継続的リアルタイム・データ・プロファイリングに Drools と JPA を適用する

JPA POJO を Drools 5 のワーキング・メモリーにファクトとしてプログラミングする

Xinyu Liu
VP Product Development
eHealthObjects

2012年 8月 02日

Drools は、JPA と Spring をベースとしたアプリケーション・コードに統合することができます。そのために、介入的な形で命令型プログラミングを使用する必要はありません。この記事では、リアルタイムのシステム監視および継続的データ・プロファイリング・プロセスに、POJO を使用してビジネス要件をコスト効果の高い方法でプログラミングする方法を説明します。Drools 5 を使用してメモリー効率の高い安定したアプリケーションを構築するための高度なヒントを含め、著者の Xinyu Liu が Java パーシスタンスおよびビジネス統合技術で培った専門知識を皆さんと分かち合います。

複雑なワークフロー、ビジネス・ルール、ビジネス・インテリジェンスの管理を任された企業の開発者は、ワークフロー・エンジン、エンタープライズ・サービス・バス (ESB)、そしてルール・エンジンを統合するエンタープライズ・プラットフォームの価値をすぐに実感するようになります。これまでのところ、IBM WebSphere Process Server/WebSphere Enterprise Service Bus (「参考文献」を参照) や Oracle SOA Suite などの商用製品が、この価値あるエンタープライズ・プラットフォームを独占していました。そのような商用製品に代わるオープンソースの手段が、JBoss Community の Drools 5 です。Drools 5 は一連の統一された API とステートフルな共有ナレッジ・セッションにより、jBPM ワークフロー・エンジンとルール・エンジンをシームレスに統合します。

入門者向けの参考情報

この記事では、読者が Spring プラットフォーム、Java Persistence API、および Drools の基礎知識を十分理解していることを前提としています。これらのトピックについて基礎から学ぶには、「参考文献」で紹介している入門記事を参照してください。

Drools 5 ビジネス・ロジック統合プラットフォームの主要な構成要素は、Drools Expert と Drools Fusion です。この 2 つのモジュールが Drools 5 プラットフォームのルール・エンジンおよびインフラストラクチャーを構成し、複雑なイベント処理/時間推論に対処します。この記事のサンプ

ル・アプリケーションは、これらのコア・モジュールによって作成されています。Drools 5 で使用できるその他のパッケージの詳細を調べるには、「参考文献」を参照してください。

Drools 5 での POJO

POJO (Plain Old Java Object) が初めて明らな形で実装されたのは、Spring フレームワークです。POJO は、依存性注入 (DI) とアスペクト指向プログラミング (AOP) に加え、単純さへの回帰を表明しました。Spring を事実上 Web アプリケーション開発の業界標準にまで押し上げたのは、この単純さです。POJO の採用は Spring から EJB 3.0 および JPA に広がり、そこからさらに JAXB や XStream などの XML と Java のバインディング技術へと広がっていきました。最近では、全文検索エンジンの Lucene に、Hibernate Search を介して POJO が統合されました (「参考文献」を参照)。

このように POJO が徐々に広まった結果、現在ではアプリケーションの POJO データ・モデルを複数の層に伝播して、Web ページや SOAP/REST Web サービスポイントから直接公開できるようになっています。プログラミング・モデルとしての POJO は、コスト効果が高く、非介入的なモデルであるため、開発者の時間を節約するとともに、エンタープライズ・アーキテクチャーを単純化します。

現在の Drools 5 では、POJO プログラミングの単純さを次のレベルへと引き上げるために、プログラマーが POJO をファクトとして直接ナレッジ・セッションに挿入できるようになりました。ナレッジ・セッションとは、ルール・エンジン用語で「ワーキング・メモリー」と呼ばれるものです。この記事では、JPA エンティティーをファクトとして Drools のワーキング・メモリーで操作する、コスト効果の高い非介入的な手法を紹介します。この手法は、継続的リアルタイム・データ・プロファイリングを未だかつてないほど容易にします。

課題とする Drools プログラミング

多くの医療プロバイダーは診療記録 (診療、処方箋、診断) を追跡するために、コスト効果の高い方法として症例管理システムを使用しています。このようなシステムが、この記事のサンプル・プログラムのベースとなっています。このプロラグムのフローと要件は、以下のとおりです。

- システム内では、症例がすべての臨床医に回覧されます。
- 臨床医は、少なくとも週に 1 件の診断タスクに責任を持ちます。そうでない場合、臨床医の監督者に通知が送信されます。
- 臨床医の診断タスクは、システムが自動的にスケジューリングします。
- ある症例が 30 日間を過ぎても診断されない場合、その症例グループのすべての臨床医に督促状が送信されます。
- 応答がない場合、システムはそのビジネス・ルールで定義されたアクションを行います。例えば、臨床医グループに診断の遅れを通知して、別のスケジュールを提案するなどのアクションです。

このユース・ケースに対してビジネス・プロセス・マネジメント (BPM) のワークフローとルール・エンジンを選ぶのは理に適っています。システムがデータ・プロファイリング/分析ルール (上記のリストで斜体になっている項目) を使用すれば、各ケースを jBPM で長期間実行されるプロセス/ワークフローとして扱い、自動スケジューリング要件には Drools Planner で対処できるためです。この記事では、プログラムのビジネス・ルールに焦点を当てます。また、ルールの条件に

適合した場合には、督促状と通知が瞬時に生成されることをシステムがリアルタイムで要求するという前提です。したがって、これは、継続的リアルタイム・データ・プロファイリングのユース・ケースということになります。

リスト 1 に、システムで宣言する 3 つのエンティティー・クラス、`MemberCase`、`Clinician`、および `CaseSupervision` を記載します。

リスト 1. エンティティー・クラス

```
@Entity
@EntityListeners({DefaultWorkingMemoryPartitionEntityListener.class})
public class MemberCase implements Serializable
{
  private Long id; // pk
  private Date startDtm;
  private Date endDtm;
  private Member member; // not null (memberId)
  private List<CaseSupervision> caseSupervisions = new ArrayList<CaseSupervision>();
  //...
}

@Entity
@EntityListeners({DefaultWorkingMemoryPartitionEntityListener.class})
public class Clinician implements Serializable
{
  private Long id; // pk
  private Boolean active;
  private List<CaseSupervision> caseSupervisions = new ArrayList<CaseSupervision>();
 //...
}

@Entity
@EntityListeners({SupervisionStreamWorkingMemoryPartitionEntityListener.class})
public class CaseSupervision implements Serializable
{
  private Long id; // pk
  private Date entryDtm;
  private MemberCase memberCase;
  private Clinician clinician;
  //...
}
```

`MemberCase` の各インスタンスは患者の症例を表し、`Clinician` は機関に所属する臨床医を表します。臨床医が症例の診断を行うたびに、`CaseSupervision` レコードが生成されます。この 3 つすべてのエンティティーが、これからビジネス・ルールに定義するファクト型です。上記の `CaseSupervision` は、Drools ではイベント型として宣言されることに注意してください。

アプリケーションの観点からは、この 3 つのタイプのエンティティーを、システム内のどこからでも、さまざまな画面とさまざまなワークフローで変更できるようにすることも可能です。さらに、Spring Batch のようなツールを使ってエンティティーを一括更新することもできますが、このサンプルでは JPA パーシスタンスのコンテキストからのみ、エンティティーを更新することを前提とします。

このサンプル・アプリケーションは、Spring と Drools の統合によるものであり、ビルドには Maven を使用することに注意してください。構成の詳細については後から検討しますが、ソースの zip ファイルは随時ダウンロードすることができます。まずは、Drools 5 の操作に関する概念的特徴について検討しましょう。

ファクトと FactHandle

ファクトとは、ルールが推論対象とするデータ・オブジェクトであるというのが、ルール・エンジンの一般概念です。Drools でのファクトは、アプリケーションから取得して、ルール・エンジンのワーキング・メモリーにアサートする任意の Java Bean です。あるいは、JBoss Drools のリファレンス・マニュアルでは以下のように説明しています。

> ルール・エンジンがファクトを「複製」することはありません。ファクトとは、1 日が終わる時点でのすべての参照/ポインターであり、ファクトがアプリケーションのデータとなります。ゲッターおよびセッターのないストリングやその他のクラスは有効なファクトではありません。そのようなクラスは、ゲッターとセッターの JavaBean 標準を使用してオブジェクトを操作するフィールド制約では使用することはできません。

ルールにキーワード `no-loop` または `lock-on-active` を指定しない限り、ワーキング・メモリー内でファクトが変更されるたびに、Drools ルール・エンジンのルールが再評価されます。ただし、`@PropertyReactive` および `@watch` アノテーションを使用すれば、Drools に変更を監視させるファクトのプロパティーを指定することもできます。その場合、Drools は、ファクトのプロパティーのうち、指定された以外のプロパティーの更新はすべて無視します。

真の保守を実現するために Drools ワーキング・メモリー内のファクトを安全に更新するには、次の 3 つの方法があります。

1. Drools の構文では、ルールの右側 (Right-Hand Side: RHS) がアクション/結果の部分です。RHS は、modify ブロック内で更新することができます。アクティブにされたルールの結果としてファクトを変更する際には、この方法を使用してください。
2. Java クラスで FactHandle を使用して、ファクトを外部で変更します。この方法は、アプリケーションの Java コードによってファクトが変更される場合に使用します。
3. Fact クラスに JavaBeans 仕様で定義された PropertyChangeSupport を実装させます。この方法を使用して、Drools を PropertyChangeListener として Fact オブジェクトに登録します。

サンプルで使用するのは傍観者的なルールです。つまり、Drools ワーキング・メモリー内の JPA エンティティー・ファクトを更新することはせず、推論の結果として論理ファクトを生成します (リスト 6 を参照)。ルールで JPA エンティティーを更新する場合には、特に注意が必要です。更新されたエンティティーが分離した状態になるか、あるいは現行スレッドにトランザクションが関連付けられていなかったり、トランザクションが関連付けられているとしても、それが読み取り専用であったりすることが考えられるためです。その場合、エンティティーで行われた変更はデータベースに保存されません。

ファクト・オブジェクトは参照渡しであるため、(JPA/Hibernate とは異なる) Drools は、ルール外部で行われたファクトの変更を追跡することができません。ルールの推論結果が矛盾しないようにするには、`FactHandle` を使用して、Drools にアプリケーションの Java コードで行われたファクトの変更を通知するという方法を採ることができます。この場合、Drools はその通知に応じて、ルールを再評価します。`FactHandle` とは、ワーキング・メモリーにアサートされたファクト・オブジェクトを表すトークンです。ファクトを変更または削除するときのワーキング・メモリーの操作では、一般にこのトークンを使用することになります。サンプル・アプリケーション (リスト

2 およびリスト 3) では、`FactHandle` を使用してワーキング・メモリー内のエンティティー・ファクトを操作します。

Drools がファクトの変更を追跡できないという問題に対処するには、`PropertyChangeSupport` を実装することもできます (このオブジェクトは、Bean のプロパティーで行われたすべての変更を取り込みます)。ただし、この方法を使用すると、頻繁にルールが再評価されることになるため、パフォーマンス・ヒットへの対処が必要であることを忘れないでください。

JPA エンティティーをファクトとして使用する

POJO ファクトを使用して、JPA エンティティーをドメイン・データ・オブジェクトとして Drools のワーキング・メモリーに挿入することができます。こうすれば、Value Object/DTO 層でも、JPA エンティティーと DTO との間の対応する変換層でも、データをモデル化する必要がなくなります。

エンティティーをファクトとして使用すると、アプリケーション・コードは単純化されますが、エンティティーのライフサイクル・フェーズに対して特別な注意を払わなければならなくなります。ファクトとしてのエンティティーは、管理状態 (永続) または分離状態のいずれかに維持しなければなりません。過渡的エンティティーはデータベースにまだ保存されていないエンティティーであるため、Drools のワーキング・メモリーに過渡的エンティティーを挿入することは厳禁です。同様に、データベースから削除済みのエンティティーは、ワーキング・メモリーから削除してください。そのようにしなかった場合には、アプリケーション・データベースとルール・エンジンのワーキング・メモリーが同期しなくなります。

このことから、正解すると百万ドルの賞金に値するほどの難しい質問が持ち上がってきます。それは、「アプリケーション・コードで行われたエンティティーの変更を、`FactHandle` によって効率的にルール・エンジンに通知するにはどうすればよいのか？」という質問です。

命令型プログラミングと AOP の違い

この質問の答えを出すために、命令型プログラミングの考え方で取り組むとしたら、対応する JPA API メソッドに続き、ナレッジ・セッションで `insert()`、`update()`、`retract()` メソッドを呼び出すという結論に至ることでしょう。この方法は Drools API を介入的に使用するものであり、アプリケーションにスパゲッティー・コードを残すことになります。さらに悪いことに、読み取り/書き込みトランザクションの終了時には、パーシスタンス・コンテキストを明示的に呼び出さなくても、JPA の更新された (ダーティー) エンティティーがデータベースと同期されます。これらの変更をインターセプトして Drools に通知するにはどうすればよいのでしょう？別の方法として、典型的なビジネス・インテリジェンス (BI) ツールが行うように、別のプロセスでエンティティーの変更をポーリングすれば、コア・ビジネス機能に手を加えることはありません。けれども、この方法は難しく、実装するのにコストがかかり、しかも瞬時に結果を得ることができません。

このようなケースによく適しているのは、一種の AOP インターセプターである JPA の `EntityListener` です。リスト 2 では、2 つの `EntityListener` を定義し、これらがサンプル・アプリケーションの 3 つのタイプのエンティティーに対する変更をすべてインターセプトするようにします。この手法は、JPA でのエンティティーのライフサイクルを、Drools でのそのライフサイクルと同期した状態に維持します。

エンティティー・ライフサイクル・コールバック・メソッドの中では、特定のエンティティー・インスタンスに対応する `FactHandle` を探して取得した後、JPA ライフサイクルのフェーズに応じて、返された `FactHandle` でファクトを更新または削除します。`FactHandle` がない場合は、エンティティーを更新または永続化するために、エンティティーが新しいファクトとしてワーキング・メモリーに挿入されます。エンティティーはワーキング・メモリー内に実在しないことから、JPA の delete が呼び出される際に、エンティティーをワーキング・メモリーから削除する必要はありません。リスト 2 に記載する 2 つの JPA `EntityListener` は、ワーキング・メモリーへの 2 つの異なるエントリー・ポイント (パーティション) に対応します。最初のエントリー・ポイントは `MemberCase` と `Clinician` の間で共有されます。2 つ目のエントリー・ポイントの対象は、`CaseSupervision` イベント型です。

リスト 2. EntityListener

```
@Configurable
public class DefaultWorkingMemoryPartitionEntityListener
{
  @Value("#{ksession}") //unable to make @Configurable with compile time weaving work here
  private StatefulKnowledgeSession ksession;

  @PostPersist
  @PostUpdate
  public void updateFact(Object entity)
  {
    FactHandle factHandle = getKsession().getFactHandle(entity);
    if(factHandle == null)
      getKsession().insert(entity);
    else
      getKsession().update(factHandle, entity);
  }

  @PostRemove
  public void retractFact(Object entity)
  {
    FactHandle factHandle = getKsession().getFactHandle(entity);
    if(factHandle != null)
      getKsession().retract(factHandle);
  }

  public StatefulKnowledgeSession getKsession()
  {
    if(ksession != null)
    {
      return ksession;
    }
    else
    {
      // a workaround for @Configurable
      setKsession(ApplicationContextProvider.getApplicationContext()
        .getBean("ksession", StatefulKnowledgeSession.class));
      return ksession;
    }
  }
  //...
}

@Configurable
public class SupervisionStreamWorkingMemoryPartitionEntityListener
{
  @Value("#{ksession}")
  private StatefulKnowledgeSession ksession;

  @PostPersist
```

```
  // CaseSupervision is an immutable event,
  // thus we don’t provide @PostUpdate and @PostRemove implementations.
  public void insertFact(Object entity)
  {
    WorkingMemoryEntryPoint entryPoint = getKsession()
      .getWorkingMemoryEntryPoint("SupervisionStream");
    entryPoint.insert(entity);
  }
  //...
}
```

AOP と同じように、リスト 2 の `EntityListener` 手法は、システムのコア・ビジネス・ロジックには手を加えません。この手法では、1 つ以上の Drools グローバル・ナレッジ・セッションを 2 つの `EntityListener` に注入する必要があることに注意してください。ナレッジ・セッションは、後でシングルトン Spring Bean として宣言します。

ヒント: 共有グローバル・ナレッジ・セッション

この EntityListener 手法をシステム全体の BI データ・プロファイリングおよび分析要件に適した手法にしているのは、基本的に、共有グローバル・ナレッジ・セッションです。共有グローバル・ナレッジ・セッションは、オンライン・ショッピング・システムで使用されるようなユーザー固有のプロセスおよびルール実行には向きません。オンライン・ショッピング・システムでは、ユーザー固有のデータを処理し、その後データを破棄するために、通常はナレッジ・セッションがオンザフライで生成されます。

ワーキング・メモリーの初期化

アプリケーションが起動されると、3 つのタイプのエンティティーの既存のレコードのすべてが、ルールを評価するためにデータベースからワーキング・メモリーにプリロードされます (リスト 3 を参照)。それ以降は、2 つの `EntityListener` を介して、エンティティーに対して行われたあらゆる変更がワーキング・メモリーに通知されます。

リスト 3. ワーキング・メモリーの初期化および Drools クエリーの実行

```
@Service("droolsService")
@Lazy(false)
@Transactional
public class DroolsServiceImpl
{
  @Value("#{droolsServiceUtil}")
  private DroolsServiceUtil droolsServiceUtil;

  @PostConstruct
  public void launchRules()
  {
    droolsServiceUtil.initializeKnowledgeSession();
    droolsServiceUtil.fireRulesUtilHalt();
  }

  public Collection<TransientReminder> findCaseReminders()
  {
    return droolsServiceUtil.droolsQuery("CaseReminderQuery",
      "caseReminder", TransientReminder.class, null);
  }

  public Collection<TransientReminder> findClinicianReminders()
  {
    return droolsServiceUtil.droolsQuery("ClinicianReminderQuery",
      "clinicianReminder", TransientReminder.class, null);
  }
}
```

```
@Service
public class DroolsServiceUtil
{
  @Value("#{ksession}")
  private StatefulKnowledgeSession ksession;

  @Async
  public void fireRulesUtilHalt()
  {
    try{
      getKsession().fireUntilHalt();
    }catch(ConsequenceException e)
    {
      throw e;
    }
  }

  public void initializeKnowledgeSession()
  {
    getKsession().setGlobal("droolsServiceUtil", this);
    syncFactsWithDatabase();
  }

  @Transactional //a transaction-scoped persistence context
  public void syncFactsWithDatabase()
  {
    synchronized(ksession)
    {
      // Reset all the facts in the working memory
      Collection<FactHandle> factHandles = getKsession().getFactHandles(
        new ObjectFilter(){public boolean accept(Object object)
        {
          if(object instanceof MemberCase)
            return true;
          return false;
        }
      });
      for(FactHandle factHandle : factHandles)
      {
        getKsession().retract(factHandle);
      }

      factHandles = getKsession().getFactHandles(
        new ObjectFilter(){public boolean accept(Object object)
        {
          if(object instanceof Clinician)
            return true;
          return false;
        }
      });
      for(FactHandle factHandle : factHandles)
      {
        getKsession().retract(factHandle);
      }

      WorkingMemoryEntryPoint entryPoint = getKsession()
        .getWorkingMemoryEntryPoint("SupervisionStream");
      factHandles = entryPoint.getFactHandles();
      for(FactHandle factHandle : factHandles)
      {
        entryPoint.retract(factHandle);
      }

      List<Command> commands = new ArrayList<Command>();
      commands.add(CommandFactory.newInsertElements(getMemberCaseService().findAll()));
      getKsession().execute(CommandFactory.newBatchExecution(commands));
```

```
      commands = new ArrayList<Command>();
      commands.add(CommandFactory.newInsertElements(getClinicianService().findAll()));
      getKsession().execute(CommandFactory.newBatchExecution(commands));

      for(CaseSupervision caseSupervision : getCaseSupervisionService().findAll())
      {
        entryPoint.insert(caseSupervision);
      }

    }
  }

  public <T> Collection<T> droolsQuery(String query, String variable,
    Class<T> c, Object... args)
  {
    synchronized(ksession)
    {
      Collection<T> results = new ArrayList<T>();
      QueryResults qResults = getKsession().getQueryResults(query, args);
      for(QueryResultsRow qrr : qResults)
      {
        T result = (T) qrr.get("$"+variable);
        results.add(result);
      }
      return results;
    }
  }
}
```

fireAllRules() に関する注意点

リスト 3 では、各 `EntityListener` のコールバック・メソッドの中に `fireAllRules()` を呼び出す方法が含まれていましたが、コードを単純化するために、即時ロードされる Spring Bean の `@PostConstruct` メソッド内で、`fireUntilHalt()` メソッドを一度だけ呼び出すようにしました。`fireUtilHalt()` メソッドは、別個のスレッド (Spring の `@Async` アノテーションを参照) で一度だけ呼び出されることになっており、呼び出された後は、halt が呼び出されるまでルール・アクティベーションを起動し続けます。起動するアクティベーションがない場合、`fireUtilHalt` はアクティブ・アジェンダ・グループまたはルール・フロー・グループにアクティベーションが追加されるまで待機します。

アプリケーションの Spring XML 構成ファイル (以下を参照) で、ルールを起動する、あるいはプロセスを開始するという方法を採ることもできましたが、そのように構成しようとするなかで、`fireUntilHalt()` メソッドでのスレッド処理に潜在的問題があることを発見しました。その問題は、ルールの評価中にエンティティーの関係を遅延ロードするときに「データベース接続が閉じられるというエラー」が発生する原因となります (高度な Drools プログラミングに関するトピックを参照)。

Spring と Drools の統合

ここで、Spring と Drools の統合構成の詳細に目を向けましょう。リスト 4 は、アプリケーションの Maven pom.xml から抜粋したスニペットです。ここに、Drools コア、Drools コンパイラー、および Drools Spring 統合パッケージに対する依存関係が組み込まれています。

リスト 4. Maven pom.xml からの抜粋

```
<dependency>
```

```
  <groupId>org.drools</groupId>
  <artifactId>drools-core</artifactId>
  <version>5.4.0.Final</version>
  <type>jar</type>
</dependency>
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-compiler</artifactId>
  <version>5.4.0.Final</version>
  <type>jar</type>
</dependency>
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-spring</artifactId>
  <version>5.4.0.Final</version>
  <type>jar</type>
  <exclusions>
    <!-- The dependency pom includes spring and hibernate dependencies by mistake. -->
  </exclusions>
</dependency>
```

同一性と同等性の違い

リスト 5 で、グローバル・ステートフル・ナレッジ・セッションをシングルトン Spring Bean として構成します (ステートレス・ナレッジ・セッションは、呼び出しの反復中にその状態を維持することはないため、長時間存続するセッションとして機能しません)。リスト 5 で注目すべき重要な設定は、`<drools:assert-behavior mode="EQUALITY" />` です。

JPA/Hibernate では、管理対象エンティティーは同一性 (identity) による比較が行われる一方、分離されたエンティティーは同等性 (equality) による比較が行われます。ステートフル・ナレッジ・セッションに挿入されたエンティティーは、直ちに JPA パースペクティブから分離されます。トランザクションをスコープとするパーシスタンス・コンテキストは、「拡張」されたパーシスタンス・コンテキストや「フロー・スコープ」が設定されたパーシスタンス・コンテキスト (「参考文献」を参照) でさえも、シングルトン・ステートフル・ナレッジ・セッションの存続期間と比べると、短命であるためです。同じエンティティーでも、異なるパーシスタンス・コンテキストのオブジェクトによって取得されるたびに、異なる Java オブジェクトとなります。デフォルトでは、Drools は同一性による比較を行います。したがって、`ksession.getFactHandle(entity)` によって、ワーキング・メモリー内の既存のエンティティー・ファクトで `FactHandle` を探して取得すると、Drools はほとんどの場合に一致を検出できないはずです。分離されたエンティティーと一致させるためには、構成ファイルで `EQUALITY` モードを選択しなければなりません。

リスト 5. Spring の applicationContext.xml からの抜粋

```
<drools:kbase id="kbase">
  <drools:resources>
    <drools:resource  type="DRL" source="classpath:drools/rules.drl" />
  </drools:resources>
  <drools:configuration>
    <drools:mbeans enabled="true" />
    <drools:event-processing-mode mode="STREAM" />
    <drools:assert-behavior mode="EQUALITY" />
  </drools:configuration>
</drools:kbase>
<drools:ksession id="ksession" type="stateful" name="ksession" kbase="kbase" />
```

完全な構成の詳細については、サンプル・アプリケーションのソース・コードを参照してください。

Drools のルール

リスト 6 に、2 つの複合イベント処理 (Complex Event Processing: CEP) ルールを定義します。JPA エンティティーとしての 2 つのファクト型、`MemberCase` と `Clinician` の他に、`CaseSupervision` エンティティー・クラスをイベントとして宣言します。`CaseSupervision` レコードは、臨床医による症例の診断タスクごとに生成されます。生成された後のレコードに変更が行われる可能性は、まずあり得ません。

リスト 6 の Case Supervision ルールの条件では、過去 30 日間に特定の症例に対する診断が行われたかどうかの判断を行います。症例の診断が行われていない場合には、ルールの結果/アクションの部分によって `TransientReminder` ファクト (リスト 7 に定義) が生成され、当然の流れとしてそのファクトがワーキング・メモリーに挿入されます。Clinician Supervision ルールでは、臨床医が過去 7 日以内に少なくとも 1 つの症例の診断を完了しなければならないことを規定しています。この規定に従っていない場合、このルールの結果/アクションの部分によって同様の `TransientReminder` ファクトが生成されます。このファクトも当然の流れとして、ワーキング・メモリーに挿入されます。

リスト 6. 症例診断ルール

```
package ibm.developerworks.article.drools;

import ibm.developerworks.article.drools.service.*
import ibm.developerworks.article.drools.domain.*

global DroolsServiceUtil droolsServiceUtil;

declare Today
  @role(event)
  @expires(24h)
end

declare CaseSupervision
  @role(event)
  @timestamp(entryDtm)
end

rule "Set Today"
  timer (cron: 0 0 0 * * ?)
  salience 99999  // optional
  no-loop
  when
  then
    insert(new Today());
end

rule "Case Supervision"
  dialect "mvel"
  when
    $today : Today()
    $memberCase : MemberCase(endDtm == null, startDtm before[30d] $today)
    not CaseSupervision(memberCase == $ memberCase)
      over window:time(30d) from entry-point SupervisionStream
    then
      insertLogical(new TransientReminder($memberCase, (Clinician)null,
        "CaseReminder", "No supervision on the case in last 30 days."));
end

query "CaseReminderQuery"
  $caseReminder : TransientReminder(reminderTypeCd == "CaseReminder")
end
```

```
rule "Clinician Supervision"
  dialect "mvel"
  when
    $clinician : Clinician()
    not CaseSupervision(clinician == $clinician)
      over window:time(7d) from entry-point SupervisionStream
  then
    insertLogical(new TransientReminder((MemberCase)null, $clinician,
      "ClinicianReminder", "Clinician completed no evaluation in last 7 days."));
end

query "ClinicianReminderQuery"
  $clinicianReminder : TransientReminder(reminderTypeCd == "ClinicianReminder")
end
```

リスト 7 に記載する `TransientReminder` ファクトは JPA エンティティーではなく、通常の POJO であることに注意してください。

リスト 7. TransientReminder

```
public class TransientReminder implements Comparable, Serializable
{
  private MemberCase memberCase;
  private Clinician clinician;
  private String reminderTypeCd;
  private String description;

  public String toString()
  {
    return ReflectionToStringBuilder.toString(this);
  }

  public boolean equals(Object pObject)
  {
    return EqualsBuilder.reflectionEquals(this, pObject);
  }

  public int compareTo(Object pObject)
  {
    return CompareToBuilder.reflectionCompare(this, pObject);
  }

  public int hashCode()
  {
    return HashCodeBuilder.reflectionHashCode(this);
  }
}
```

ファクトとイベントの違い

イベントとは、時間メタデータ (`@timestamp`、`@duration`、`@expires` など) で修飾されたファクトのことを指します。ファクトとイベントとの最も大きな違いは、イベントは Drools のコンテキストでは不変であることです。イベントが変更される可能性があるとしても、(「イベント・データの拡充」と表現される) その変更がルールの実行結果に影響を与えることはありません。このことから、`CaseSupervision` の `EntityListener` では (リスト 2 を参照)、`@PostPersist` エンティティー・ライフサイクル・フェーズだけを監視しています。

Drools のスライディング・ウィンドウ・プロトコルのサポートにより、イベントはとりわけ時間推論に大きな威力を発揮します。スライディング・ウィンドウとは、関心対象のイベントのス

コープを、常時変化している時間ウィンドウに属するイベントであるかのように設定する方法です。最もよく使われているスライディング・ウィンドウの実装には、時刻ベースのウィンドウと期間ベースのウィンドウの 2 つがあります。

リスト 6 に記載したサンプル・ルールで、`over window:time(30d)` が意味しているのは、過去 30 日以内に作成された `CaseSupervision` イベントがルール・エンジンによって評価されるということです。作成されてから 30 日が経過すると、不変のイベントが再びこのスライディング・ウィンドウに入ることはありません。Drools はそのイベントを自動的にワーキング・メモリーから削除し、それに応じてルールが再評価されます。イベントは不変であることから、Drools は自動的にイベントのライフサイクルを管理します。したがって、イベントのメモリー効率はファクトよりも優れています (ただし、Drools-Spring 構成でイベント処理モードを `STREAM` に設定する必要があります。そうでないと、スライディング・ウィンドウのような時相演算子は機能しません)。

宣言型を扱う

リスト 6 でもう 1 つ注目すべき点は、過去 30 日内に行われた症例診断のみを評価することから、(イベント型ではない) `MemberCase` ファクトも時間制約に対して評価されることです。症例診断が行われてからの経過日数は、今日時点で 29 日だとしても、翌日には 30 日になります。これは、毎日一日が始まる時点で、Case Supervision ルールを再評価しなければならないことを意味します。あいにく、Drools では「today (今日)」というスライディング変数を提供していません。その次善策として、`Today` という名前のイベント型を追加しました。これは、Drools の宣言型です。つまり、Java コードの中ではなく、ルール言語の中で宣言され、Drools の構成概念となるデータです。

この特殊なイベント型は、明示的な属性を一切宣言しません。宣言するのは、`Today` イベントがワーキング・メモリーにアサートされるときに自動的に取り込まれる暗黙的 `@timestamp` メタデータです。そしてもう 1 つのメタデータ `@expires(24h)` で、アサーションから 24 時間後に `Today` イベントが有効でなくなることを指定します。

一日が始まる時点で `Today` をリセットするために、Set Today ルールをベースとした `timer` も追加しました。このルールは、毎日一日が始まる時点でアクティブにされて起動され、有効期限が切れた `Today` イベントに代わる新しい Today イベントを挿入します。すると、この新しい `Today` イベントが Case Supervision ルールの再評価をトリガーします。ここで、ルールの条件でファクトが変更されない限り、タイマー自体がルールの再評価をトリガーできないことにも注意してください。タイマーは関数も、インライン `eval` も再評価しません。Drools は、これらの構成概念となるデータの戻りを時間定数として取り、その値をキャッシュに入れるためです。

ファクトを使用すべき場合とイベントを使用すべき場合

ファクトとイベントの違いを理解すれば、どちらの型を使用するべきかを容易に判断することができます。

- データがある時点または期間におけるシステム状態の不変のスナップショットを表わし、データが時間に依存して短時間で有効期限切れになるか、データの量が急速に増加し続けることが予想されるシナリオには、イベントを使用してください。
- ビジネス・ドメインにとってデータの重要性が高く、データが継続的に変更され、ルールを常に再評価しなければならないシナリオには、ファクトを使用してください。

Drools のクエリー

次のステップでは、ルールの実行結果を抽出します。それには、ワーキング・メモリー内のファクトに対してクエリーを実行します (あるいは、ルール構文の RHS で `global` のメソッドを呼び出して、ルール・エンジンが結果をアプリケーションに渡すようにするという方法もあります)。このサンプル・アプリケーションでは、ファクトのアサーションとルールの起動は瞬時に行われ、その間には遅延がないため、リスト 6 のクエリーは確実にリアルタイムのレポートを返します。必然的に `TransientReminder` ファクトはアサートされるため、ルール・エンジンはファクトの条件が満たされなくなると、自動的にファクトを削除します。

今朝、ある特定の症例でルール・エンジンによって督促状が生成されたとします。そこで、リスト 3 に記載した Java コードのクエリー `CaseReminderQuery` を実行しました。これにより、督促状が返されて、システムのすべての臨床医に表示されました。その日の午後、臨床医がその特定の症例の診断を完了して新しい症例診断レコードを生成すると、このイベントにより、督促状ファクトの条件が満たされなくなります。すると、Drools はそのファクトを自動的に削除します。督促状ファクトが削除されたことを確認するには、症例の診断が完了した直後に同じクエリーを実行します。論理アサーションにより、推論結果は最新に維持され、ルール・エンジンがイベントと同じようにメモリー効率の高いモードで実行し続けます。

論理ファクト・カウンター

論理的にアサートされたファクトには、カウンターが伴うことに注意してください。このカウンターは、同等ファクトがアサートされるたびにインクリメントされます。同等ファクトを繰り返しアサートしたルールのうち、あるルールが維持されなくなると、論理ファクトのカウンターがデクリメントされます。カウンターがゼロになると、ファクトは自動的に削除されます。

ライブ・クエリーは不可欠ではないものの、嬉しいおまけです。ライブ・クエリーはオープン状態を維持し、クエリー結果のビューを作成して、その特定のビューの内容として変更イベントをパブリッシュします。つまり、ライブ・クエリーを一度実行すれば、結果のビューは、ルール・エンジンによってパブリッシュされる継続的変更で自動的に更新されるということです。

これまでの説明で、Drools、JPA、および Spring の多少の基礎知識があれば、継続的なリアルタイムのデータ・プロファイリング・アプリケーションを実行するのは難しくはないことを明らかにしました。ここからは、症例管理ソリューションを強化する高度なプログラミング・ステップを紹介して、記事を締めくくることにします。

高度な Drools プログラミング

リレーションを管理する

`FactHandle` に伴う興味深い制約は、このトークンを関連付けられるのは現行のファクトだけで、ファクトのネストされたリレーションには関連付けることができないことです。`getKsession().update(factHandle, memberCase)` を呼び出すと、`MemberCase` の `id` に対して行われた変更 (主キーは不変であるため、これが変更されることはありません) や `startDtm`、`endDtm` などに対する変更がその `FactHandle` を介して Drools に通知されます。ただし、同じメソッドを呼び出しても、member および `caseSupervisions` プロパティーに対する変更は通知されません。

同様に、JPA の `EntityListener` には、1 対多の関係および多対多の関係に対する変更が通知されません。外部キーは、リレーションがあるテーブルまたはリンク・テーブルに置かれるためです。

これらの関係を更新されたファクトとして関連付けるには、ネストされたリレーションごとの `FactHandle` を取得する再帰ロジックを作成することができますが、それよりも有効なソリューションは、リンク・テーブルを含むすべてのエンティティーに、ルールの条件に関与する `EntityListener` を配置することです。このソリューションを使用した `Member` と `CaseSupervision` では、変更が各エンティティーに固有の `EntityListener` および `FactHandle` で処理されます (リスト 2 およびリスト 3 を参照)。

ルール評価中にエンティティーを遅延ロードする

ナレッジ・ベースのパーティション (つまり、並列処理) を指定しない限り、ルールは `ksession.insert()`、`ksession.update()`、または `ksession.retract()` が呼び出されたスレッドと同じスレッドで評価されます。リスト 2 とリスト 3 でのファクトのアサーションはどちらもトランザクションのコンテキストで行われ、トランザクションにスコープを設定した JPA パーシスタンス・コンテキスト (Hibernate セッション) を使用することができます。このことから、遅延ロードされたエンティティーのリレーションでルール・エンジンを評価することができます。ナレッジ・ベースのパーティションを有効にする場合には、JPA LazyInitializationException が発生しないように、エンティティーのリレーションが即時ロードされるように構成する必要があります。

トランザクションを有効にする

Drools はワーキング・メモリー内のデータの履歴スナップショットを保持しないため、デフォルトでは、Drools はトランザクションをサポートしません。このことは、サンプル・アプリケーションの `EntityListener` には問題になります。なぜなら、ライフサイクル・コールバック・メソッドは、データベースがフラッシュされた後、トランザクションがコミットされる前に呼び出されるためです。トランザクションがロールバックされた場合を考えてみてください。その場合、JPAパーシスタンス・コンテキストのエンティティーは分離され、データベース・テーブル内の行と矛盾することになり、ワーキング・メモリー内のファクトも矛盾します。そうなると、ルール・エンジンの推論結果の信頼性が損なわれます。

トランザクションを有効にすると、この症例管理システムは安定したものになります。トランザクションが有効にされていれば、ワーキング・メモリー内のデータとアプリケーション・データベース内のデータが常に同期され、したがってルール推論の結果も常に正確なものになるためです。Drools では、JPA 実装と JTA 実装を配備し、`drools-jpa-persistence` パッケージをクラス・パスに含めることによって、ステートフル・ナレッジ・セッションを作成するように `JPAKnowledgeService` (「参考文献」を参照) を構成することができます。プロセス・インスタンス、変数、およびファクト・オブジェクトを含めたステートフル・ナレッジ・セッション全体が、`ksessionId` を主キーとした SessionInfo テーブル内の行に、バイナリー列としてマッピングされます。

サンプル・アプリケーションのコードにアノテーションまたは XML を使用してトランザクション境界を指定すると、アプリケーションが開始したトランザクションがルール・エンジンにまで及びます。トランザクションがロールバックされた場合は常に、ステートフル・ナレッジ・セッ

ションはデータベースに保存されている前の状態にリストアされます。したがって、アプリケーション・データベースと Drools データベースとの間の整合性と統合が保持されるというわけです。メモリー内のシングルトン・ステートフル・ナレッジ・セッションは、複数の JTA トランザクションから同時にアクセスされた場合、REPEATABLE READ のように振る舞う必要があります。そうでないと、単一の SessionInfo エンティティー・インスタンスに異なる複数のトランザクションによる状態変更が混在し、トランザクション区分が壊れてしまう恐れがあります。この記事を執筆している時点で、drools-jpa-persistence パッケージのトランザクション・マネージャーによって REPEATABLE READ が実装されるかどうかは確認されていないことに注意してください。

クラスタリング

サンプル・アプリケーションをクラスタリング環境で実行するとしたら、これまで説明した手法はたちまち失敗することになります。クラスタリング環境では、組み込みルール・エンジンの各インスタンスが同じノードで発生したエンティティー・イベントを受け取ることになるため、さまざまなノード上のワーキング・メモリーの同期が外れる結果となるためです。この問題は、汎用リモート Drools サーバー (「参考文献」を参照) を使用することで是正することができます。汎用リモート Drools サーバーを使用すると、異なる複数のノード上のエンティティー・リスナーが、リッスンした全イベントを REST/SOAP Web サービス通信によって中央の Drools サーバーにパブリッシュするようになるため、アプリケーションが Drools サーバーからの推論結果をサブスクライブできるようになるはずです。注意する点として、Drools サーバーでの SOAP の Apache CXF 実装は、現在 `ws-transaction` をサポートしていません。この実際のユース・ケースで概説した必須のトランザクション要件を考えると、この `ws-transaction` のサポートがすぐに実現されることが望まれます。

まとめ

この記事では、Spring と JPA での POJO プログラミングの知識を、Drools 5に用意された新機能と組み合わせる機会を提供しました。記事で具体的に説明したのは、`EntityListener`、グローバル Drools セッション、`fireUtilHalt()` メソッドなどを賢く利用して、POJO ベースの継続的リアルタイム・データ・プロファイリング・アプリケーションを開発する方法です。そのなかで、サブジェクトをファクトまたはイベントとして処理するなどの Drools の中核的な概念、論理アサーションの作成方法、さらにはトランザクションの管理や Droools 実装のクラスタリング環境への拡張といった高度な Droools プログラミングに関するトピックとその使用法についても説明しました。Drools 5 についてさらに詳しく学ぶには、アプリケーション・ソース・コードを参照してください。

ダウンロード

内容	ファイル名	サイズ
Sample code for this article	j-drools5-src.zip	5KB

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、サーバー・サイドの最先端技術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、他のシステム、サービス、アプリケーションにシームレスに統合できる医療関係の製品、ソリューション、サービス、交換プラットフォームを提供する医療テクノロジー企業、eHealthObjects で製品開発部長を務めています。Java.net、JavaWorld.com、IBM developerWorks に、JSF、Spring Security、Hibernate Search、Spring Web Flow、Servlet 3.0 仕様などの話題を取り上げた記事を書いている他、Packt Publishing で出版している『Spring Web Flow 2 Web Development』、『Grails 1.1 Web Application Development』、および『Application Development for IBM WebSphere Process Server 7 and Enterprise Service Bus 7』の校閲も手掛けました。