookings` メソッドと `cancelBooking` メソッドは、それぞれに異なるデータベース・トランザクションで実行されるため、2 つの別個のパーシスタンス・コンテキストのオブジェクトに関連付けられます。したがって、一方のパーシスタンス・コンテキストで管理される `booking` エンティティーは、もう一方のパーシスタンス・コンテキストの観点からは分離されているというわけです。

この問題を回避するには、リスト 5 に記載する実際の `cancelBooking` メソッドの中で、`booking` エンティティーが削除される前に、エンティティーの主キーによって同じエンティティーを再ロードします。

リスト 5. 修正後の cancelBooking メソッド

```
@Service("bookingService")
@Repository
public class JpaBookingService implements BookingService {

//...

@Transactional
public cancelBooking(Booking booking){

  booking = em.find(Booking.class, booking.getId()); // reinstate the persistent entity

  if (booking != null) {
    em.remove(booking);
  }
}
```

トランザクション・スコープ・パーシスタンス・コンテキストは事実上、`singleSession=false` が設定された `OpenSessionInViewFilter/Interceptor` と同じように機能します。つまり、同じリクエスト内の各トランザクションに、それぞれ独自のセッションが関連付けられるということです。ただし、Open Session in View の「遅延クローズ・モード」のメリットは失われます。

トランザクション・スコープ・パーシスタンス・コンテキストが各トランザクションの完了直後にクローズされることから、ビュー・レンダリング中の遅延読み取りによって `LazyInitializationException` が発生します。`OpenSessionInViewInterceptor` / `OpenEntityManagerInViewInterceptor` のようなものを実装するという方法もありますが、Spring Core が提供する方法をそのまま使用しても、Spring Web Flow では機能しません。それよりも、組み込みフロー・スコープド・パーシスタンス・コンテキストのオブジェクトを使用したほうが遥かに簡単です。

スコープを各呼び出しに設定したパーシスタンス・コンテキスト

フロー・スコープド・パーシスタンス・コンテキストの助けを借りない非トランザクション・データ・アクセスは、最悪のシナリオです。この方法は、可能な限り使用しないでください。

トランザクションの外部では、パーシスタンス・コンテキストのスコープが各呼び出しに設定され、`FlushMode` は `AUTO`、`auto-commit` は `true` になります (非トランザクション・データ・アクセスの場合、Hibernate は自動フラッシュを無効に設定することを思い出してください)。つまり、`@PersistenceContext` によって注入された同じパーシスタンス・コンテキストのプロキシーでメソッドを呼び出すと、呼び出しごとに異なるエンティティー・マネージャーのインスタンスが返されます。これらのインスタンスはオープンした後すぐにクローズします。

基本的に、エンティティー・マネージャーは「短いトランザクション」をスコープとすることから、リスト 5 のコード・スニペットを実行すると、以下のエラー・メッセージを受け取ります。

```
java.lang.IllegalArgumentException: Removing a detached instance
```

異なるフロー間でのエンティティーの受け渡し

最後に、場合によっては問題を引き起こすシナリオとして、異なるフローにエンティティーを渡さなければならない場合について説明します。

フロー・スコープド・パーシスタンス・コンテキストのオブジェクトは、フローのスコープに従います。そのため、あるフローから別のフローにエンティティーが渡されると、途端に分離された状態になってしまいます。この問題を解決するには、これらのエンティティーを、その主キーを使って現行フローのパーシスタンス・コンテキストにマージ/再アタッチするか再ロードします。このストラテジーは、Open Session in View の手法を反映したものです。

まとめ

Spring Web Flow は高度な Web 開発フレームワークとして、独特の機能によって JPA/Hibernate でのパーシスタンス・プログラミングおよびトランザクション管理をサポートします。この記事では、Java 開発者が Spring Web Flow アプリケーションをプログラミングする際に直面する複雑な問題と課題について詳しく調べました。

この記事で説明したような実際の使用ケースから私が引き出した、Spring Web Flow でトランザクション・アトミックおよび非アトミック Web アプリケーションをコーディグする際の「経験則」を以下に紹介します。

- 第一に優先する方法として、常にフロー・スコープド・パーシスタンス・コンテキストを使用すること
- アトミック Web フローで参照されるすべてのメソッドには例外なく、読み取り専用トランザクションを適用すること
- 非アトミック Web フローで参照されるすべてのメソッドには例外なく、読み取り/書き込みトランザクションを適用すること

著者について

Xinyu Liu

Xinyu Liu は Sun Microsystems 認定エンタープライズ・アーキテクトとして、Java EE、Java SE、および Java ME プラットフォームでのアプリケーション設計および開発を専門に経験を積んできました。George Washington University で学位を取得した彼は、現在、医療関係の企業で IT 部門の主要スタッフとして活躍しています。Liu 博士は、Java.net および JavaWorld.com で JSF、Spring Security、Hibernate Search、Spring Web Flow、Servlet 3.0 仕様などに関する執筆活動を行っている他、Packt Publishing では『Spring Web Flow 2 Web Development』、『Grails 1.1 Web Application Development』の校正も行いました。この記事は、彼の初めてのIBM developerWorks 記事です。