



JBoss Enterprise SOA Platform 5

JBPM リファレンスガイド

JBoss 開発者向け

エディション 5.2.0

JBoss Enterprise SOA Platform 5 JBPM リファレンスガイド

JBoss 開発者向け
エディション 5.2.0

編集者

David Le Sage
Red Hat エンジニアリングコンテンツサービス

Darrin Mison
Red Hat エンジニアリングコンテンツサービス

法律上の通知

Copyright © 2011 Red Hat, Inc.

This document is licensed by Red Hat under the [Creative Commons Attribution-ShareAlike 3.0 Unported License](#). If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

概要

JBoss Enterprise SOA Platform での JBPM および JPDL の使用方法については本書をご参照ください。

目次

第1章 はじめに	6
1.1. 概要	6
1.2. JPD L スイート	7
1.3. JPD L グラフィカルプロセスデザイナー	8
1.4. JBPM WEB コンソールアプリケーション	8
1.5. JBPM のコアコンポーネント	8
1.6. アイデンティティコンポーネント	9
1.7. JBPM ジョブエグゼキューター	9
1.8. まとめ	9
第2章 チュートリアル	10
2.1. 「HELLO WORLD」 サンプル	10
2.2. データベースサンプル	11
2.3. コンテキストサンプル: プロセス変数	15
2.4. タスク割り当てのサンプル	16
2.5. カスタムアクションのサンプル	18
第3章 設定	21
3.1. ファクトリのカスタマイズ	24
3.2. 設定プロパティ	25
3.3. その他の設定ファイル	25
3.4. 楽観的同時実行制御のロギング	26
3.5. オブジェクトファクトリ	26
第4章 永続性	30
4.1. 永続性アプリケーションプログラミングインターフェース	30
4.1.1. 設定フレームワークとの関係	30
4.1.2. JbpmContext の便利なメソッド	30
4.2. 永続サービスの設定	33
4.2.1. データベース互換性	33
4.2.1.1. JDBC 接続の分離レベル	34
4.2.1.2. データベースの変更	34
4.2.1.3. データベーススキーマ	34
4.2.1.3.1. プログラムでデータベーススキーマ操作	34
4.2.1.4. Hibernate クラスとの統合	35
第5章 JAVA EE アプリケーションサーバー機能	36
5.1. エンタープライズ BEAN	36
5.2. JBPM エンタープライズ設定	38
5.3. HIBERNATE エンタープライズ設定	40
5.4. クライアントのコンポーネント	41
5.5. まとめ	44
第6章 プロセスのモデリング	45
6.1. 便利な定義	45
6.2. プロセスグラフ	45
6.3. ノード	47
6.3.1. ノードの責任	47
6.3.2. ノードタイプ: タスクノード	48
6.3.3. ノードタイプ: 状態	48
6.3.4. ノードタイプ - 決定	48
6.3.5. ノードタイプ - フォーク	48
6.3.6. ノードタイプ - ジョイン	49

6.3.7. ノードタイプ - ノード	49
6.4. 遷移(TRANSITION)	49
6.5. アクション	49
6.5.1. アクション参照	51
6.5.2. イベント	51
6.5.3. イベントを渡す	51
6.5.4. スクリプト	51
6.5.5. カスタムイベント	52
6.6. SUPER-STATES	52
6.6.1. Super-state 遷移	52
6.6.2. Super-state イベント	53
6.6.3. 階層名	53
6.7. 例外ハンドリング	53
6.8. プロセス構成	54
6.9. カスタムノードの動作	54
6.10. グラフ実行	55
6.11. トランザクション境界	57
第7章 コンテキスト	59
7.1. プロセス変数へのアクセス	59
7.2. 変数のライフ	60
7.3. 変数の永続性	60
7.4. 変数スコープ	60
7.4.1. 変数オーバーローディング	60
7.4.2. 変数オーバーライディング	60
7.4.3. タスクインスタンス変数スコープ	61
7.5. 一時変数	61
第8章 タスク管理	62
8.1. タスク	62
8.2. タスクインスタンス	62
8.2.1. タスクインスタンスのライフサイクル	62
8.2.2. タスクインスタンスとグラフ実行	63
8.3. 割り当て	64
8.3.1. Assignment インターフェース	64
8.3.2. 割り当てデータモデル	65
8.3.3. パーソナルタスクリスト	65
8.3.4. グループタスクリスト	66
8.4. タスクインスタンス変数	66
8.5. タスクコントローラ	67
8.6. スイムレーン	69
8.7. 開始タスクのスイムレーン	69
8.8. タスクイベント	69
8.9. タスクタイマー	70
8.10. タスクインスタンスのカスタマイズ	70
8.11. アイデンティティコンポーネント	70
8.11.1. アイデンティティモデル	71
8.11.2. 割り当て式	71
8.11.2.1. 最初の条件 (First term)	72
8.11.2.2. 次の条件 (Next term)	72
8.11.3. アイデンティティコンポーネントの削除	72
第9章 スケジューラー	74
9.1. タイマー	74

9.2. スケジューラーデプロイメント	74
第10章 非同期の続行	76
10.1. コンセプト	76
10.2. 例	76
10.3. ジョブエグゼキューター	78
10.4. JBPM 組み込み非同期メッセージング	80
第11章 ビジネスカレンダー	82
11.1. DUE DATE (期限)	82
11.1.1. Duration (期間)	82
11.1.2. 基準日	82
11.1.3. 期限の例	82
11.2. カレンダー設定	83
11.3. 使用例	83
第12章 電子メールサポート	85
12.1. JPDL でのメール	85
12.1.1. メールアクション	85
12.1.2. メールノード	86
12.1.3. タスクが割り当てられた電子メール	86
12.1.4. タスクリマインダ電子メール	86
12.2. メールにおける表現	87
12.3. メール受信者の指定	87
12.3.1. 複数の受信者	87
12.3.2. BCC アドレスへ電子メールを送信	87
12.3.3. アドレス解決	88
12.4. 電子メールテンプレート	88
12.5. メールサーバーの設定	89
12.6. 送信者アドレスの設定	89
12.7. 電子メールサポートのカスタマイズ	89
第13章 ロギング	91
13.1. ログの作成	91
13.2. ログの設定	92
13.3. ログの読み出し	92
第14章 JBPM プロセス定義言語	94
14.1. プロセスアーカイブ	94
14.1.1. プロセスアーカイブのデプロイ	94
14.1.2. プロセスバージョンニング	95
14.1.3. デプロイ済プロセス定義の変更	95
14.1.4. プロセスインスタンスの移行	96
14.2. 委譲	96
14.2.1. JBPMクラスローダー	96
14.2.2. プロセスクラスローダ	97
14.2.3. 委譲設定	97
14.2.3.1. config-type フィールド	97
14.2.3.2. config-type Bean	98
14.2.3.3. config-type コンストラクター	98
14.2.3.4. config-type configuration-property	99
14.3. 式	99
14.4. JPDL XML スキーマ	99
14.4.1. バリデーション	99

14.4.2. プロセス定義	100
14.4.3. node	100
14.4.4. 一般的なノード要素	101
14.4.5. start-state	101
14.4.6. end-state	102
14.4.7. state	102
14.4.8. task-node	102
14.4.9. process-state	103
14.4.10. super-state	103
14.4.11. fork	104
14.4.12. join	104
14.4.13. decision	104
14.4.14. event	105
14.4.15. transition	105
14.4.16. action	105
14.4.17. script	106
14.4.18. expression	107
14.4.19. variable	107
14.4.20. handler	108
14.4.21. timer	108
14.4.22. create-timer	109
14.4.23. cancel-timer	110
14.4.24. task	110
14.4.25. スイムレーン	111
14.4.26. 割り当て	111
14.4.27. Controller	112
14.4.28. sub-process	113
14.4.29. condition	113
14.4.30. exception-handler	114
第15章 WORKFLOW のテスト駆動開発 (TDD)	115
15.1. WORKFLOW のテスト駆動開発の紹介	115
15.2. XMLソース	116
15.2.1. プロセスアーカイブの構文解析	116
15.2.2. XML ファイルの構文解析	116
15.2.3. XML String の構文解析	116
付録A GNU LESSER GENERAL PUBLIC LICENSE 2.1	118
付録B 改訂履歴	128

第1章 はじめに

本書は開発者やルール記述者向けのガイドで、企業環境で JBPM や JPDL を使う方法を説明しています。本書はソフトウェアの使用方法を取り上げるだけではなく、ソフトウェアが動作する仕組みについても深く掘り下げて説明しています。



注記

本ガイドには多くの用語が使用されています。主な用語の意味は「[便利な定義](#)」に記載されています。

JBoss Business Process Manager(JBPM) はプロセス言語のフレームワークです。

JBoss Business Process Manager(JBPM) はこのフレームワーク上に構築されたプロセス言語です。これは非常にわかりやすい言語で、ビジネスプロセスをグラフィカルに表現できます。この言語でタスク、(非同期通信)の待機状態、タイマー、自動化アクションを表現できます。これらの操作をバインドするために、この言語には制御フローメカニズムが備えられています。

JPDL が持つ依存関係はほとんどないため、簡単にインストールできます。インストールするには、J2EE クラスター化アプリケーションサーバー上にデプロイします。



注記

これは、極度のスループットが重要な要件となる環境で特に便利です。



注記

JPDL は、どのようなデータベースでも設定可能であり、どのようなアプリケーションサーバ上でもデプロイ可能です。

1.1. 概要

本項を読んで、JBPM の動作方法の概要について学びましょう。

コアワークフローとビジネスプロセス管理機能は、単純な Java ライブラリとしてパッケージ化されています。このライブラリには、JPDL データベースプロセスを管理し実行するサービスが含まれています。

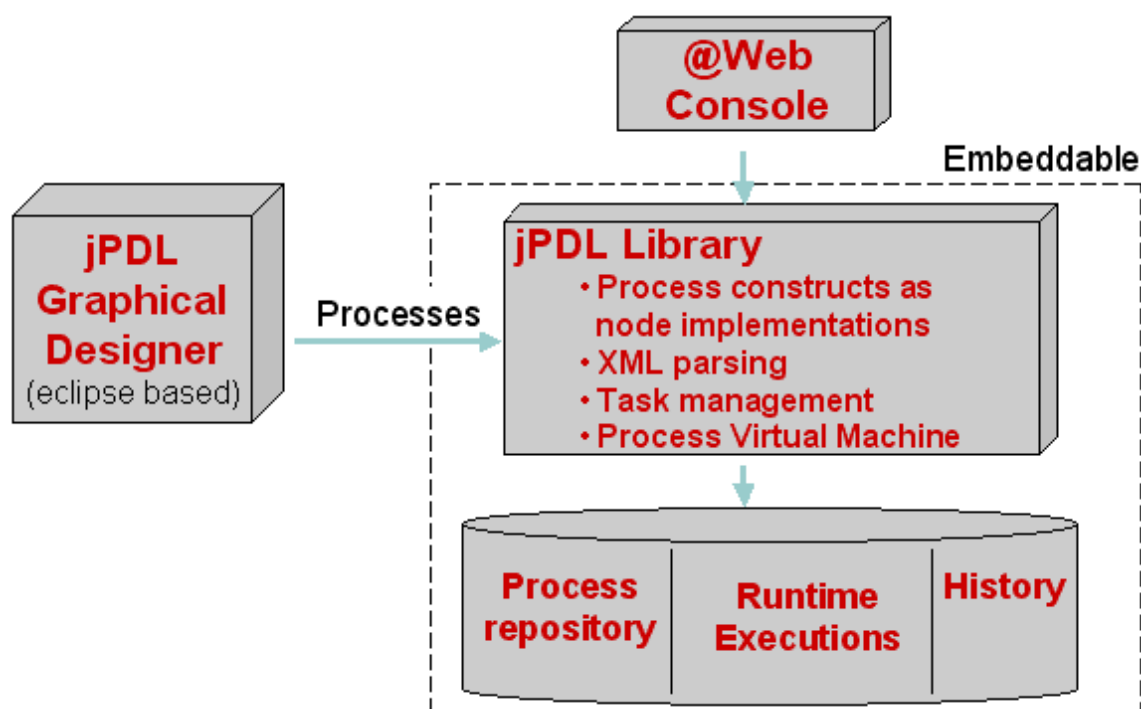


図1.1 JPDL コンポーネント概要

1.2. JPDL スイート

本スイートには、全ての JBPM コンポーネントと次のサブディレクトリが含まれています。

- config
- database
- deploy
- designer
- examples
- lib
- src

JBoss Application Server は次のコンポーネントで構成されます。

JBoss JBPM Web コンソール

Web アーカイブとしてパッケージ化されます。プロセスパーティシパントと JBPM 管理者の両者がこのコンソールを使用できます。

ジョブエグゼキュータ

Console Web Application の一部です。 *servlet* によって開始され、タイマーと非同期メッセージを監視し実行するスレッドプールを生成します。

JBPM テーブル

デフォルトの **Hypersonic** データベースに格納されます (既にプロセスが含まれています)。

サンプルプロセス

1つのサンプルプロセスが既に JBPM データベースへデプロイされています。

アイデンティティコンポーネント

アイデンティティコンポーネントライブラリは **Web コンソール アプリケーション** の一部です。JBPM_ID_ の接頭辞を持つデータベース内にあるテーブルを所有します。

1.3. JPDL グラフィカルプロセスデザイナー

JPDL には **Graphical Process Designer Tool** が含まれています。このツールを使用してビジネスプロセスを設計します (**JBoss Developer Studio** 製品に含まれるプラグインです)。また、ビジネスプロセスモデルから実装までをスムーズに移行しやすくし、ビジネスアナリストや技術開発者の両者に便利なものとなっています。

1.4. JBPM WEB コンソールアプリケーション

Web コンソールアプリケーション は3つの役割があります。まず、集約ユーザーインターフェースとして機能し、プロセス実行が生成したランタイムタスクとのやり取りを可能にします。

2つ目はランタイムインスタンスの検査、操作を可能にする管理・監視コンソールとしての役割を果たします。

このソフトウェアの3番目の役割は、ビジネスアクティビティモニターとしての機能です。この役割で、プロセス実行の統計を表示します。この情報は、ボトルネックを検出し取り除くことができるためパフォーマンスの最適化を目指すマネージャーにとって便利です。

1.5. JBPM のコアコンポーネント

JBoss Business Process Manager には2つのコアコンポーネントがあります。プロセス定義を管理する「プレーンJava」(J2SE) ライブラリと、プロセスインスタンスを実行するランタイム環境です。

JBPM 自体が Java ライブラリであるため、**Web** や **Swing** アプリケーション、**Enterprise Java Bean**、**Web サービス** などすべての Java 環境で使用することができます。

JBPM ライブラリをステートレスセッション **Enterprise Java Bean** としてパッケージ化し、公開することもできます。クラスター化デプロイメントを作成する必要があり、極めて高いスループットに対応する拡張性を提供する必要がある場合に行います。



注記

ステートレスセッション **Enterprise Java Bean** は **J2EE 1.3** 仕様に準拠しているため、すべてのアプリケーションサーバーにデプロイ可能です。



重要

jbpm-jpdl.jar ファイルの一部は、**Hibernate** や **Dom4J** などのサードパーティーライブラリに依存しています。

Hibernate は JBPM に 永続性 機能を提供します。また、従来の *O/R* マッピングを提供するだけでなく、**Hibernate** は別のデータベースが使用する様々な **SQL** (構造化照会言語) ダイアレクトの違いを解決します。この機能により JBPM の移植性が非常に高くなります。

JBoss Business Process Manager の API は、Web アプリケーションや Enterprise Java Bean、Web サービスコンポーネント、メッセージ駆動型 Bean などすべてのカスタム Java コードよりアクセス可能です。

1.6. アイデンティティコンポーネント

JBPM はユーザーデータ (およびその他の組織データ) を格納するすべての企業ディレクトリと統合可能です。(組織情報のコンポーネントがないプロジェクトでは、**アイデンティティコンポーネント**を使用します。このコンポーネントには、従来のサーブレット、Enterprise Java Bean、ポートレットによって使用されるモデルよりも「リッチ」なモデルが含まれます)。



注記

詳細は「[アイデンティティコンポーネント](#)」を参照してください。

1.7. JBPM ジョブエグゼキューター

ジョブエグゼキューターコンポーネントを使い標準の Java 環境でジョブを監視し実行します。また、ジョブを使いタイマーや非同期メッセージを作成します。



注記

エンタープライズ環境では、Java Message Service や Enterprise Java Bean **TimerService** が上記の目的で利用される場合もありますが、「標準」環境においてはジョブエグゼキューターが最適です。

ジョブエグゼキューターコンポーネントは、コアの **jbpdm-jpdl** ライブラリにパッケージされています。次に示す 2 つのシナリオのうちいずれかでのみ、このコンポーネントはデプロイ可能です。

- **JbpmThreadsServlet** がジョブエグゼキューターを開始するために設定されている場合
- 別の Java Virtual Machine が起動されている (そして、そこからジョブエグゼキュータースレッドを実行できる) 場合

1.8. まとめ

本章をお読みいただき、JBPM や JBPM を構成するコンポーネントの概要を理解していただけたと思います。

第2章 チュートリアル

次のチュートリアルを読んで JPDL の基本的なプロセス構築について学びましょう。このチュートリアルでは、API よりランタイム実行を管理する方法も示しています。

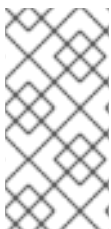
チュートリアルにあるサンプルの詳細説明はそれぞれ、**src/java.examples** サブディレクトリにある JBPM ダウンロードパッケージにあります。



注記

Red Hat はこの時点でプロジェクトの作成を推奨します。自由に各サンプルを実験し、各サンプルを変更しバリエーションを作成することができます。

まず JBPM をダウンロードしインストールしてください。



注記

JBPM には例示されている XML をオーサリングするためのグラフィックデザイナーツールが含まれています。本書の「ダウンロードの概要」のセクションにグラフィックデザイナーのダウンロードに関する説明があります。グラフィックデザイナーを使う必要なく、このチュートリアルを完了することも可能です。

2.1. 「HELLO WORLD」 サンプル

プロセス定義は有向グラフで、ノードと遷移で構成されています。**Hello World** プロセス定義には 3 つのノードがあります (**Designer Tool** を使わずに簡単なプロセスから始めると、まとまりや組み合わせが理解できるはずです)。

以下の図は、**Hello World** プロセスのグラフ表示です。

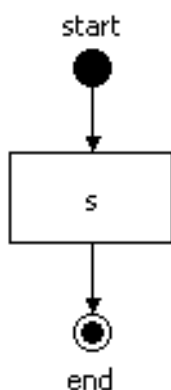


図2.1 Hello World プロセスグラフ

```
public void testHelloWorldProcess() {  
    // This method shows a process definition and one execution  
    // of the process definition. The process definition has  
    // 3 nodes: an unnamed start-state, a state 's' and an  
    // end-state named 'end'.  
    // The next line parses a piece of xml text into a  
    // ProcessDefinition. A ProcessDefinition is the formal  
    // description of a process represented as a java object.  
    ProcessDefinition processDefinition = ProcessDefinition.parseXmlString(  

```

```

    "<process-definition>" +
    "  <start-state>" +
    "    <transition to='s' />" +
    "  </start-state>" +
    "  <state name='s'>" +
    "    <transition to='end' />" +
    "  </state>" +
    "  <end-state name='end' />" +
    "</process-definition>"
  );

  // The next line creates one execution of the process definition.
  // After construction, the process execution has one main path
  // of execution (=the root token) that is positioned in the
  // start-state.
  ProcessInstance processInstance =
    new ProcessInstance(processDefinition);

  // After construction, the process execution has one main path
  // of execution (=the root token).
  Token token = processInstance.getRootToken();

  // Also after construction, the main path of execution is positioned
  // in the start-state of the process definition.
  assertSame(processDefinition.getStartState(), token.getNode());

  // Let's start the process execution, leaving the start-state
  // over its default transition.
  token.signal();
  // The signal method will block until the process execution
  // enters a wait state.

  // The process execution will have entered the first wait state
  // in state 's'. So the main path of execution is now
  // positioned in state 's'
  assertSame(processDefinition.getNode("s"), token.getNode());

  // Let's send another signal. This will resume execution by
  // leaving the state 's' over its default transition.
  token.signal();
  // Now the signal method returned because the process instance
  // has arrived in the end-state.

  assertSame(processDefinition.getNode("end"), token.getNode());
}

```

2.2. データベースサンプル

JBPM の基本機能の1つが、**wait state** である間にデータベースプロセスの実行を永続化する機能です。次のサンプルはこの機能を例示しています。

ここでは異なるユーザーコードに個別の **methods** を作成しています。例えば、ある **Web** アプリケーションのコードがプロセスを開始し、データベースで実行を永続化します。その後、メッセージ駆動型 **bean** がそのプロセスインスタンスをロードし、実行を再開します。



注記

jbpm 永続化機能の詳細は、[4章永続性](#)を参照してください。

```
public class HelloWorldDbTest extends TestCase {

    static JbpmConfiguration jbpmConfiguration = null;

    static {
        // An example configuration file such as this can be found in
        // 'src/config.files'. Typically the configuration information
        // is in the resource file 'jbpm.cfg.xml', but here we pass in
        // the configuration information as an XML string.

        // First we create a JbpmConfiguration statically. One
        // JbpmConfiguration can be used for all threads in the system,
        // that is why we can safely make it static.

        jbpmConfiguration = JbpmConfiguration.parseXmlString(
            "<jbpm-configuration>" +

            // A jbpm-context mechanism separates the jbpm core
            // engine from the services that jbpm uses from
            // the environment.

            "<jbpm-context>" +
            "<service name='persistence' " +
            " factory='org.jbpm.persistence.db.DbPersistenceServiceFactory' />" +
+
            "</jbpm-context>" +

            // Also all the resource files that are used by jbpm are
            // referenced from the jbpm.cfg.xml

            "<string name='resource.hibernate.cfg.xml' " +
            " value='hibernate.cfg.xml' />" +
            "<string name='resource.business.calendar' " +
            " value='org/jbpm/calendar/jbpm.business.calendar.properties' />" +
            "<string name='resource.default.modules' " +
            " value='org/jbpm/graph/def/jbpm.default.modules.properties' />" +
            "<string name='resource.converter' " +
            " value='org/jbpm/db/hibernate/jbpm.converter.properties' />" +
            "<string name='resource.action.types' " +
            " value='org/jbpm/graph/action/action.types.xml' />" +
            "<string name='resource.node.types' " +
            " value='org/jbpm/graph/node/node.types.xml' />" +
            "<string name='resource.varmapping' " +
            " value='org/jbpm/context/exe/jbpm.varmapping.xml' />" +
            "</jbpm-configuration>"
        );
    }

    public void setUp() {
        jbpmConfiguration.createSchema();
    }
}
```



```

public void tearDown() {
    jbpmConfiguration.dropSchema();
}

public void testSimplePersistence() {
    // Between the 3 method calls below, all data is passed via the
    // database. Here, in this unit test, these 3 methods are executed
    // right after each other because we want to test a complete process
    // scenario. But in reality, these methods represent different
    // requests to a server.

    // Since we start with a clean, empty in-memory database, we have to
    // deploy the process first. In reality, this is done once by the
    // process developer.
    deployProcessDefinition();

    // Suppose we want to start a process instance (=process execution)
    // when a user submits a form in a web application...
    processInstanceIsCreatedWhenUserSubmitsWebappForm();

    // Then, later, upon the arrival of an asynchronous message the
    // execution must continue.
    theProcessInstanceContinuesWhenAnAsyncMessageIsReceived();
}

public void deployProcessDefinition() {
    // This test shows a process definition and one execution
    // of the process definition. The process definition has
    // 3 nodes: an unnamed start-state, a state 's' and an
    // end-state named 'end'.
    ProcessDefinition processDefinition =
        ProcessDefinition.parseXmlString(
            "<process-definition name='hello world'>" +
            "  <start-state name='start'>" +
            "    <transition to='s' />" +
            "  </start-state>" +
            "  <state name='s'>" +
            "    <transition to='end' />" +
            "  </state>" +
            "  <end-state name='end' />" +
            "</process-definition>"
        );

    //Lookup the pojo persistence context-builder that is configured above
    JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
    try {
        // Deploy the process definition in the database
        jbpmContext.deployProcessDefinition(processDefinition);
    } finally {
        // Tear down the pojo persistence context.
        // This includes flush the SQL for inserting the process definition
        // to the database.
        jbpmContext.close();
    }
}

```

```

}

public void processInstanceIsCreatedWhenUserSubmitsWebappForm() {
    // The code in this method could be inside a struts-action
    // or a JSF managed bean.

    //Lookup the pojo persistence context-builder that is configured above
    JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
    try {

        GraphSession graphSession = jbpmContext.getGraphSession();

        ProcessDefinition processDefinition =
            graphSession.findLatestProcessDefinition("hello world");

        //With the processDefinition that we retrieved from the database, we
        //can create an execution of the process definition just like in the
        //hello world example (which was without persistence).
        ProcessInstance processInstance =
            new ProcessInstance(processDefinition);

        Token token = processInstance.getRootToken();
        assertEquals("start", token.getNode().getName());
        // Let's start the process execution
        token.signal();
        // Now the process is in the state 's'.
        assertEquals("s", token.getNode().getName());

        // Now the processInstance is saved in the database. So the
        // current state of the execution of the process is stored in the
        // database.
        jbpmContext.save(processInstance);
        // The method below will get the process instance back out
        // of the database and resume execution by providing another
        // external signal.

    } finally {
        // Tear down the pojo persistence context.
        jbpmContext.close();
    }
}

public void theProcessInstanceContinuesWhenAnAsyncMessageIsReceived() {
    //The code in this method could be the content of a message driven bean.

    // Lookup the pojo persistence context-builder that is configured
    // above
    JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
    try {

        GraphSession graphSession = jbpmContext.getGraphSession();
        // First, we need to get the process instance back out of the
        // database. There are several options to know what process
        // instance we are dealing with here. The easiest in this simple
        // test case is just to look for the full list of process instances.
        // That should give us only one result. So let's look up the

```

```

// process definition.

ProcessDefinition processDefinition =
    graphSession.findLatestProcessDefinition("hello world");

//Now search for all process instances of this process definition.
List processInstances =
    graphSession.findProcessInstances(processDefinition.getId());

// Because we know that in the context of this unit test, there is
// only one execution. In real life, the processInstanceId can be
// extracted from the content of the message that arrived or from
// the user making a choice.
ProcessInstance processInstance =
    (ProcessInstance) processInstances.get(0);

// Now we can continue the execution. Note that the processInstance
// delegates signals to the main path of execution (=the root
token).
processInstance.signal();

// After this signal, we know the process execution should have
// arrived in the end-state.
assertTrue(processInstance.hasEnded());

// Now we can update the state of the execution in the database
jbpmContext.save(processInstance);

} finally {
    // Tear down the pojo persistence context.
    jbpmContext.close();
}
}
}

```

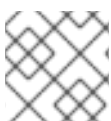
2.3. コンテキストサンプル: プロセス変数

プロセス実行中、コンテキスト情報はプロセス変数に保持されます。プロセス変数は変数名を値にマッピングする点が **java.util.Map** クラス (こちらは Java オブジェクト) と類似します (プロセス変数はプロセスインスタンスの一部として永続化されます)。



注記

次のサンプルをシンプルにするため、変数と動作する必要がある API のみを表します (永続性機能は省いています)。



注記

変数の詳細については、[7章 コンテキスト](#) を参照してください。

```

// This example also starts from the hello world process.
// This time even without modification.
ProcessDefinition processDefinition = ProcessDefinition.parseXmlString(

```

```

"<process-definition>" +
"  <start-state>" +
"    <transition to='s' />" +
"  </start-state>" +
"  <state name='s'>" +
"    <transition to='end' />" +
"  </state>" +
"  <end-state name='end' />" +
"</process-definition>"
);

ProcessInstance processInstance =
  new ProcessInstance(processDefinition);

// Fetch the context instance from the process instance
// for working with the process variables.
ContextInstance contextInstance =
  processInstance.getContextInstance();

// Before the process has left the start-state,
// we are going to set some process variables in the
// context of the process instance.
contextInstance.setVariable("amount", new Integer(500));
contextInstance.setVariable("reason", "i met my deadline");

// From now on, these variables are associated with the
// process instance. The process variables are now accessible
// by user code via the API shown here, but also in the actions
// and node implementations. The process variables are also
// stored into the database as a part of the process instance.

processInstance.signal();

// The variables are accessible via the contextInstance.

assertEquals(new Integer(500),
  contextInstance.getVariable("amount"));
assertEquals("i met my deadline",
  contextInstance.getVariable("reason"));

```

2.4. タスク割り当てのサンプル

次のサンプルはユーザーにタスクを割り当てる方法を表しています。JBPM ワークフローエンジンと組織的モデルは分離されているため、常に表現言語は、アクターの算出に利用するには数が制限されます。表現言語を使用する代わりに **AssignmentHandler** の実装を指定し、タスクに対するアクターの算出が含まれるよう使用します。

```

public void testTaskAssignment() {
  // The process shown below is based on the hello world process.
  // The state node is replaced by a task-node. The task-node
  // is a node in JPDL that represents a wait state and generates
  // task(s) to be completed before the process can continue to
  // execute.
  ProcessDefinition processDefinition = ProcessDefinition.parseXmlString(
    "<process-definition name='the baby process'>" +

```

```

" <start-state>" +
"   <transition name='baby cries' to='t' />" +
" </start-state>" +
" <task-node name='t'>" +
"   <task name='change nappy'>" +
"     <assignment" +
"       class='org.jbpm.tutorial.taskmgmt.NappyAssignmentHandler' />"
+
"   </task>" +
"   <transition to='end' />" +
" </task-node>" +
" <end-state name='end' />" +
"</process-definition>"
);

// Create an execution of the process definition.
ProcessInstance processInstance =
    new ProcessInstance(processDefinition);
Token token = processInstance.getRootToken();

// Let's start the process execution, leaving the start-state
// over its default transition.
token.signal();
// The signal method will block until the process execution
// enters a wait state. In this case, that is the task-node.
assertSame(processDefinition.getNode("t"), token.getNode());

// When execution arrived in the task-node, a task 'change nappy'
// was created and the NappyAssignmentHandler was called to determine
// to whom the task should be assigned. The NappyAssignmentHandler
// returned 'papa'.

// In a real environment, the tasks would be fetched from the
// database with the methods in the org.jbpm.db.TaskMgmtSession.
// Since we don't want to include the persistence complexity in
// this example, we just take the first task-instance of this
// process instance (we know there is only one in this test
// scenario).
TaskInstance taskInstance = (TaskInstance)
    processInstance
        .getTaskMgmtInstance()
        .getTaskInstances()
        .iterator().next();

// Now, we check if the taskInstance was actually assigned to 'papa'.
assertEquals("papa", taskInstance.getActorId() );

// Now we suppose that 'papa' has done his duties and mark the task
// as done.
taskInstance.end();
// Since this was the last (only) task to do, the completion of this
// task triggered the continuation of the process instance execution.

assertSame(processDefinition.getNode("end"), token.getNode());
}

```

2.5. カスタムアクションのサンプル

アクションはカスタム Java コードを JBPM プロセスにバインドするためのメカニズムです。アクションはアクション独自のノードに関連付けることができます (プロセスのグラフ表示に関係ある場合)。

また、アクションをイベント上に「置く」ことも可能です (遷移する時やノードへ入退場する時)。アクションがイベント上に置かれると、グラフィカル表現の一部として扱われなくなります (ランタイムプロセス実行中にイベントが発生した時はアクションは実行されます)。

最初に次のサンプルで使用されるアクションハンドラー実装、**MyActionHandler** を見てください。**MyActionHandler** はブール型変数 **isExecuted** を **true** に設定するだけです。この変数は静的であるため、アクションハンドラー (およびアクション自体) からアクセスし、値を検証することができます。



注記

アクションの詳細については、「[アクション](#)」を参照してください。

```
// MyActionHandler represents a class that could execute
// some user code during the execution of a jBPM process.
public class MyActionHandler implements ActionHandler {

    // Before each test (in the setUp), the isExecuted member
    // will be set to false.
    public static boolean isExecuted = false;

    // The action will set the isExecuted to true so the
    // unit test will be able to show when the action
    // is being executed.
    public void execute(ExecutionContext executionContext) {
        isExecuted = true;
    }
}
```



重要

各テストの前に、静的フィールド **MyActionHandler.isExecuted** を **false** に設定します。

```
// Each test will start with setting the static isExecuted
// member of MyActionHandler to false.
public void setUp() {
    MyActionHandler.isExecuted = false;
}
```

最初のサンプルは、遷移上のアクションになります。

```
public void testTransitionAction() {
    // The next process is a variant of the hello world process.
    // We have added an action on the transition from state 's'
    // to the end-state. The purpose of this test is to show
    // how easy it is to integrate Java code in a jBPM process.
    ProcessDefinition processDefinition =
```

```

ProcessDefinition.parseXmlString(
    "<process-definition>" +
    "  <start-state>" +
    "    <transition to='s' />" +
    "  </start-state>" +
    "  <state name='s'>" +
    "    <transition to='end'>" +
    "      <action class='org.jbpm.tutorial.action.MyActionHandler' />"
+
    "    </transition>" +
    "  </state>" +
    "  <end-state name='end' />" +
    "</process-definition>"
);

// Let's start a new execution for the process definition.
ProcessInstance processInstance =
    new ProcessInstance(processDefinition);

// The next signal will cause the execution to leave the start
// state and enter the state 's'
processInstance.signal();

// Here we show that MyActionHandler was not yet executed.
assertFalse(MyActionHandler.isExecuted);
// ... and that the main path of execution is positioned in
// the state 's'
assertSame(processDefinition.getNode("s"),
    processInstance.getRootToken().getNode());

// The next signal will trigger the execution of the root
// token. The token will take the transition with the
// action and the action will be executed during the
// call to the signal method.
processInstance.signal();

// Here we can see that MyActionHandler was executed during
// the call to the signal method.
assertTrue(MyActionHandler.isExecuted);
}

```

次のサンプルでは、**enter-node** および **leave-node** イベント両方に置かれた同じアクションを示しています。



注記

ノードは複数のイベントタイプを持っていることに注意してください。1つのイベントしか持たない遷移とは対照的です。アクションをノードに置く場合は、必ずイベント要素に入れるようにしてください。

```

ProcessDefinition processDefinition = ProcessDefinition.parseXmlString(
    "<process-definition>" +
    "  <start-state>" +
    "    <transition to='s' />" +
    "  </start-state>" +

```

```
"    <state name='s'>" +
"        <event type='node-enter'>" +
"            <action class='org.jbpm.tutorial.action.MyActionHandler' />" +
"        </event>" +
"        <event type='node-leave'>" +
"            <action class='org.jbpm.tutorial.action.MyActionHandler' />" +
"        </event>" +
"        <transition to='end' />" +
"    </state>" +
"    <end-state name='end' />" +
"</process-definition>"
);
```

```
ProcessInstance processInstance =
    new ProcessInstance(processDefinition);

assertFalse(MyActionHandler.isExecuted);
// The next signal will cause the execution to leave the start
// state and enter the state 's'. So the state 's' is entered
// and hence the action is executed.
processInstance.signal();
assertTrue(MyActionHandler.isExecuted);

// Let's reset the MyActionHandler.isExecuted
MyActionHandler.isExecuted = false;

// The next signal will trigger execution to leave the
// state 's'. So the action will be executed again.
processInstance.signal();
// Voila.
assertTrue(MyActionHandler.isExecuted);
```


第3章 設定

JBPM の設定方法について学習するには本章を参照してください。

JBPM を設定する最も簡単な方法は、クラスパスのルートに **jbpm.cfg.xml** 設定ファイルを指定することです。このファイルがリソースとして存在しない場合は、JBPM ライブラリに含まれるデフォルトの最小設定が使用されます (**org/jbpm/default/jbpm.cfg.xml**)。

反対に JBPM 設定ファイルが提供された場合、設定された値がデフォルト値として使用されます。したがって、デフォルトの設定ファイルと異なる部分だけ指定する必要があります。

JBPM の設定は、Java クラス **org.jbpm.JbpmConfiguration** によって表されています。**singleton** インスタンスメソッド (**JbpmConfiguration.getInstance()**) を利用して取得します。



注記

他のソースから設定をロードするには、**JbpmConfiguration.parseXxxx** メソッドを使います。

```
static JbpmConfiguration jbpmConfiguration =
    JbpmConfiguration.parseResource("my.jbpm.cfg.xml");
```

JbpmConfiguration は「スレッドセーフ」であるため、静的メンバーに保存できます。

各スレッドは **JbpmConfiguration** を **JbpmContext** オブジェクトのファクトリとして利用できます。**JbpmContext** は通常1つのトランザクションを表します。次のようにコンテキストブロック内でサービスを使用できるようにします。

```
JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
try {
    // This is what we call a context block.
    // Here you can perform workflow operations

} finally {
    jbpmContext.close();
}
```

JbpmContext はサービスと構成設定の両方を JBPM が使用できるようにします。

サービスは、**jbpm.cfg.xml** ファイルの値によって設定されます。前述の環境にあるサービスを使用して JBPM がすべての Java 環境で稼働できるようにします。

次は **JbpmContext** のデフォルト設定になります。

```
<jbpm-configuration>

<jbpm-context>
  <service name='persistence'
    factory='org.jbpm.persistence.db.DbPersistenceServiceFactory' />
  <service name='message'
    factory='org.jbpm.msg.db.DbMessageServiceFactory' />
  <service name='scheduler'
```

```

        factory='org.jbpm.scheduler.db.DbSchedulerServiceFactory' />
    <service name='logging'
        factory='org.jbpm.logging.db.DbLoggingServiceFactory' />
    <service name='authentication'
        factory=
'org.jbpm.security.authentication.DefaultAuthenticationServiceFactory' />
</jbpm-context>

<!-- configuration resource files pointing to default
configuration files in jbpm-{version}.jar -->
<string name='resource.hibernate.cfg.xml' value='hibernate.cfg.xml' />

<!-- <string name='resource.hibernate.properties'
value='hibernate.properties' /> -->
<string name='resource.business.calendar'
value='org/jbpm/calendar/jbpm.business.calendar.properties' />
<string name='resource.default.modules'
value='org/jbpm/graph/def/jbpm.default.modules.properties' />
<string name='resource.converter'
value='org/jbpm/db/hibernate/jbpm.converter.properties' />
<string name='resource.action.types'
value='org/jbpm/graph/action/action.types.xml' />
<string name='resource.node.types'
value='org/jbpm/graph/node/node.types.xml' />
<string name='resource.parsers'
value='org/jbpm/jpdl/par/jbpm.parsers.xml' />
<string name='resource.varmapping'
value='org/jbpm/context/exe/jbpm.varmapping.xml' />
<string name='resource.mail.templates'
value='jbpm.mail.templates.xml' />

<int name='jbpm.byte.block.size' value="1024" singleton="true" />
<bean name='jbpm.task.instance.factory'
class='org.jbpm.taskmgmt.impl.DefaultTaskInstanceFactoryImpl'
singleton='true' />

<bean name='jbpm.variable.resolver'
class='org.jbpm.jpdl.el.impl.JbpmVariableResolver'
singleton='true' />

<string name='jbpm.mail.smtp.host' value='localhost' />

<bean name='jbpm.mail.address.resolver'
class='org.jbpm.identity.mail.IdentityAddressResolver'
singleton='true' />
<string name='jbpm.mail.from.address' value='jbpm@noreply' />

<bean name='jbpm.job.executor'
class='org.jbpm.job.executor.JobExecutor'>
    <field name='jbpmConfiguration'><ref bean='jbpmConfiguration' />
    </field>
    <field name='name'><string value='JbpmJobExecutor' /></field>
    <field name='nbrOfThreads'><int value='1' /></field>
    <field name='idleInterval'><int value='60000' /></field>
    <field name='retryInterval'><int value='4000' /></field>
<!-- 1 hour -->

```

```

<field name='maxIdleInterval'><int value='3600000' /></field>
<field name='historyMaxSize'><int value='20' /></field>
<!-- 10 minutes -->
<field name='maxLockTime'><int value='600000' /></field>
<!-- 1 minute -->
<field name='lockMonitorInterval'><int value='60000' /></field>
<!-- 5 seconds -->
<field name='lockBufferTime'><int value='5000' /></field>
</bean>
</jbpm-configuration>

```

上記ファイルには 3 つのパーツが含まれます。

1. **JbpmContext** 設定するサービス実装のセット (特定のサービス実装について取り上げている章に可能な設定オプションの詳細が記載されています)。
2. 設定リソースへの参照にリンクするすべてのマッピング。設定ファイルの1つをカスタマイズしたい場合はこれらのマッピングを更新します。カスタマイズする時は、最初に必ずデフォルトの設定ファイル (**jbpm-3.x.jar**) をクラスパス上の別の場所にバックアップします。その後、JBPM が使用するカスタマイズされたバージョンを示してこのファイルへの参照を更新します。
3. JBPM が使用するその他の設定 (特定の内容について取り上げている章に説明があります)。

デフォルト設定は、最小限に依存関係を持つ単純な web アプリケーション環境向けに最適化されています。永続サービスは、他すべてのサービスで使用する JDBC 接続を取得します。そのため、すべてのワークフロー操作は、JDBC 接続上の単一のトランザクションに置かれ集約されます (そのためトランザクションマネージャが必要ありません)。

JbpmContext には、ほとんどの共通プロセス操作に対する便利なメソッドが含まれています。このサンプルでは、これらのメソッドの使用方法について説明しています。

```

public void deployProcessDefinition(ProcessDefinition processDefinition)
public List getTaskList()
public List getTaskList(String actorId)
public List getGroupTaskList(List actorIds)
public TaskInstance loadTaskInstance(long taskId)
public TaskInstance loadTaskInstanceForUpdate(long taskId)
public Token loadToken(long tokenId)
public Token loadTokenForUpdate(long tokenId)
public ProcessInstance loadProcessInstance(long processInstanceId)
public ProcessInstance loadProcessInstanceForUpdate(long
processInstanceId)
public ProcessInstance newProcessInstance(String processDefinitionName)
public void save(ProcessInstance processInstance)
public void save(Token token)
public void save(TaskInstance taskInstance)
public void setRollbackOnly()

```



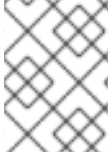
注記

XxxForUpdate メソッドは、ロードされたオブジェクトを「自動保存」登録されるように設計されているため、**save** メソッドを手動で呼び出す必要はありません。

複数の **jbpm-context** を指定することは可能ですが、その際に各 **jbpm-context** に固有の **name** 属性

を付けなければなりません (`JbpmConfiguration.createContext(String name);` を使用して名前付けされたコンテキストを読み出します)。

service要素は、独自の名前と関連する サービスファクトリを指定します。 `JbpmContext.getServices().getService(String name)` による要求があった場合のみ、このサービスが作成されます。



注記

factories を属性でなく **要素** として指定することもできます。一部の設定情報をファクトリオブジェクトに注入する際に要素の指定が必要となります。



注記

オブジェクトの作成や書き込み、XML の解析を行うコンポーネントは **object factory** と呼ばれます。

3.1. ファクトリのカスタマイズ

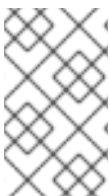


警告

ファクトリのカスタマイズ時の一般的な間違いは、短表記と長表記を混在させてしまうことです。これは行わないようにしましょう (短い表記法の例は、デフォルトの設定ファイルにあります)。

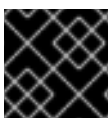
Hibernate は `StateObjectStateException` をログに記録し、これに対するスタックトレースを生成します。スタックトレースを削除するには、`org.hibernate.event.def.AbstractFlushingEventListener` を **FATAL** に設定します。

```
<service name='persistence'
    factory='org.jbpm.persistence.db.DbPersistenceServiceFactory' />
```



注記

あるいは、**log4j** を使っている場合は以下の行を設定ファイルに追加します:
`log4j.logger.org.hibernate.event.def.AbstractFlushingEventListener=FATAL`



重要

サービス上に特定のプロパティを示す必要がある場合、長表記のみが使用可能です。

```
<service name="persistence">
  <factory>
    <bean class="org.jbpm.persistence.db.DbPersistenceServiceFactory">
      <field name="dataSourceIndiName">
```

```

        <string value="java:/myDataSource"/>
    </field>
    <field name="isCurrentSessionEnabled"><true /></field>
    <field name="isTransactionEnabled"><false /></field>
</bean>
</factory>
</service>

```

3.2. 設定プロパティ

jbpm.byte.block.size

添付ファイルとバイナリ変数は、固定サイズのバイナリオブジェクトのリストとしてデータベースに保存されます (これは異なるデータベース間の移植性を向上するのが目的で、JBPM の組み込みも簡易化されます)。このパラメーターは、固定長チャンクのサイズを制御します。

jbpm.task.instance.factory

タスクインスタンスが作成される方法をカスタマイズするには、このプロパティで完全修飾クラス名を指定します。(通常、この操作は **TaskInstance bean** をカスタマイズし、新しいプロパティを追加する時に必要になります)。指定したクラス名が

org.jbpm.taskmgmt.TaskInstanceFactory インターフェースを実装するようにしなければなりません (詳細は、「[タスクインスタンスのカスタマイズ](#)」を参照してください)。

jbpm.variable.resolver

これを使い、JBPM が JSF に類似した表現にある最初の用語を検索する方法をカスタマイズします。

3.3. その他の設定ファイル

JBPM にはカスタマイズ可能な設定ファイルが複数あります。

hibernate.cfg.xml

このファイルには、**Hibernate** マッピングリソースファイルへの参照と設定の詳細が含まれます。

別のファイルを指定するには、**jbpm.properties** の **jbpm.hibernate.cfg.xml** プロパティを設定します (デフォルトの **Hibernate** 設定ファイルは **src/config.files/hibernate.cfg.xml** サブディレクトリにあります)。

org/jbpm/db/hibernate.queries.hbm.xml

このファイルには、JBPM セッション (**org.jbpm.db.*Session**) で使用される **Hibernate** クエリが含まれています。

org/jbpm/graph/node/node.types.xml

このファイルは、XML ノードエレメントを **Node** 実装クラスにマップするために使用されます。

org/jbpm/graph/action/action.types.xml

このファイルは、XML アクションエレメントを **Action** 実装クラスへマップするために使用されます。

org/jbpm/calendar/jbpm.business.calendar.properties

「営業時間」と「自由時間」の定義が含まれます。

org/jbpm/context/exe/jbpm.varmapping.xml

JBPM データベースに保存するため、プロセス変数の値 (Java オブジェクト) をどのように変数インスタンスに変換するか指定します。

org/jbpm/db/hibernate/jbpm.converter.properties

id-to-classname マッピングを指定します。id はデータベースに保存されます。

org.jbpm.db.hibernate.ConverterEnumType は、識別子を **singleton** オブジェクトにマップするため使用されます。

org/jbpm/graph/def/jbpm.default.modules.properties

デフォルトで新しい **ProcessDefinition** に追加されるモジュールを指定します。

org/jbpm/jpdl/par/jbpm.parsers.xml

プロセスアーカイブ解析のフェーズを指定します。

3.4. 楽観的同時実行制御のロギング

JBPM をクラスター構成で実行している場合、JBPM は **楽観的ロック** を使用しデータベースに基づいて同期されます。そのため、各操作は 1 つのトランザクションで実行されます。最終的に衝突が検出された場合は、このトランザクションがロールバックされ、再試行で処理される必要があります。これにより、**org.hibernate.StateObjectStateException** 例外が発生する場合があります。

この例外が発生した場合、**Hibernate** は以下のような簡単なメッセージで例外をログに記録します。

```
optimistic locking
      failed
```

Hibernate はスタックトレースで **StateObjectStateException** をログに記録することもできます。スタックトレースを削除するに

は、**org.hibernate.event.def.AbstractFlushingEventListener** クラスを **FATAL** に設定します。次の設定を使用して **log4j** でこの操作を行います。

```
log4j.logger.org.hibernate.event.def.AbstractFlushingEventListener=FATAL
```

JBPM スタックトレースをログに記録するには、以下の行を **jbpm.cfg.xml** に追加します。

```
<boolean name="jbpm.hide.stale.object.exceptions" value="false" />
```

3.5. オブジェクトファクトリ

オブジェクトファクトリは、**Bean** のような XML 設定ファイルに含まれる仕様に従ったオブジェクトを構築できます。このファイルは、オブジェクトグラフを構成するために、どのようにオブジェクトが作成、設定、配線されるかを決定します。

また、オブジェクトファクトリを使用して設定や他の **Bean** を主要 **Bean** に挿入できます。

下記の例のように最も簡単な構成では、オブジェクトファクトリは設定より基本タイプと **Java Bean** の両方を作成することができます。


```
<beans>
  <bean name="task" class="org.jbpm.taskmgmt.exe.TaskInstance"/>
  <string name="greeting">hello world</string>
  <int name="answer">42</int>
  <boolean name="javaisold">true</boolean>
  <float name="percentage">10.2</float>
  <double name="salary">1000000000.32</double>
  <char name="java">j</char>
  <null name="dusttodust" />
</beans>
```

```
ObjectFactory of = ObjectFactory.parseXmlFromAbove();
assertEquals(TaskInstance.class, of.getNewObject("task").getClass());
assertEquals("hello world", of.getNewObject("greeting"));
assertEquals(new Integer(42), of.getNewObject("answer"));
assertEquals(Boolean.TRUE, of.getNewObject("javaisold"));
assertEquals(new Float(10.2), of.getNewObject("percentage"));
assertEquals(new Double(1000000000.32), of.getNewObject("salary"));
assertEquals(new Character('j'), of.getNewObject("java"));
assertNull(of.getNewObject("dusttodust"));]]>
```

このコードはリストの設定方法を表しています。

```
<beans>
  <list name="numbers">
    <string>one</string>
    <string>two</string>
    <string>three</string>
  </list>
</beans>
```

これはマップの設定方法を表しています。

```
<beans>
  <map name="numbers">
    <entry>
      <key><int>1</int></key>
      <value><string>one</string></value>
    </entry>
    <entry>
      <key><int>2</int></key>
      <value><string>two</string></value>
    </entry>
    <entry>
      <key><int>3</int></key>
      <value><string>three</string></value>
    </entry>
  </map>
</beans>
```

直接フィールド注入とプロパティ **setter** メソッドを使用して **Bean** を設定します。このコードはその方法を示しています。

```
<beans>
```

```

    <bean name="task" class="org.jbpm.taskmgmt.exe.TaskInstance" >
      <field name="name"><string>do dishes</string></field>
      <property name="actorId"><string>theotherguy</string></property>
    </bean>
  </beans>

```

Bean は参照可能です。参照されるオブジェクトは Bean である必要はなく、文字列や整数、他のオブジェクトにすることが可能です。

```

<beans>
  <bean name="a" class="org.jbpm.A" />
  <ref name="b" bean="a" />
</beans>

```

Bean を構築したいコンストラクターを利用することができます。

```

<beans>
  <bean name="task" class="org.jbpm.taskmgmt.exe.TaskInstance" >
    <constructor>
      <parameter class="java.lang.String">
        <string>do dishes</string>
      </parameter>
      <parameter class="java.lang.String">
        <string>theotherguy</string>
      </parameter>
    </constructor>
  </bean>
</beans>

```

クラス上で **static factory** メソッドを使用して Bean を構築することができます。

```

<beans>
  <bean name="taskFactory"
    class="org.jbpm.UnexistingTaskInstanceFactory"
    singleton="true"/>

  <bean name="task" class="org.jbpm.taskmgmt.exe.TaskInstance" >
    <constructor factory="taskFactory" method="createTask" >
      <parameter class="java.lang.String">
        <string>do dishes</string>
      </parameter>
      <parameter class="java.lang.String">
        <string>theotherguy</string>
      </parameter>
    </constructor>
  </bean>
</beans>

```

クラス上で **static factory** メソッドを使用して Bean を構築することができます。

```

<beans>
  <bean name="task" class="org.jbpm.taskmgmt.exe.TaskInstance" >
    <constructor
      factory-class="org.jbpm.UnexistingTaskInstanceFactory"
      method="createTask" >

```

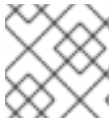


```

        <parameter class="java.lang.String">
            <string>do dishes</string>
        </parameter>
        <parameter class="java.lang.String">
            <string>theotherguy</string>
        </parameter>
    </constructor>
</bean>
</beans>

```

singleton="true" を使用して各名前付きのオブジェクトを **singleton** としてマークします。これにより、各要求に対して **object factory** が常に同じオブジェクトを返すようにします。



注記

Singletons を異なるオブジェクトファクトリで共有することはできません。



注記

singleton 機能により、**getObject** というメソッドと **getNewObject** というメソッドを区別することができます。新しいオブジェクトグラフが構築される前に **object factory** のオブジェクトキャッシュを消去するため、通常は **getNewObject** を使用します。

オブジェクトグラフの構築中、*非シングルトン*オブジェクトは **object factory** のキャッシュに保存されます。これにより、1つのオブジェクトへの参照を共有できるようになります。**singleton object cache** は **plain object cache** とは異なります。**singleton** キャッシュは消去されることがありませんが、**plain object cache** メソッドが開始される度に **plain object cache** は消去されます。

本章の内容を理解できれば、多くの可能な JBPM の設定方法について十分な知識を得たことになります。

第4章 永続性

本章は、JBoss Business Process Manager の永続性機能について詳細に説明していますのでご参照ください。

多くの場合、JBPM は複数のトランザクションにわたるプロセスを実行するために使用されます。永続性機能の主な目的は、待機状態が発生した時にプロセス実行を保存することです。プロセス実行を状態マシンとして考えると分かりやすいはずです。1つのトランザクション中で、プロセス実行状態のマシンを別の状態に移動させることが目的です。

プロセス定義は、XML、Java オブジェクト、JBPM データベースレコードの3つの異なる形式で表現することができます (ランタイムデータとログ情報は Java オブジェクトか JBPM データベースレコードで表現することができます)。



注記

プロセス定義とプロセスアーカイブの XML 表記についての詳細は、[14章JBPM プロセス定義言語](#)を参照してください。



注記

データベースにプロセスアーカイブをデプロイする方法の詳細については、「[プロセスアーカイブのデプロイ](#)」を参照してください。

4.1. 永続性アプリケーションプログラミングインターフェース

4.1.1. 設定フレームワークとの関係

永続性 API は設定フレームワークと統合されます ([3章設定](#) 参照)。統合するには、**JbpmContext** にある **convenience persistence** メソッドの一部を公開し、JBPM context block が永続 API 操作を呼び出せるようにします。

```
JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
try {
    // Invoke persistence operations here
} finally {
    jbpmContext.close();
}
```

4.1.2. JbpmContext の便利なメソッド

最も頻繁に実行される永続操作は次の3つになります。

1. プロセスのデプロイ
2. 新規プロセスの実行開始
3. プロセス実行の継続

プロセスデプロイメントは通常 **Graphical Process Designer** か **deployprocess** の **ant** タスクによって直接実行されます。Java から直接実行するには、次のコードを使用します。

```
JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
```

```
try {
    ProcessDefinition processDefinition = ...;
    jbpmContext.deployProcessDefinition(processDefinition);
} finally {
    jbpmContext.close();
}
```

インスタンスとなるプロセス定義を指定して新しいプロセス実行を作成します。最も一般的な方法は、プロセス名を参照する方法です。JBPM はデータベースにあるプロセスの最新バージョンを検索します。デモコードは次の通りです。

```
JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
try {
    String processName = ...;
    ProcessInstance processInstance =
        jbpmContext.newProcessInstance(processName);
} finally {
    jbpmContext.close();
}
```

プロセス実行を継続するには、プロセスインスタンス、トークン、**taskInstance** のいずれかをデータベースより取得し、JBPM POJO (*Plain Old Java Object*) 上でメソッドを呼び出します。その後、**processInstance** への更新をデータベースに保存します。

```
JbpmContext jbpmContext = jbpmConfiguration.createJbpmContext();
try {
    long processInstanceId = ...;
    ProcessInstance processInstance =
        jbpmContext.loadProcessInstance(processInstanceId);
    processInstance.signal();
    jbpmContext.save(processInstance);
} finally {
    jbpmContext.close();
}
```



注記

JbpmContext クラスで **ForUpdate** メソッドが使用される場合、**jbpContext.save** メソッドを手動で呼び出す必要はありません。これは、**jbpContext** クラスが閉じられると保存プロセスが自動的に実行されるからです。例えば、**taskInstance** が終了したことを JBPM に伝えたいとします。これにより実行が継続されるため、**taskInstance** に関連する **processInstance** を保存しなければなりません。**loadTaskInstanceForUpdate** メソッドを使用する方法が最も便利です。

```
JbpmContext jbpContext =
jbpConfiguration.createJbpContext();
try {
    long taskInstanceId = ...;
    TaskInstance taskInstance =
        jbpContext.loadTaskInstanceForUpdate(taskInstanceId);
    taskInstance.end();
}
finally {
    jbpContext.close();
}
```



重要

JbpConfiguration は **ServiceFactories** のセットを維持します。**ServiceFactories** は **jbp.config.xml** ファイルを使い設定され、必要な時にインスタンス化されます。

DbPersistenceServiceFactory は最初に必要となった時のみインスタンス化されます。その後、**ServiceFactory** は **JbpConfiguration** で維持されます。

DbPersistenceServiceFactory は **Hibernate** の **ServiceFactory** を管理しますが、最初に要求があった時のみインスタンス化されます。

注記

`jbpmConfiguration.createJbpmContext()` クラスが呼び出されると、**JbpmContext** のみが作成されます。この時点では永続性に関連する初期化はこれ以上発生しません。**JbpmContext** は、最初に要求があった時にインスタンス化される **DbPersistenceService** を管理します。**DbPersistenceService** クラスは、最初に必要な時のみインスタンス化される **Hibernate** を管理します (永続性を必要とする最初の操作が呼び出された時のみ **Hibernate** セッションが開きます)。

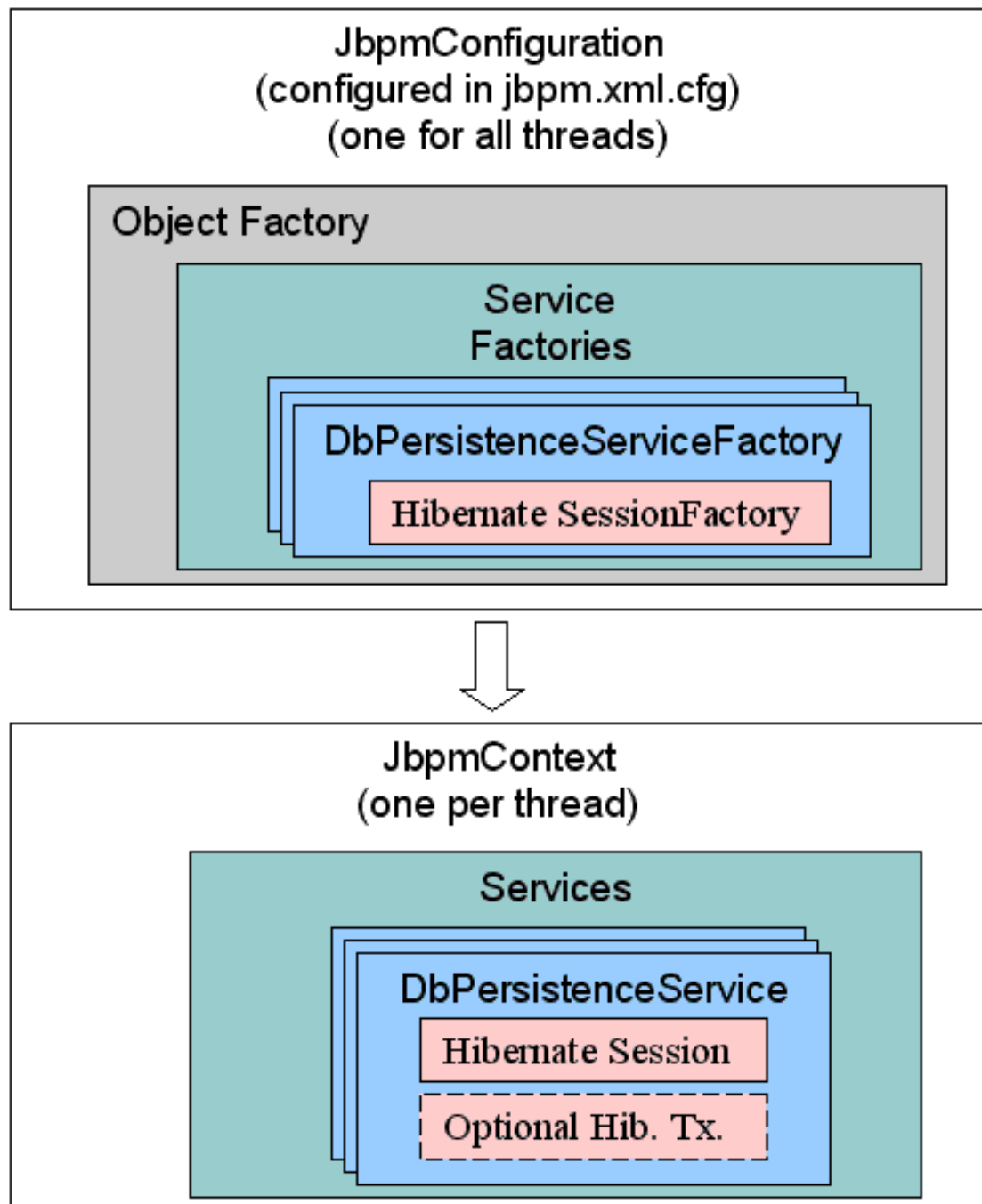


図4.1 永続性に関連するクラス

4.2. 永続サービスの設定

4.2.1. データベース互換性

Hibernate と互換のあるデータベースであれば JBPM を実行できます。

サンプル設定ファイル **src/config.files** は、開発やテストに最適な **H2** インメモリデータベースの使用を指定します (**H2** はメモリ内のすべてのデータを保持し、ディスク上には何も保存しません)。

4.2.1.1. JDBC 接続の分離レベル

JDBC 接続のデータベース分離レベルは、最低でも **READ_COMMITTED** に設定してください。



警告

READ_UNCOMMITTED (分離レベル 0、**H2** によってサポートされる唯一の分離レベル) に設定すると、**job executor** で競合状態が発生することがあります。また、複数のトークンが同期化されると競合状態が発生することもあります。

4.2.1.2. データベースの変更

JBPM が異なるデータベースを使用するよう再設定するには、次の手順に従います。

- JDBC ドライバライブラリアーカイブをクラスパスに置きます。
- JBPM が使用する **Hibernate** 設定を更新します。
- 新しいデータベースにスキーマを作成します。

4.2.1.3. データベーススキーマ

希望するデータベースを使用しやすくするため、**jbpm.db** サブプロジェクトにはドライバー、説明、スクリプトが含まれています。詳細は、**jbpm.db** プロジェクトの **root** にある **readme.html** を参照してください。



注記

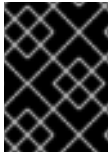
JBPM はすべてのデータベースに対して **DDL** を生成できますが、スキーマが常に最適化されているとは限りません。データベース管理者に依頼してカラムタイプとインデックスを最適化できるよう生成された **DDL** の確認をしてもらうとよいでしょう。

次の **Hibernate** 設定オプションが開発環境で使用されているかもしれません。

hibernate.hbm2ddl.auto を **create-drop** に設定すると、データベースがアプリケーションで初めて使用されるとスキーマが自動的に作成されます。アプリケーションが終了すると、スキーマはドロップされます。

4.2.1.3.1. プログラムでデータベーススキーマ操作

JBPM は、[org.jbpm.JbpmConfiguration](#) の **createSchema** と **dropSchema** メソッドを使いデータベーススキーマを作成、削除するための API を提供します。設定済みのデータベースユーザーが持つ権限以外に、これらのメソッドの呼び出しへの制限はありません。



重要

これらのメソッド呼び出しに関する制限は設定したデータベースユーザーの権限により左右される点に注意してください。



注記

前述したAPIは、[org.jbpm.db.JbpmSchema](#) クラスが提供する幅広い機能の一部となっています。

4.2.1.4. Hibernate クラスとの統合

Hibernate と **JBPM** 永続クラスを統合すると、2つの主な利点があります。1つ目の利点は、接続とトランザクションの管理が簡単になることです。これは、1つの **Hibernate** セッションファクトリとして統合することで1つの **Hibernate** セッションと1つの **JDBC** 接続のみになるからです。そのため、**JBPM** の更新がドメインモデルの更新と同じトランザクションになります。これにより、トランザクションマネージャーが不要になります。

2つ目の利点は、余分な労力を使わず **Hibernate** 永続オブジェクトをプロセス変数にドロップできるようになることです。

統合するには、中心となる **hibernate.cfg.xml** ファイルを1つ作成します。デフォルトの **JBPM hibernate.cfg.xml** を使用し、独自の **Hibernate** マッピングファイルへの参照を追加してカスタマイズするのが最も簡単な方法でしょう。

第5章 JAVA EE アプリケーションサーバー機能

本章を読んで、JPBM を使った Java EE インフラストラクチャーの活用方法について理解できるようにしましょう。

5.1. エンタープライズ BEAN

CommandServiceBeanは、別の JBPM コンテキスト内で**execute** メソッドを呼び出すことによって JBoss Business Process Manager を実行するステートレスセッション *Bean* です。

以下の表に、利用できるカスタマイズ可能なリソースと環境エントリについて要約されています。

表5.1 コマンドサービス Bean 環境

名前	種類	説明
JbpmCfgResource	環境エントリ	JBPM 設定を読み取るクラスパスリソース。オプションで、デフォルトは jbpmm.cfg.xml です。
ejb/TimerEntityBean	EJB 参照	スケジューラサービスを実装するローカルエンティティ Bean へのリンク。タイマーを含むプロセスに必要です。
jdbc/JbpmDataSource	リソースマネージャ参照	JDBC 接続を JBPM 永続サービスに提供するデータソースの論理名です。Hibernate 設定ファイルの hibernate.connection.datasource プロパティに一致する必要があります。
jms/JbpmConnectionFactory	リソースマネージャ参照	JMS 接続を JBPM メッセージサービスに提供するファクトリの論理名です。非同期続行を含むプロセスに必要です。
jms/JobQueue	メッセージ送信先参照	JBPM メッセージサービスはこのキューにジョブメッセージを送信します。ジョブリスナー Bean がメッセージを受信するキューと同じキューになるように、 message-destination-link が一般的な論理送信先 JobQueue を示します。
jms/CommandQueue	メッセージ送信先参照	コマンドリスナー Bean はこのキューからメッセージを受信します。コマンドメッセージを送信するキューと同じキューになるように、 message-destination-link element は一般的な論理送信先 CommandQueue を参照します。

CommandListenerBean はコマンドメッセージを **CommandQueue** でリスンするメッセージ駆動型 Bean です。この Bean はコマンド実行を **CommandServiceBean** に委譲します。

メッセージ本体は **org.jbpm.Command** インタフェースを実装できる Java オブジェクトである必要があります (メッセージプロパティがある場合は無視されます)。メッセージが必要とされる書式でない

場合、メッセージは **DeadLetterQueue** へ転送され、それ以降の処理は行われません。送信先参照が存在しない場合、メッセージは拒否されます。

受信されたメッセージが **replyTo** 返信先を指定している場合、コマンド実行の結果は **object message** にラッピングされ、その返信先に送信されます。

command connection factory environment reference は、Java メッセージサービス接続を提供するために使用されるリソースマネージャを示します。

反対に、**JobListenerBean** は、*asynchronous continuations* をサポートするために **JbpmJobQueue** でジョブメッセージをリスンするメッセージ駆動型 Bean です。



注記

メッセージには、タイプ **long** の **jobId** と呼ばれるプロパティがなければなりません。このプロパティは、データベースで待機中の **Job** への参照が含まれていなければなりません。メッセージ本体が存在する場合は無視されます。

この Bean は **CommandListenerBean** を拡張し、後の環境エントリとカスタム化に利用可能なリソース参照を継承します。

表5.2 コマンド/ジョブリスナー Bean 環境

名前	種類	説明
ejb/LocalCommandServiceBean	EJB 参照	別の JBPM コンテキストでコマンドを実行するローカルセッション Bean へのリンクです。
jms/JbpmConnectionFactory	リソースマネージャ参照	結果メッセージを生成するために Java メッセージサービスの接続を提供するファクトリの論理名です。返信先を指定するコマンドメッセージに必要です。
jms/DeadLetterQueue	メッセージ送信先参照	コマンドを含まないメッセージは、ここで参照されるキューに送信されます。省略可能であり、使用しない場合はこのようなメッセージは拒否され、コンテナが再配信することがあります。

TimerEntityBean は、スケジューリングのため *Enterprise Java Bean* タイマサービスによって使用されます。有効期限が切れると、タイマーの実行は **command service Bean** に委譲されます。

TimerEntityBean は、ビジネスプロセスマネージャのデータソースにアクセスする必要があります。Enterprise Java Bean デプロイメント記述子は、エンティティ Bean をデータベースにマップする方法を定義しません (コンテナプロバイダによって定義されます)。JBoss Application Server では、**jbosscomp-jdbc.xml** 記述子がデータソースの JNDI 名とリレーショナルマッピングデータ (テーブルや列名など) を定義します。



注記

JBoss CMP (*container-managed persistence*) 記述子は、リソースマネージャ参照 (`java:comp/env/jdbc/JbpmDataSource`) ではなくグローバル JNDI 名 (`java:JbpmDS`) を使用します。



注記

ビジネスプロセスマネージャの以前のバージョンは、Enterprise Java Bean タイマーサービスと対話するために **TimerServiceBean** という名前のステートレスセッション Bean を使用しました。このセッション方式は、**cancellation** メソッドに回避できないボトルネックが発生する原因となっていたため廃止しなければなりません。セッション Bean に ID がいないため、タイマーサービスはすべてのタイマーを反復してキャンセルするタイマを見つけなければなりません。

後方互換性を維持する目的でこの Bean は今でも使用可能です。**TimerEntityBean** と同じ環境で動作するため、移行も簡単です。

表5.3 タイマーエンティティ / サービス Bean 環境

名前	種類	説明
<code>ejb/LocalCommandServiceBean</code>	EJB 参照	別の JBPM コンテキストでタイマーを実行するローカルのセッション Bean へのリンクです。

5.2. JBPM エンタープライズ設定

JBPM 設定は、`jbpm.cfg.xml` に含まれています。

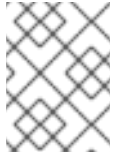
```
<jbpm-context>
  <service name="persistence"
    factory="org.jbpm.persistence.jta.JtaDbPersistenceServiceFactory" />
  <service name="message"
    factory="org.jbpm.msg.jms.JmsMessageServiceFactory" />
  <service name="scheduler"
    factory="org.jbpm.scheduler.ejbtimer.EntitySchedulerServiceFactory" />
</jbpm-context>
```

JtaDbPersistenceServiceFactory は、Business Process Manager が JTA トランザクションに参加できるようにします。既存のトランザクションがある場合は、JTA の永続サービスがそのトランザクションに「紐付け」されますが、そうでない場合は新規トランザクションが開始されます。Business Process Manager のエンタープライズ bean はトランザクション管理をコンテナに委譲するように設定されます。ただし、(Web アプリケーションなど) 有効なトランザクションがない環境において **JbpmContext** を作成する場合、新規トランザクションが自動的に開始されます。JTA persistence service factory には、下記の設定可能なフィールドが含まれます。

isCurrentSessionEnabled

これが **true** に設定されている場合、Business Process Manager は、作動中の JTA トランザクションに関連付いた現在の **Hibernate** セッションを使います。これはデフォルト設定です (詳細は http://www.hibernate.org/hib_docs/v3/reference/en/html/architecture.html#architecture-current-session を参照してください)。

コンテキストセッションの仕組みを活用しアプリケーションの別の箇所で JBPM と同じセッションを使います。これは `SessionFactory.getCurrentSession()` への呼び出しを使うことで実行します。または、`isCurrentSessionEnabled` を `false` に設定し、`JbpmContext.setSession(session)` メソッドを使い当セッションを投入することで、JBPM へ `Hibernate` セッションを提供します。これにより、JBPM が同アプリケーションの他の部分と同じ `Hibernate` セッションを使うようにします。



注記

Hibernate セッションは (永続コンテキストなどを使い) ステートレスセッション bean に注入可能です。

isTransactionEnabled

これが `true` に設定されると、JBPM は `JbpmConfiguration.createJbpmContext()` メソッドを使いコミットさせることで、**Hibernate** の `transaction API` からトランザクションを開始します (`JbpmContext.close()` が呼び出されると **Hibernate** セッションは終了します)。



警告

Business Process Manager が **EAR** としてデプロイされている、つまり `isTransactionEnabled` がデフォルトで `false` に設定されている場合、これは望ましい動作ではありません (詳細は http://www.hibernate.org/hib_docs/v3/reference/en/html/transactions.html#transaction-demarcation を参照してください)。

JmsMessageServiceFactory は、**Java Message Service** インターフェースで公開された信頼性のある通信インフラストラクチャーを活用することで、**asynchronous continuation messages** を **JobListenerBean** に配信します。**JmsMessageServiceFactory** は以下の設定可能なフィールドを公開します。

connectionFactoryJndiName

JNDI の初期コンテキストでの **JMS** 接続ファクトリ名で、デフォルトは `java:comp/env/jms/JbpmConnectionFactory` となっています。

destinationJndiName

ジョブメッセージが送信される **JMS** の宛て先名で、**JobListenerBean** のメッセージ受信元と一致します。デフォルトは `java:comp/env/jms/JobQueue` となっています。

isCommitEnabled

Business Process Manager が **Java Message Service** セッションを `JbpmContext.close()` 時にコミットすべきか指定します。**JMS** メッセージサービスが作成するメッセージは、現在のトランザクションがコミットする前に受信されるべきではありません。そのため、このサービスが作成したセッションは常に処理されます。デフォルト値は `false` ですが、**JTA** トランザクション全体で **Java Message Service** セッションが制御されているため、使用中の `connection factory` が **XA** 対応で

ある場合に適しています。反対に、JMS接続ファクトリがXA対応でない場合、このフィールドを`true`に設定し、Business Process Manager がJMSセッションのローカルトランザクションを明示的にコミットするようにしてください。

EntitySchedulerServiceFactory を使いビジネスプロセスタイマーをスケジュールします。Enterprise Java Bean コンテナ提供の予定イベントに対してトランザクション通知サービス上に構築することで行います。**EJBscheduler service factory**には、以下のような設定可能なフィールドがあります。

timerEntityHomeJndiName

JNDI初期コンテキストにおける**TimerEntityBean**のローカルホームインターフェース名で、デフォルト値は`java:comp/env/ejb/TimerEntityBean`となっています。

5.3. HIBERNATE エンタープライズ設定

`hibernate.cfg.xml`ファイルには以下の設定アイテムが含まれます。これらを変更し、他のデータベースやアプリケーションサービスをサポートします。

```
<!-- sql dialect -->
<property name="hibernate.dialect">
    org.hibernate.dialect.HSQLDialect
</property>

<property name="hibernate.cache.provider_class">
    org.hibernate.cache.HashtableCacheProvider
</property>

<!-- DataSource properties (begin) -->
<property name="hibernate.connection.datasource">
    java:comp/env/jdbc/JbpmDataSource
</property>
<!-- DataSource properties (end) -->

<!-- JTA transaction properties (begin) -->
<property name="hibernate.transaction.factory_class">
    org.hibernate.transaction.JTATransactionFactory
</property>
<property name="hibernate.transaction.manager_lookup_class">
    org.hibernate.transaction.JBossTransactionManagerLookup
</property>
<!-- JTA transaction properties (end) -->

<!-- CMT transaction properties (begin) ==
<property name="hibernate.transaction.factory_class">
    org.hibernate.transaction.CMTTransactionFactory
</property>
<property name="hibernate.transaction.manager_lookup_class">
    org.hibernate.transaction.JBossTransactionManagerLookup
</property>
==== CMT transaction properties (end) -->
```

hibernate.dialect 設定をお使いのデータベース管理システムに適したものに置き換えます (詳細はhttp://www.hibernate.org/hib_docs/v3/reference/en/html/session-configuration.html#configuration-optional-dialectsを参照してください)。

HashtableCacheProvider は、他の対応キャッシュプロバイダーで置き換えることが可能です (詳細はhttp://www.hibernate.org/hib_docs/v3/reference/en/html/performance.html#performance-cacheを参照してください)。

デフォルトで、JBPMは**JTATransactionFactory**を使うよう設定されています。既存のトランザクションがある場合、JTAトランザクションファクトリはこれを使いますが、ない場合は新規トランザクションを作成します。JBPM エンタープライズ bean はコンテナへトランザクション管理を委譲するよう設定されています。ただし、(Web アプリケーションなど) アクティブなトランザクションがないコンテキストでJBPM APIが使われている場合、自動で起動します。

Container-Managed Transactions (CMT)の利用時に予定外にトランザクションが作成されないように、**CMTTransactionFactory**へ切り替えます。この設定により、**Hibernate** は常に既存のトランザクションを検索し、何も見つからなかった場合は問題を報告します。

5.4. クライアントのコンポーネント

エンタープライズサービスを有効活用可能な**Business Process Manager API**に直接記述されているクライアントコンポーネントに対する配備記述子に、適切な環境参照が設定されているよう確認してください。以下の記述子は、クライアントセッション bean にとっては典型的なものとみなされています。

```
<session>

  <ejb-name>MyClientBean</ejb-name>
  <home>org.example.RemoteClientHome</home>
  <remote>org.example.RemoteClient</remote>
  <local-home>org.example.LocalClientHome</local-home>
  <local>org.example.LocalClient</local>
  <ejb-class>org.example.ClientBean</ejb-class>
  <session-type>Stateless</session-type>
  <transaction-type>Container</transaction-type>

  <ejb-local-ref>
    <ejb-ref-name>ejb/TimerEntityBean</ejb-ref-name>
    <ejb-ref-type>Entity</ejb-ref-type>
    <local-home>org.jbpm.ejb.LocalTimerEntityHome</local-home>
    <local>org.jbpm.ejb.LocalTimerEntity</local>
  </ejb-local-ref>

  <resource-ref>
    <res-ref-name>jdbc/JbpmDataSource</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>

  <resource-ref>
    <res-ref-name>jms/JbpmConnectionFactory</res-ref-name>
    <res-type>javax.jms.ConnectionFactory</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>

  <message-destination-ref>
```

```

    <message-destination-ref-name>
        jms/JobQueue
    </message-destination-ref-name>
    <message-destination-type>javax.jms.Queue</message-destination-type>
    <message-destination-usage>Produces</message-destination-usage>
</message-destination-ref>

</session>

```

上記の環境参照は、以下のように対象となる操作環境のリソースにバインドされる可能性があります。JNDI名は **Business Process Mananer** エンタープライズ bean が使う値と一致する点に注意してください。

```

<session>

    <ejb-name>MyClientBean</ejb-name>
    <jndi-name>ejb/MyClientBean</jndi-name>
    <local-jndi-name>java:ejb/MyClientBean</local-jndi-name>

    <ejb-local-ref>
        <ejb-ref-name>ejb/TimerEntityBean</ejb-ref-name>
        <local-jndi-name>java:ejb/TimerEntityBean</local-jndi-name>
    </ejb-local-ref>

    <resource-ref>
        <res-ref-name>jdbc/JbpmDataSource</res-ref-name>
        <jndi-name>java:JbpmDS</jndi-name>
    </resource-ref>

    <resource-ref>
        <res-ref-name>jms/JbpmConnectionFactory</res-ref-name>
        <jndi-name>java:JmsXA</jndi-name>
    </resource-ref>

    <message-destination-ref>
        <message-destination-ref-name>
            jms/JobQueue
        </message-destination-ref-name>
        <jndi-name>queue/JbpmJobQueue</jndi-name>
    </message-destination-ref>

</session>

```

クライアントコンポーネントがエンタープライズ bean でなく Web アプリケーションの場合、配備記述子は以下のようになるはずです。

```

<web-app>

    <servlet>
        <servlet-name>MyClientServlet</servlet-name>
        <servlet-class>org.example.ClientServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>MyClientServlet</servlet-name>
        <url-pattern>/client/servlet</url-pattern>
    </servlet-mapping>

```

```

</servlet-mapping>

<ejb-local-ref>
  <ejb-ref-name>ejb/TimerEntityBean</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <local-home>org.jbpm.ejb.LocalTimerEntityHome</local-home>
  <local>org.jbpm.ejb.LocalTimerEntity</local>
  <ejb-link>TimerEntityBean</ejb-link>
</ejb-local-ref>

<resource-ref>
  <res-ref-name>jdbc/JbpmDataSource</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>

<resource-ref>
  <res-ref-name>jms/JbpmConnectionFactory</res-ref-name>
  <res-type>javax.jms.ConnectionFactory</res-type>
  <res-auth>Container</res-auth>
</resource-ref>

<message-destination-ref>
  <message-destination-ref-name>
    jms/JobQueue
  </message-destination-ref-name>
  <message-destination-type>javax.jms.Queue</message-destination-type>
  <message-destination-usage>Produces</message-destination-usage>
  <message-destination-link>JobQueue</message-destination-link>
</message-destination-ref>

</web-app>

```

上記の環境参照は、このコード例に従い、対象の操作環境にあるリソースとバインドされる可能性があります。

```

<jboss-web>

<ejb-local-ref>
  <ejb-ref-name>ejb/TimerEntityBean</ejb-ref-name>
  <local-jndi-name>java:ejb/TimerEntityBean</local-jndi-name>
</ejb-local-ref>

<resource-ref>
  <res-ref-name>jdbc/JbpmDataSource</res-ref-name>
  <jndi-name>java:JbpmDS</jndi-name>
</resource-ref>

<resource-ref>
  <res-ref-name>jms/JbpmConnectionFactory</res-ref-name>
  <jndi-name>java:JmsXA</jndi-name>
</resource-ref>

<message-destination-ref>
  <message-destination-ref-name>
    jms/JobQueue

```



```
        </message-destination-ref-name>  
        <jndi-name>queue/JbpmJobQueue</jndi-name>  
    </message-destination-ref>
```

```
</jboss-web>
```

5.5. まとめ

本章をお読みいただき、Java EE インフラストラクチャで利用できる JBPM の機能を十分理解していただけたと思います。また、ご自分の企業の環境でこれら機能の一部を安心してテストしていただけると幸いです。

第6章 プロセスのモデリング

6.1. 便利な定義

本項を読んで、本書全体で使われる用語について学びましょう。

プロセス定義は、ビジネスプロセスの形式仕様を表したもので、**有向グラフ**に基づいています。そのグラフは、**ノード**と**遷移**で構成されています。グラフ内の全てのノードは、特定のタイプを持っています。ノードのタイプはランタイムの動作を定義します。プロセス定義は、開始状態を1つのみ持っています。

トークンは実行パスです。トークンは、グラフ中でノードへのポインターを維持するランタイムの概念です。

プロセスインスタンスとは、ひとつのプロセス定義の実行です。プロセスインスタンスが作成される時、実行のメインパスのトークンも作成されます。このトークンは、プロセスインスタンスのルートトークンと呼ばれ、プロセス定義の**開始状態**に位置しています。

シグナルがトークンに、グラフ実行の継続を指示します。名前のないシグナルを受け取った場合、トークンは、デフォルト**退場遷移**で、その時に位置していたノードを退場します。**遷移名**がシグナル上で、指定されている場合、指定されている遷移で、その時のノードを退場します。プロセスインスタンスに与えられたシグナルは、ルートトークンに委譲されます。

トークンがノードに入場後、そのノードは実行されます。ノード自身はグラフ実行の継続の責務があります。トークンをノードから退場させることで、グラフ実行の継続が行われます。各ノードタイプは、グラフ実行のための異なる動作を実装することもできます。実行を渡さないノードは、**状態**として振る舞います。

アクションは、実行プロセスの中で、イベント上で実行される **Java コード**の一部です。グラフは、ソフトウェア要件を伝達する重要な道具です。しかし、グラフは、作成されるソフトウェアの単なるビュー(投影)にすぎず、多くの技術的な詳細を隠します。アクションは、グラフ表現とは別に、技術的な詳細を追加するメカニズムです。グラフが導入されると、アクションを付け足すことが可能です。主なイベントタイプは、**entering a node**、**leaving a node**、**taking a transition**です。

定義について理解いただけたと思います。次にプロセスモデリングの仕組みについて説明します。

6.2. プロセスグラフ

プロセス定義の基本は、ノードと遷移で作成されるグラフです。その情報は **XML** によって表現され、**processdefinition.xml** という名前の **XML ファイル**に格納されます。各ノードは **type (state、decision、fork、join など)** を持たなければなりません。各ノードは、**退場遷移**のセットを持っています。明確にするために、ノードから退場する遷移に名前付けることができます。下図はオークションプロセスのプロセスグラフを表しています。

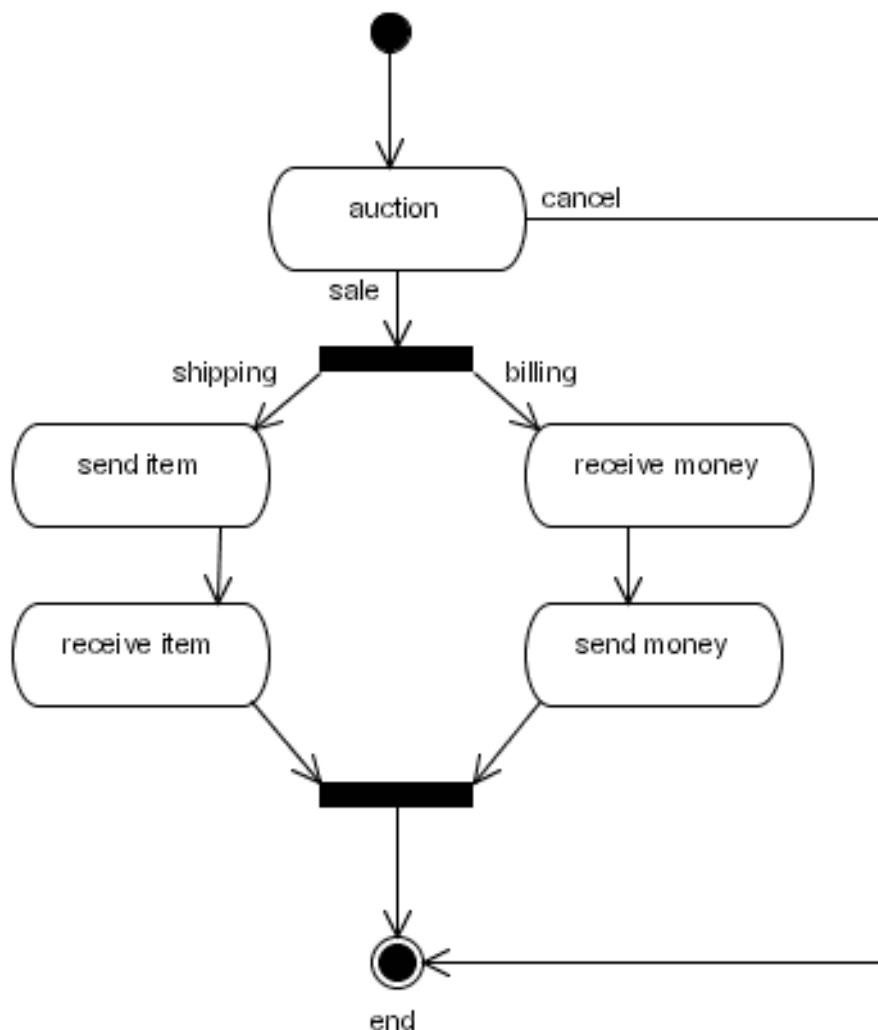


図6.1 オークションプロセスグラフ

下記は、XML で表現したオークションプロセスのプロセスグラフです。

```

<process-definition>

  <start-state>
    <transition to="auction" />
  </start-state>

  <state name="auction">
    <transition name="auction ends" to="salefork" />
    <transition name="cancel" to="end" />
  </state>

  <fork name="salefork">
    <transition name="shipping" to="send item" />
    <transition name="billing" to="receive money" />
  </fork>

  <state name="send item">
    <transition to="receive item" />
  </state>

  <state name="receive item">

```

```

    <transition to="salejoin" />
</state>

<state name="receive money">
    <transition to="send money" />
</state>

<state name="send money">
    <transition to="salejoin" />
</state>

<join name="salejoin">
    <transition to="end" />
</join>

<end-state name="end" />

</process-definition>

```

6.3. ノード

プロセスグラフはノードと遷移によって構成されます。各ノードは特定のタイプです。ノードタイプは、ランタイムにノードへ実行が入場した時に何をするかを決定します。ビジネスプロセスマネージャは使用するノードタイプのセットを提供します。この代わりに、特別なノード動作を実装するためカスタムコードを書くこともできます。

6.3.1. ノードの責任

各ノードが担当するのは主に2つあり、1つ目は、ノードは通常ノードの関数に関連するプレーン Java コードを実行できること、2つ目はプロセス実行を渡すことです。

ノードがプロセス実行を渡そうとすると、次のようなオプションを選択しなければならないことがあります。ノードは最も適切なコースに従います。

1. 実行を伝播できません(ノードは **wait state** として挙動します)。
2. ノードの **leaving transitions** の1つで実行を伝播できます(既にノードに入場したトークンが、API 呼び出し **executionContext.leaveNode(String)** により、**executionContext.leaveNode(String)** の1つで渡されることを意味します)。このノードは、カスタムプログラミングロジックの一部を実行し、待機背図にプロセス実行を自動的に継続するため、ある意味自動的に動作するようになります。
3. ノードは、新しい実行パスを表す新しいトークンの作成を決定できます。新しい各トークンはノードの **leaving transitions** 上で開始されます。このような動作のよい例が **fork node** になります。
4. 実行パスを終了できます。これは、トークンが終了したことを意味します。
5. プロセスインスタンスはランタイム構造全体を変更できます。ランタイム構造は、各トークンが実行パスを表すトークンのツリーを含んだプロセスインスタンスです。ノードは、トークンの作成や終了を行い、各トークンをグラフのノードに置いたり、遷移でトークンを開始することができます。

ビジネスプロセスマネージャーには事前実装のノードタイプのセットが含まれています。各ノードタイプは特定の設定と動作を持っていますが、独自のノード動作を書いて、プロセスで使用することもできます。

6.3.2. ノードタイプ: タスクノード

タスクノードは、人間が実行する1つ以上のタスクを表します。そのため、実行プロセスがノードに入場すると、ワークフロー参加者に属するリストにタスクインスタンスが作成されます。中に作成されます。その後、ノードは、**wait state**に入場します。ユーザーがタスクを終えると、実行がトリガーされ再開されます。

6.3.3. ノードタイプ: 状態

状態は、最低限機能を備えた**wait state**です。タスクノードとの違いは、タスクリスト中ではどのようなタスクインスタンスも作成されないことです。これは、プロセスが外部システムを待つ場合に役立ちます。その後で、プロセスは待機状態に入ります。外部システムが応答メッセージを送信すると、通常 **token.signal()** が呼び出され、プロセス実行の再開がトリガーされます。

6.3.4. ノードタイプ - 決定

決定をモデリングする方法は2つあり、どちらの方法を選択するかはユーザーが判断します。2つの方法は次の通りです。

1. 決定はプロセスにより行われ、プロセス定義に指定されます。
2. 外部のエンティティが決定します。

プロセスが決定を行う場合、**decision node** が使用されます。2つの方法の1つを決定基準として指定します。最も簡単な指定方法は、遷移へ **condition** 要素を追加することです (条件は **EL** 表現またはブール型変数を返す **beanshell** スクリプトです)。

ランタイム時に決定ノードは条件を指定した **leaving transitions** 上で最初にループします。これにより、**xml** で指定された順序でこれらの遷移が評価されます。条件が **true** に解決された最初の遷移が取得されます。条件を持つすべての遷移が **false** に解決されると、デフォルトの遷移 (**XML** で最初の遷移) が取得されます。

他の方法は取得する遷移の名前を返す表現を使用することです。 **expression** 属性を使用して決定上で表現を指定します。これは、決定ノードの **leaving transitions** の1つへ解決する必要があります。

handler 要素を使用して、決定ノードで指定できる **DecisionHandler** インターフェースの実装を指定できるため、決定で **handler** 要素を使用することができます。この場合、決定は **Java** クラスで計算され、選択された **leaving transition** が **DecisionHandler** 実装の **decide** メソッドによって返されます。

決定が外部パーティによって実行される場合、**state** や **wait state** を退場する複数の遷移を常に使用します。退場遷移は、**wait state** の終了後、実行処理を再開させる外部トリガーに用意できます (例えば、**Token.signal(String transitionName)** や **TaskInstance.end(String transitionName)** など)。

6.3.5. ノードタイプ - フォーク

フォークは、1つの実行パスを複数の並列実行パスに分岐します。デフォルトでは、フォークは退場する各遷移の子トークンを作成します(よって、フォークに入場するトークンとの親子関係を作成します)。

6.3.6. ノードタイプ - ジョイン

デフォルトでは、ジョインはジョイン自体へ入場するすべてのトークンが同じ親の子であると仮定します(前述のようにフォークを使用したり、同じジョインに入場するフォークによってすべてのトークンが作成されるとこのような状態が発生します)。

ジョインは、ジョインに入場したすべてのトークンを終了します。次に、ジョインは、ジョインに入場したトークンの親子関係を調べます。すべての兄弟トークンがジョインに入場した場合、親トークンは、**leaving transition**へ渡されます。まだ兄弟トークンがアクティブな場合、ジョインは **wait state** として振る舞います。

6.3.7. ノードタイプ - ノード

カスタムコードを書く必要をなくすためこのノードを使用します。このノードは実行がノードに入場すると実行される1つのサブエレメントアクションのみを予期します。**actionhandler** に記述されたカスタムコードはどのような処理も行えますが、実行を渡す責任もあることに注意してください(詳細は「[ノードの責任](#)」を参照してください)。

このノードは、**Java API** を利用して企業ビジネスアナリスト向けに関数論理を実装する時に使用することもできます。プロセスのグラフィック描写中に可視できるため、この方法は便利です(プロセスのグラフィック描写に不可視のコードを追加するにはアクションを使用します)。

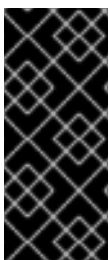
6.4. 遷移(TRANSITION)

遷移は、送り元ノードと送り先ノードを持っています。送り元ノードは、**from**プロパティで表され、送り先ノードは、**to**プロパティで表されます。

遷移は、オプションで名前を持つことが可能です(ビジネスプロセスマネージャーのほとんどの機能は遷移名の一意性に依存しています)。複数の遷移が同じ名前を持つと、最初の遷移が有効となります(ノードに重複する遷移名がある場合、**Map getLeavingTransitionsMap()** メソッドは、**List getLeavingTransitions()** より少ない要素を返します)。

6.5. アクション

アクションは、実行プロセスの中で、イベント上で実行される **Java** コードの一部です。グラフは、ソフトウェア要件を伝達する重要な道具ですが、作成されるソフトウェアの単なるビュー(投影)にすぎず、多くの技術的な詳細を隠します。また、アクションは、グラフ表現とは別に、技術的な詳細を追加するメカニズムで、グラフが導入されると、アクションを付け足すことが可能です。主なイベントタイプは、**entering a node**、**leaving a node**、**taking a transition**です。



重要

ノードに置かれたアクションと、イベントに置かれたアクションとは違いがあることに注意してください。イベントに置かれたアクションは、イベントが起動された時に実行されます。プロセスの制御フローに影響を与えることはありません(オブザーバーパターンに似ています)。一方で、ノードに置かれたアクションは、実行を渡す責務があります。

本項を読んで、イベント上のアクション例を見てみましょう。決められた遷移上で、データベースを更新する方法を表しています (データベースの更新は技術的に重要ですが、ビジネスアナリストにとっては重要ではありません)。

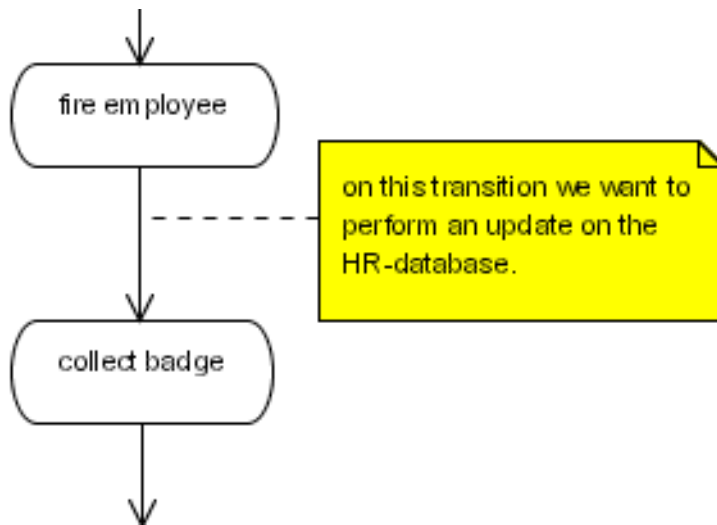


図6.2 データベース更新のアクション

```

public class RemoveEmployeeUpdate implements ActionHandler {
    public void execute(ExecutionContext ctx) throws Exception {
        // get the fired employee from the process variables.
        String firedEmployee =
            (String) ctx.getContextInstance().getVariable("fired employee");

        // by taking the same database connection as used for the jbpm
        // updates, we reuse the jbpm transaction for our database update.
        Connection connection =

ctx.getProcessInstance().getJbpmSession().getSession().getConnection();
        Statement statement = connection.createStatement();
        statement.execute("DELETE FROM EMPLOYEE WHERE ...");
        statement.execute();
        statement.close();
    }
}
  
```

```

<process-definition name="yearly evaluation">
    <state name="fire employee">
        <transition to="collect badge">
            <action class="com.nomercy.hr.RemoveEmployeeUpdate" />
        </transition>
    </state>

    <state name="collect badge">

</process-definition>
  
```



注記

カスタムアクションへの設定の追加については「[委譲設定](#)」を参照してください。

6.5.1. アクション参照

アクションに名前を付けることが可能です。名前を付けると、アクションが指定されている他の場所からアクションを参照することができます。名前付きのアクションを子要素としてプロセス定義へ追加することも可能です。

この機能を使用してアクション設定の重複に制限を設けます (アクションの設定が複雑であったり、ランタイムのアクションをスケジュールしたり実行したりする必要がある場合に便利です)。

6.5.2. イベント

イベントはプロセス実行における特定の瞬間です。ビジネスプロセスマネージャーのエンジンは、ソフトウェアが次の状態を算出する時 (シグナルを処理する時) に発生するグラフ実行の間にイベントを「ファイア」します。イベントは常にプロセス定義の要素に関連します。

プロセス要素の多くが異なるタイプのイベントを発生することができます。例えば、ノードの場合、**node-enter** イベントと **node-leave** イベントの両方を発生することができます (イベントはアクションの「フック」です。各イベントはアクションの一覧を持っています。JBPM エンジンがイベントを開始すると、アクションの一覧が実行されます)。

6.5.3. イベントを渡す

Super-state は、プロセス定義の要素に親子関係を作成します (**super-state** に含まれるノードと遷移はその **super-state** を親とします。トップレベルの要素は、プロセス定義を親としますが、そのプロセス定義は、親を持ちません)。イベントが発生すると、そのイベントは親階層に渡されます。これによりプロセス内ですべての遷移イベントをキャプチャーしたり、集中化された場所でアクションとこれらのイベントに関連付けることができるようになります。

6.5.4. スクリプト

スクリプトは、**Beanshell** スクリプトを実行するアクションです (**Beanshell** の詳細は、<http://www.beanshell.org/> を参照してください)。デフォルトでは、すべてのプロセス変数をスクリプト変数として使用できますが、スクリプト変数はプロセス変数には記述できません。次のスクリプト変数を使用できます。

- `executionContext`
- `token`
- `node`
- `task`
- `taskInstance`

```
<process-definition>
  <event type="node-enter">
    <script>
      System.out.println("this script is entering node "+node);
    </script>
  </event>
  ...
</process-definition>
```


変数をスクリプトにロードしたり保存するデフォルトの動作をカスタマイズするには、**variable** 要素をスクリプトのサブ要素として使用します。この場合、スクリプト表現は、スクリプトのサブ要素 **expression** に置かなければいけません。

```
<process-definition>
  <event type="process-end">
    <script>
      <expression>
        a = b + c;
      </expression>
      <variable name='XXX' access='write' mapped-name='a' />
      <variable name='YYY' access='read' mapped-name='b' />
      <variable name='ZZZ' access='read' mapped-name='c' />
    </script>
  </event>
  ...
</process-definition>
```

スクリプトが開始する前に、プロセス変数 **YYY** と **ZZZ** は、それぞれスクリプト変数 **b** と **c** として、スクリプトで使えるようになります。スクリプトが終了後、スクリプト変数 **a** の値は、プロセス変数 **XXX** に保存されます。

変数の **access** 属性に **read** が含まれる場合、スクリプト評価の前にプロセス変数はスクリプト変数として読み込まれます。**access** 属性に **write** が含まれる場合、スクリプト評価の後にプロセス変数はスクリプト変数として保存されます。**mapped-name** 属性は、スクリプト中で別名でプロセス変数を使用可能にします。プロセス変数の名前に空白文字や無効な文字が含まれている場合にこの属性を使用します。

6.5.5. カスタムイベント

プロセスの実行中にカスタムイベントを実行したい場合は、**GraphElement.fireEvent(String eventType, ExecutionContext executionContext);** メソッドを呼び出します。イベントタイプの名前は自由に選択できます。

6.6. SUPER-STATES

Super-state はノードのグループです。再帰的にネストし、プロセス定義へ階層を追加するために使用することができます。例えば、この機能を使用してフェーズの 1 プロセスに属するすべてのノードをグループ化することができます。

アクションを **super-state** のイベントに関連付けることもできます。ネスト化されたノードでトークンにより開始されたイベントは、プロセス定義まで **super-state** 階層まで上がっていきます。そのため、階層内の全ノードに同時に存在するように機能します。これは、実行プロセスが起動フェーズにあるかをチェックする時などに便利です。

6.6.1. Super-state 遷移

Super-state を退出する遷移は、その **super-state** 内にあるノードのいずれかに置かれているトークンにより取得可能です。この機能のユースケースの 1 つに、いつでも行うことのできる **キャンセル** の遷移モデルが挙げられます。

遷移は **Super-state** にも到達可能です。その場合、トークンは最初のノードに文書の順番にリダイレクトされます。**Super-state** の外側のノードから **super-state** の内側のノードに直接遷移することも可能です。逆に、**Super-state** の内側のノードから **super-state** の外側のノードの遷移も、また **Super-state**

そのものへの遷移も可能です。 **super-state** は、自身を参照することもできます。

6.6.2. Super-state イベント

Super-state 固有のイベントは **superstate-enter** と **superstate-leave** の2つがあります。これらのイベントはノードが入退場した遷移に関係なく発生します (**super-state** 内でトークンが遷移する限り、イベントは発生しません)。



注記

状態と **super-state** には別々のイベントタイプがあります。実際の **super-state** イベントと、**super-state** 内より渡されたノードイベントを簡単に区別できるようにするため、ソフトウェアがこのように設計されました。

6.6.3. 階層名

ノード名はスコープ内で固有でなければなりません。ノードのスコープは、そのノードコレクションのことです (プロセス定義と **super-state** は、ノードコレクションです)。 **Super-state** にあるノードを参照するためには、スラッシュ (/) で分けた名前を指定しなければなりません (スラッシュはノード名を分けます。上位レベルを参照する場合は、. を利用します)。次のサンプルは、**super-state** にあるノードの参照する方法になります。

```
<process-definition>
  <state name="preparation">
    <transition to="phase one/invite murphy"/>
  </state>
  <super-state name="phase one">
    <state name="invite murphy"/>
  </super-state>
</process-definition>
```

次の例は親階層へ上がる方法を表しています。

```
<process-definition>
  <super-state name="phase one">
    <state name="preparation">
      <transition to="../phase two/invite murphy"/>
    </state>
  </super-state>
  <super-state name="phase two">
    <state name="invite murphy"/>
  </super-state>
</process-definition>
```

6.7. 例外ハンドリング

ビジネスプロセスマネージャーの例外ハンドリングメカニズムは **Java** の例外のみに適用できます。グラフ実行自身が問題の原因になることはありません。委譲クラスが実行された時のみ例外が発生します。

exception-handler の一覧を **process-definition**、**node**、**transition** で指定することができます。各例外ハンドラはアクションの一覧を持ちます。委譲されたクラスで例外が発生すると、適切な **exception-handler** に対してプロセス要素の親階層が検索され、アクションが実行されます。



重要

ビジネスプロセスマネージャの例外ハンドリングは、Java 例外ハンドリングとは一部異なります。Javaでは、検出された例外は制御フローに影響を与えます。JBPM では、制御フローを例外ハンドリングメカニズムで変更できません。例外は検出されるか、されないかのいずれかになります。キャッチされなかった例外は、`token.signal()` を呼び出したクライアントへスローされます。検出された例外については、何も発生しなかったようにグラフ実行が継続されます。



注記

`Token.setNode(Node node)` を使用して、例外ハンドリング **action** のグラフ内にある任意のノードにトークンを置きます。

6.8. プロセス構成

ビジネスプロセスマネージャは **process-state** を用いてプロセス構成をサポートします。別のプロセス定義に関連する状態があります。グラフ実行が **process-state** に入場すると、サブプロセスの新しいインスタンスが作成されます。そのサブプロセスはプロセス状態に入場した実行パスに関連付けられます。親プロセスの実行パスは、サブプロセスが終了するまで待機し、サブプロセス終了後にプロセス状態を退場し、スーパープロセスでグラフ実行を続けます。

```
<process-definition name="hire">
  <start-state>
    <transition to="initial interview" />
  </start-state>
  <process-state name="initial interview">
    <sub-process name="interview" />
    <variable name="a" access="read,write" mapped-name="aa" />
    <variable name="b" access="read" mapped-name="bb" />
    <transition to="..." />
  </process-state>
  ...
</process-definition>
```

上記の例では、**hire** プロセスには **interview** プロセスを引き起こす **process-state** が含まれています。実行が **first interview** に入場すると、**interview** プロセスの新しい実行 (プロセスインスタンス) が作成されます。バージョンを明示的に指定しないと、サブプロセスの最新バージョンが使用されます。ビジネスプロセスマネージャが特定バージョンのインスタンスを作成するには、任意の **version** 属性を指定します。サブプロセスが実際に作成されるまで指定バージョンまたは最新バージョンのバインディングを延期するには、任意の **binding** 属性を **late** に設定します。

次に、**hire** プロセス変数 **a** が **interview** プロセス変数 **aa** へコピーされます。同様に、**hire** 変数 **b** が **interview** 変数 **bb** へコピーされます。**interview** プロセスが終了すると、変数 **aa** のみが **a** 変数にコピーし直されます。

一般的に、サブプロセスが開始すると、読み取りアクセスを持つ変数はすべて、スーパープロセスから読み込まれ、新たに作成されたサブプロセスへ送り込まれます。これは、開始ステートから退場するためシグナルが渡される前に発生します。サブプロセスインスタンスが終了すると、書き込みアクセスを持ったすべての変数は、サブプロセスからスーパープロセスにコピーされます。変数の **mapped-name** 属性を使用してサブプロセスで使用する変数名を指定します。

6.9. カスタムノードの動作

すべてのビジネスロジックを実行でき、グラフ実行を渡す役目を果たす **ActionHandler** の特別な実装を使用してカスタムノードを作成します。以下は、ERP システムより値を読み取り、プロセス変数より数値を追加し、結果を ERP システムに保存する例になります。数値の大きさを基に、**small amounts** 遷移または **large amounts** 遷移を使用して終了します。

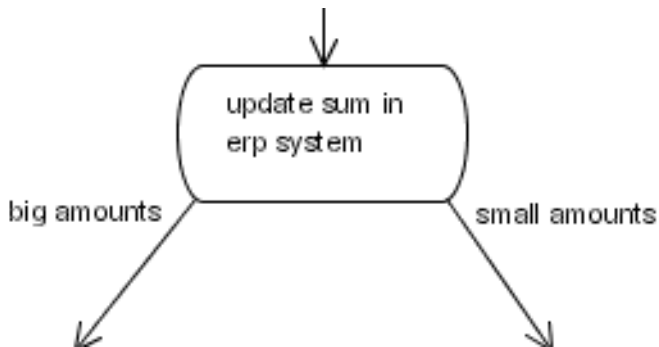


図6.3 ERP サンプルを更新するためのプロセススニペット

```

public class AmountUpdate implements ActionHandler {
    public void execute(ExecutionContext ctx) throws Exception {
        // business logic
        Float erpAmount = ...get amount from erp-system...;
        Float processAmount = (Float)
        ctx.getContextInstance().getVariable("amount");
        float result = erpAmount.floatValue() + processAmount.floatValue();
        ...update erp-system with the result...;

        // graph execution propagation
        if (result > 5000) {
            ctx.leaveNode(ctx, "big amounts");
        } else {
            ctx.leaveNode(ctx, "small amounts");
        }
    }
}

```



注記

カスタムノード実装でトークンの作成や結合も可能です この方法について学ぶため、JBPM ソースコードにある **Fork** と **Join** のノードの実装を見てみましょう。

6.10. グラフ実行

ビジネスプロセスマネージャーのグラフ実行モデルは、プロセス定義と「コマンドチェーンパターン」の解釈に基づいています。

プロセス定義のデータはデータベースに保存され、プロセス実行中に使用されます。



注記

Hibernate の 2 次レベルキャッシュは、ラインタイムに定義情報をロードしないようにするため使用されます。プロセス定義は変更しないため、**Hibernate** はプロセス定義をメモリにキャッシュすることができます。

「コマンドチェーンパターン」は、グラフの各ノードがプロセス実行を渡すようにします。ノードが渡さない場合、**wait state**のように挙動します。

実行がプロセスインスタンスを開始し、**wait state**に入るまで継続するようにしてみましょう。

トークンは実行パスを表します。トークンはプロセスグラフのノードへのポインタを持っています。**wait state**中、トークンをデータベースで永続化することができます。

このアルゴリズムはトークンの実行を算出するために使用されます。シグナルがトークンへ送信され、コマンドチェーンパターンより遷移とノードで渡されると実行が開始されます。関連するメソッドは次の通りです。

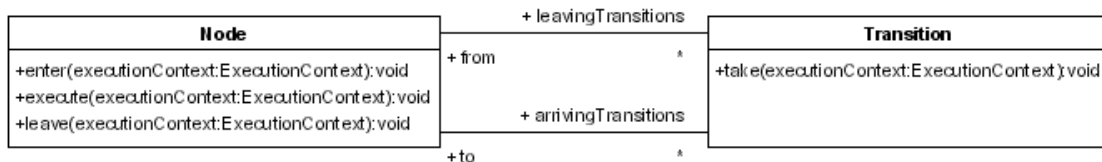


図6.4 グラフ実行に関連するメソッド

トークンがノードにある場合、シグナルをトークンに送ることが可能です。シグナルは実行を開始する指示として扱われるため、トークンの現ノードから **leaving transition** を指定しなければなりません。最初の遷移がデフォルトになります。トークンへのシグナルで、現ノードを取得し、**Node.leave(ExecutionContext, Transition)** メソッドを呼びます (**ExecutionContext** 内のオブジェクトはトークンであるため、**ExecutionContext** はトークンであると考えるのがよいでしょう)。メソッドは **node-leave** イベントを発生させ、**Transition.take(ExecutionContext)** を呼び出します。そのメソッドは遷移イベントを実行し、遷移の宛先ノードにある **Node.enter(ExecutionContext)** を呼び出します。

ノードの各タイプは独自の動作を持っており、**execute** メソッドより実装されます。各ノードは **Node.leave(ExecutionContext, Transition)** を再び呼び出してグラフ実行を渡す役割を果たします。要約は次の通りです。

- **Token.signal(Transition)**
- **Node.leave(ExecutionContext, Transition)**
- **Transition.take(ExecutionContext)**
- **Node.execute(ExecutionContext)**
- **Node.execute(ExecutionContext)**



注記

アクション呼び出しを含む次の状態は、クライアントのスレッドで算出されます。すべての計算がこの方法で行われなければならないという考えは一般的な誤解です。*非同期呼び出し*と同様に、*非同期メッセージング*(**Java Message Service** より)を利用できます。プロセスインスタンスの更新として、同じトランザクション内でメッセージが送信されると、同期の課題はすべて適切に対応されなければなりません。ワークフローシステムの一部は、グラフにあるすべてのノード間で、非同期メッセージングを利用します。しかし、スループットが高い環境では、このアルゴリズムは、ビジネスプロセスのパフォーマンスを最大限にするため、より優れた制御と柔軟性をもたらします。

6.11. トランザクション境界

「[グラフ実行](#)」で説明したように、ビジネスプロセスマネージャーはクライアントのスレッドでプロセスを実行し、本質的に同期処理されます。つまり、`token.signal()` や `taskInstance.end()` は、プロセスが新たに **wait state** に入った時のみ返されます。



注記

本項で説明した JPDL 機能に関する詳細は、[10章 非同期の続行](#)を参照してください。

プロセス実行をサーバー側トランザクションにバインドするのは簡単であるため、ほとんどの状況でこれが最も分かりやすい方法です。プロセスは、1つのトランザクション内である状態から次の状態に移動します。

インプロセスでの算出に時間が掛かる場合、この動作は望ましくないかもしれません。この問題に対応するには、ビジネスプロセスマネージャーに非同期でプロセスを継続できるようにする非同期メッセージングシステムが含まれるようにします (Java エンタープライズ環境では、組み込みのメッセージングシステムの代わりに、Java Message Service を使用するよう JBPM を設定することができます)。

JPDL は属性 `async="true"` 属性をすべてのノードでサポートします。非同期ノードはクライアントのスレッドで実行されません。代わりに、非同期メッセージングシステム上にメッセージが送信され、スレッドがクライアントに返されます (`token.signal()` か `taskInstance.end()` が返されます)。

ビジネスプロセスマネージャーのクライアントコードはトランザクションをコミットできるようになりました。プロセスの更新が含まれる同じトランザクションでメッセージを送信します (このようなトランザクションの結果として、トークンがまだ実行されていない次のノードへ移動し、`org.jbpm.command.ExecuteNodeCommand` メッセージが、非同期メッセージングシステムから `jBPM Command Executor` へ送信されます。これがキューからコマンドを読み取り、コマンドを実行します。`org.jbpm.command.ExecuteNodeCommand` の場合、ノードが実行されるとプロセスが継続されます。各コマンドは個別のトランザクションで実行されます)。



重要

非同期プロセスが継続されるよう、`jBPM Command Executor` が実行しているようにしてください。これには、Web アプリケーションの `CommandExecutionServlet` を設定します。



注記

プロセスモデラーは、非同期メッセージについて過度に心配する必要はありません。注意すべき主な点はトランザクション境界です。デフォルトで、ビジネスプロセスマネージャーはクライアントのトランザクションで操作し、プロセスが **wait state** になるまで全体の計算を行います (プロセスでトランザクションの境界を設定するには、`async="true"` を使用します)。

例は次の通りです。

```
<start-state>
  <transition to="one" />
</start-state>
<node async="true" name="one">
  <action class="com...MyAutomaticAction" />
</node>
```

```

    <transition to="two" />
</node>
<node async="true" name="two">
    <action class="com...MyAutomaticAction" />
    <transition to="three" />
</node>
<node async="true" name="three">
    <action class="com...MyAutomaticAction" />
    <transition to="end" />
</node>
<end-state name="end" />
...

```

プロセス実行の開始と再開に必要なクライアントコードは、通常の動機プロセスで必要となるクライアントコードと全く同じです。

```

//start a transaction
JbpmContext jbpmContext = jbpmConfiguration.createContext();
try {
    ProcessInstance processInstance =
        jbpmContext.newProcessInstance("my async process");
    processInstance.signal();
    jbpmContext.save(processInstance);
} finally {
    jbpmContext.close();
}

```

最初のトランザクションの後、プロセスインスタンスの **root token** は **node one** を示し、**ExecuteNodeCommand** が、コマンドエグゼキュータへ送信されます。

後続のトランザクションでは、コマンドエグゼキュータはキューからメッセージを読み込み、**node one** を実行します。実行を渡すか、それとも **wait state** に入るのかをアクションが決定することができます。アクションが実行を渡すよう決定した場合、実行が **node two** に入った時にトランザクションが終了します。

第7章 コンテキスト

本章を読んでプロセス変数について理解してください。プロセス変数はプロセスインスタンスに関連する情報を維持する鍵と値のペアです。



注記

ユーザーがコンテキストをデータベースに保存できなければならないため、若干の制限が一部適用されます。

7.1. プロセス変数へのアクセス

`org.jbpm.context.exe.ContextInstance` は、プロセス変数の中心的なインターフェースとなります。次のように `ContextInstance` をプロセスインスタンスより取得します。

```
ProcessInstance processInstance = ...;
ContextInstance contextInstance =
    (ContextInstance) processInstance.getInstance(ContextInstance.class);
```

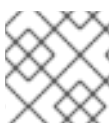
基本的な操作は次の通りです。

```
void ContextInstance.setVariable(String variableName, Object value);
void ContextInstance.setVariable(
    String variableName, Object value, Token token);

Object ContextInstance.getVariable(String variableName);
Object ContextInstance.getVariable(String variableName, Token token);
```

変数名は `java.lang.String` です。デフォルトでは、ビジネスプロセスマネージャは次の値タイプをサポートします(`Hibernate` と永続する他のクラスもサポートします)。

<code>java.lang.String</code>	<code>java.lang.Boolean</code>
<code>java.lang.Character</code>	<code>java.lang.Float</code>
<code>java.lang.Double</code>	<code>java.lang.Long</code>
<code>java.lang.Byte</code>	<code>java.lang.Integer</code>
<code>java.util.Date</code>	<code>byte[]</code>
<code>java.io.Serializable</code>	



注記

タイプのない `null` 値も永続的に保存できます。



警告

例外エラーの原因となるため、他のタイプがプロセス変数に保存されている場合はプロセスインスタンスを保存しないでください。

7.2. 変数のライフ

変数はプロセスアーカイブの中で宣言する必要はありません。実行時に、Java オブジェクトを変数に置くようにします。変数が存在しない場合はプレーン `java.util.Map` と同様に作成されます。変数を削除することもできます。

```
ContextInstance.deleteVariable(String variableName);
ContextInstance.deleteVariable(String variableName, Token token);
```

タイプは自動的に変化できます。そのため、タイプは異なるタイプの値で変数を上書きできます。タイプの変更はプレーンカラムの更新よりもデータベースとの通信が多くなるため、タイプ変更の回数を制限するようにすることが重要です。

7.3. 変数の永続性

変数はプロセスインスタンスの一部です。データベースにプロセスインスタンスを保存すると、データベースがプロセスインスタンスと同期されます(これにより変数が作成、更新、削除されます)。詳細については、[4章永続性](#)を参照してください。

7.4. 変数スコープ

各実行のパス(トークンとも呼ばれます)は、独自のプロセス変数のセットを持っています。変数は常に実行パス上で要求されます。プロセスインスタンスは、これらのパスのツリーを有しています。パスの指定なしに変数が要求された場合、デフォルトで **root token** トークンが使用されます。

変数ルックアップは再帰的に発生します。実行パスの親上で実行されます(プログラミング言語で変数がスコープされる方法と似ています)。

存在していない変数が実行パスに設定されると、その変数は **root token** 上に作成されます。そのため、デフォルトでは各変数はプロセススコープを持っています。変数トークンを「ローカル」にするには、次の例のように明示的に作成します。

```
ContextInstance.createVariable(String name, Object value, Token token);
```

7.4.1. 変数オーバーローディング

変数オーバーローディングとは、各実行パスが、同名の変数のコピーを持つことができることです。それらのコピーは、すべて独立して取り扱われ、異なるタイプであることができます。変数オーバーローディングは、複数の並列実行パスを同じ遷移で実行する場合に興味深いかもしれません。それらのパスを区別するのは対応する変数のセットのみであるからです。

7.4.2. 変数オーバーライディング

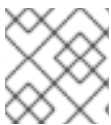
変数オーバーライディングは、ネストされた実行パスの変数がより広いグローバルの実行パスでの変数を上書きすることです。一般的に、ネストされた実行パスは並列処理に関係しています。そのフォークとジョイン間の実行パスは、フォークに入場した実行パスの子(ネストされた)になります。例えば、プロセスインスタンススコープにある **contact** という変数を、**shipping** および **billing** のネストされた実行パスにある変数で上書きできます。

7.4.3. タスクインスタンス変数スコープ

タスクインスタンス変数の詳細については、「[タスクインスタンス変数](#)」を参照してください。

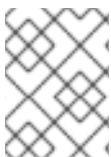
7.5. 一時変数

プロセスインスタンスがデータベースに永続された時、普通の変数も永続されます。しかし、データベースに保存せずに委譲クラスの変数を使用したい時があるかもしれません。このような場合、一時変数を使用します。



注記

一時変数の生存期間は、**ProcessInstance** Java オブジェクトと同じです。



注記

本質上、一時変数は実行パスとは関係ありません。そのため、プロセスインスタンスオブジェクトは一時変数の1つのマップのみを持ちます。

その一時的変数は、コンテキストインスタンスにある独自のメソッドのセットよりアクセス可能で、**processdefinition.xml** ファイルに宣言される必要はありません。

```
Object ContextInstance.getTransientVariable(String name);
void ContextInstance.setTransientVariable(String name, Object value);
```

本章ではプロセス変数の詳細について説明しました。本章を読み終えた今、プロセス変数について理解していただいたと思います。

第8章 タスク管理

JBPM の中心的な役割は、プロセスの実行を永続化することです。この機能はタスクやユーザー向けのタスクリストを管理したい場合に大変有用です。JBPM は全体的なプロセスを記述するソフトウェアのコンポーネントを指定することができます。このようなソフトウェアのコンポーネントではヒューマンタスクに対して待機状態を持つことができます。

8.1. タスク

タスクはプロセス定義の一部で、プロセス実行中にどのようにタスクインスタンスを作成、割り当てるかを定義します。

タスクは、**task-node**と **process-definition** 内で定義できます。最も一般的な方法は、1つ以上の**task**を**task-node**で定義することです。その場合、**task-node**は、ユーザが行うタスクを表現し、プロセス実行は、アクターがタスクを終了するまで待機するはずで、アクターがタスクを終了した時、プロセス実行は、継続します。**task-node**により多くのタスクが定義されたとき、デフォルトの動作は、すべてのタスクが終了するまで待機します。

タスクは、**process-definition**上にも指定できます。プロセス定義に指定されたタスクは、名前でもルックアップされ、**task-node**の中から参照されるか、またはアクションの中から利用することができます。実は、すべての名前付きタスク(**task-node**でも)は、**process-definition**内で名前でもルックアップできます。

タスク名は、プロセス定義全体で一意でなければいけません。また、タスクには、**priority**を設定できます。このタスク向けに作成された各タスクインスタンスに対し、優先順位の初期値として利用されます (後ほど、この初期の優先順位を変更することも可能です)。

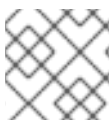
8.2. タスクインスタンス

タスクインスタンスは、**actorId(java.lang.String)**に割り当てることができます。すべてのタスクインスタンスは、データベースの1つのテーブル (**JBPM_TASKINSTANCE**) に保存されます。特定の**actorId**における全タスクインスタンスについて、このテーブルをクエリーすることで、その特定ユーザのタスクリストが取得できます。

タスクがプロセス実行と関係ない場合でも JBPM タスクリストのメカニズムを使うとJBPM タスクと他のタスクの結合が可能です。こうすることで、1つの集約リポジトリで、容易に JBPM プロセスタスクと他のアプリケーションのタスクを組み合わせることができます。

8.2.1. タスクインスタンスのライフサイクル

タスクインスタンスライフサイクルは分かりやすく簡単です。作成後、タスクインスタンスを起動可能です。次に、タスクインスタンスは終了でき、タスクインスタンスが完了としてマークされます。



注記

柔軟性を保つため、**Assignment**はライフサイクルの一部に含まれていません。

1. タスクインスタンスは通常、プロセス実行で (**TaskMgmtInstance.createTaskInstance(...)**メソッドを使い) **task-node** を入力すると作成されます

2. その後、ユーザーインターフェースのコンポーネントは、データベースをクエリし、タスクリストを入手します。**TaskMgmtSession.findTaskInstancesByActorId(...)** を使うことでクエリの実行ができます。
3. 次に、ユーザーからの入力を集め、UIコンポーネントは**TaskInstance.assign(String)**、**TaskInstance.start()**、あるいは**TaskInstance.end(...)**を呼び出します。

タスクインスタンスは、3つのdate-propertyにより状態を管理します。

1. create
2. start
3. end

TaskInstanceにある該当の"getter"でこれらのプロパティにアクセスします。

現在は、完了タスクインスタンスは終了日がマークされ、その結果、以降のタスクリストのクエリ検索で取得されません。しかし、これらの完了タスクインスタンスは、**JBPM_TASKINSTANCE** テーブルには残っています。

8.2.2. タスクインスタンスとグラフ実行

タスクインスタンスは、アクターのタスクリストにある項目です。**Signalling** タスクインスタンスはタスクインスタンスの1つで、完了時にトークンヘシグナルを送信しプロセス実行を継続します。また、別のタスクインスタンスに、**Blocking** タスクインスタンスというのがあり、関係のあるトークン (=実行パス)はこのタスクインスタンス終了前に、**task-node**を退場することはできません。デフォルトでは、タスクインスタンスは**signalling**および**non-blocking**となっています。

複数のタスクインスタンスが1つの**task-node**に関連付けられている場合、プロセス開発者は、タスクインスタンスが完了することでプロセスの継続にどのような影響与えるか指定することができます。**task-node**の**signal-property**にこれらの値を渡します。

last

デフォルトです。最後のタスクインスタンスが終了しても実行を続けます。このノードに入った際にタスクが作成されない場合、実行は継続されます。

last-wait

最後のタスクインスタンスが終了しても実行を続けます。このノードに入った際にタスクが作成されない場合、タスクノードにてタスクが作成されるまで実行を待機します。

first

最初のタスクインスタンスが終了しても実行を続けます。このノードに入った際にタスクが作成されない場合、実行は継続されます。

first-wait

最初のタスクインスタンスが終了しても実行を続けます。このノードに入った際にタスクが作成されない場合、タスクノードにてタスクが作成されるまで実行を待機します。

unsynchronized

この場合タスクの作成あるいは、未完了に拘らず常に実行は継続されます。

never

この場合、タスクの作成、あるいは未完了に拘らず、実行は継続されません。

タスクインスタンス作成はランタイムの計算に基づいている場合があります。その場合、**task-node**の**node-enter**イベント上の**ActionHandler**へ追加し、**create-tasks="false"**に設定します。以下に例を示します。

```
public class CreateTasks implements ActionHandler {
    public void execute(ExecutionContext executionContext) throws Exception
    {
        Token token = executionContext.getToken();
        TaskMgmtInstance tmi = executionContext.getTaskMgmtInstance();

        TaskNode taskNode = (TaskNode) executionContext.getNode();
        Task changeNappy = taskNode.getTask("change nappy");

        // now, 2 task instances are created for the same task.
        tmi.createTaskInstance(changeNappy, token);
        tmi.createTaskInstance(changeNappy, token);
    }
}
```

ここでは、作成されるタスクは、**task-node**に指定されます。それらは、**process-definition**でも指定ができ **TaskMgmtDefinition**からも取得できます (**TaskMgmtDefinition** は、タスク管理情報を追加することで、プロセス定義を継承します)。

TaskInstance.end() を使い、タスクインスタンスの完了をマークします。任意で、**end**メソッドに遷移を指定できます。タスクインスタンス完了が実行プロセスの継続を引き起こす場合、指定された遷移を通るため、その**task-node**から退場します。

8.3. 割り当て

プロセス定義はタスクノードを含みます。**task-node**には0個または1個以上のタスクが含まれます。タスクは、プロセス定義の一部であり静的な記述です。実行時、タスクはタスクインスタンス作成の結果として発生します。タスクインスタンスはユーザのタスクリストにおける1エントリに対応しています。

JBPM で、タスク管理のプッシュ (パーソナルタスクリスト) とプル (グループタスクリスト) のモデルの組み合わせが適用できます。プロセスは、タスクへの責務を導出して、ユーザのタスクリストにそれをプッシュすることができます。または、別の方法として、タスクはアクタープールに割り当てることができ、プールにある各アクターがタスクをプルして、それをアクターのタスクリストに置くことができます (詳細は「[パーソナルタスクリスト](#)」と「[グループタスクリスト](#)」を参照してください)。

8.3.1. Assignment インターフェース

タスクインスタンスの割り当ては、**AssignmentHandler** インターフェース経由で行われます。

```
public interface AssignmentHandler extends Serializable {
    void assign( Assignable assignable, ExecutionContext executionContext );
}
```

Assignment Handler実装はタスクインスタンスが作成されると呼び出されます。その時、タスクインスタンスは1つ以上のアクターに割り当てられます。**AssignmentHandler** 実装は、タスクを割り当て

るために割り当て可能なメソッド (`setActorId` あるいは `setPooledActors`) を呼び出します。割り当て可能なアイテムは、`TaskInstance` あるいは `SwimlaneInstance` (つまり、プロセスロール) です。

```
public interface Assignable {
    public void setActorId(String actorId);
    public void setPooledActors(String[] pooledActors);
}
```

`TaskInstance` と `SwimlaneInstance` の両方を指定されたユーザか、アクターのプールに割り当てることができます。ユーザに `TaskInstance` を割り当てるには、`Assignable.setActorId(String actorId)` を呼び出します。`TaskInstance` をプールのアクター候補に割り当てるために、`Assignable.setPooledActors(String[] actorIds)` を呼び出します。

プロセス定義の各タスクは `handler` 実装と関連付け、実行時に割り当てを行うことができます。

プロセス内で複数のタスクが同じ人、もしくはアクターのグループに割り当てられると、スイムレーンの使用を考慮します。「[スイムレーン](#)」を参照してください。

再利用可能な `AssignmentHandler` を作成するには、`processdefinition.xml` ファイルを使いそれぞれ設定します。割り当てハンドラへの設定を追加する方法については、「[委譲](#)」を参照してください。

8.3.2. 割り当てデータモデル

アクターへのタスクインスタンスとスイムレーンインスタンスの割り当てを管理するためのデータモデルは以下のとおりです。各 `TaskInstance` は、`actorId` と プールされたアクターのセットを保持しています

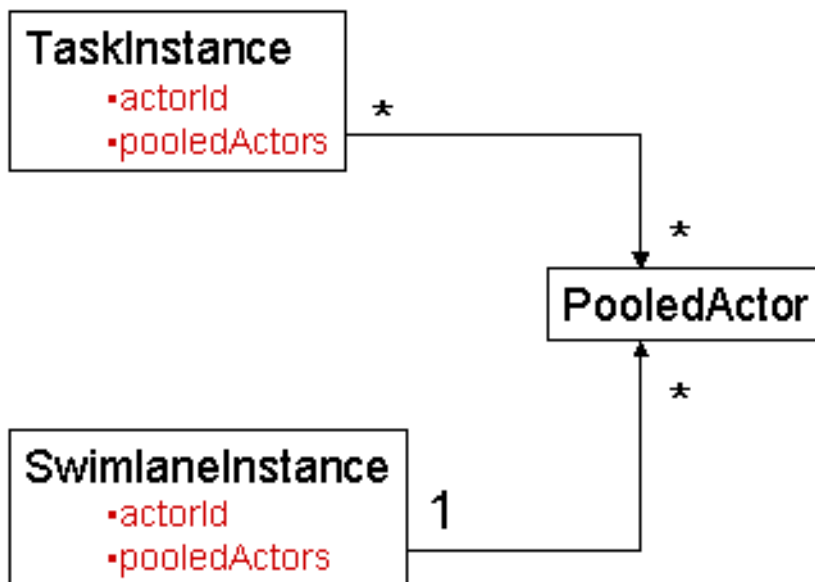


図8.1 割り当てモデルのクラス図

`actorId` はタスクに対応するのに対し、プールされたアクターのセットはこれらの候補の集まりで、タスクを取得した場合その中の1つがそのタスクに対応することになります。`actorId` および `pooledActors` は任意で、組み合わせることも可能です。

8.3.3. パーソナルタスクリスト

パーソナルタスクリストは、特定の個人に割り当てられたすべてのタスクインスタンスのことで、これは、**TaskInstance**の**actorId** プロパティの存在で示します。以下の方法で、**TaskInstance**を誰かのパーソナルタスクリストに置きます。

- **task** 要素の**actor-id** 属性に表現を指定
- コードの任意の場所から **TaskInstance.setActorId(String)** メソッドを使用
- **AssignmentHandler**の **assignable.setActorId(String)** を使用

特定ユーザーのパーソナルタスクリストを取得するに

は、**TaskMgmtSession.findTaskInstances(String actorId)**を使用します。

8.3.4. グループタスクリスト

プールされたアクターはタスクの提供先候補のグループのことで、いずれかの候補がそのタスクを引き受ける必要があります。多くのユーザーが同じタスクを開始するとコンフリクトが発生してしまうため、ユーザーはグループタスクリストのタスクをすぐに開始できません。これを回避するには、ユーザーがまずグループタスクリストのタスクインスタンスのみを取得し自分のパーソナルタスクリストに移動できるようになっています。ユーザーは、自分のパーソナルタスクリストにタスクが置かれた時点のみ、タスクへの作業を開始できます。

誰かのグループタスクリストに **taskInstance** を置くには、ユーザーの**actorId** またはユーザーの**groupIds**の1つを **pooledActorIds** に追加する必要があります。以下のいずれかのメソッドを使い、プールされたアクターを指定します。

- このプロセスの**task**要素の属性**pooled-actor-ids** に表現を指定
- コードの任意の場所から **TaskInstance.setPooledActorIds(String[])** を使用
- **AssignmentHandler**の **assignable.setPooledActorIds(String[])** を使用

あるユーザーのグループタスクリストを取得するにはユーザーの**actorId**とユーザーが属するグループのすべてのIDを含むコレクションを作成します。**TaskMgmtSession.findPooledTaskInstances(String actorId)** または**TaskMgmtSession.findPooledTaskInstances(List actorIds)** を使用してすると、パーソナルタスクリストに存在せず(**actorId==null**)、プール**actorId**にマッチしないタスクインスタンスを検索できます。



注記

当ソフトウェアは、アイデンティティコンポーネントを JBPM タスク割り当てと分けるよう設計されました。JBPM は **actorId** として文字列のみを保管し、ユーザー、グループ、他の ID 情報間の関係は認識しません。

actorId は常にプールアクターよりも優先されます。したがって、**actorId** と **pooledActorId** のリストを持つ **taskInstance** はアクターのパーソナルタスクリストにのみ現れます。また、タスクインスタンスをグループに配置しなおすために**pooledActorId**を保持しますが、これは**taskInstance**の**actorId** プロパティを **null** に設定するだけで行うことができます。

8.4. タスクインスタンス変数

タスクインスタンスは独自の変数セットを持ち、プロセス変数を「参照」できます。また、通常タスクインスタンスは実行パス(トークン)で作成されます。この場合トークン同士の親子関係のように、

トークンとタスクインスタンスの間に親子関係が作成されます。通常のスコープルールが適用される点に注意してください。

コントローラを使い、タスクインスタンススコープとプロセススコープ化した変数の間で、変数の作成、生成、サブミットを行います。

これは、タスクインスタンスが、自身の変数と、関係しているトークンのすべての変数を'見る'ことができるということです。

このコントローラを使い、タスクインスタンススコープとプロセススコープ化した変数の間で、変数の作成、生成、そしてサブミット可能です。

8.5. タスクコントローラ

タスクインスタンス作成時、タスクコントローラを使い、タスクインスタンス変数の生成が可能です。また、タスクインスタンス終了時、タスクコントローラを使いそれに所属するデータをプロセス変数へサブミットできます。

注記

タスクコントローラの利用は任意です。タスクインスタンスは、そのトークンと関係のあるプロセス変数を「参照する」ことができます。タスクコントローラは以下のタスクを実行する際に利用してください。

- 中間のタスクインスタンス変数の更新によりプロセスが完了するまでにプロセス変数へ影響を与えないように、タスクインスタンス変数のコピーを作成します。この時、これらのコピーは再度プロセス変数にサブミットされます。
- タスクインスタンス変数は、プロセス変数に対して1対1の関係を持ちません。例えば、プロセスに**sales in January**、**sales in February**、**sales in March**の変数があれば、タスクインスタンスのフォームは、この3カ月間の平均売上高を表示する必要があるかもしれません。

タスクはユーザからの入力を集めようとします。しかし、タスクをユーザにみせるために利用可能なユーザインターフェースは、たくさんあります。例えば、**web**アプリケーション、**swing**アプリケーション、インスタントメッセージャー、**e**メールフォームなど。そのため、タスクコントローラは、プロセス変数（＝プロセスコンテキスト）とユーザインターフェースの橋渡しをします。タスクコントローラは、ユーザインターフェースアプリケーションにプロセス変数のビューを提供します。

タスクインスタンス作成時にタスクコントローラは、プロセス変数があればそれをタスク変数に変換します。タスク変数は、ユーザインターフェースフォームの入力として取り扱われます。ユーザ入力自体は、タスク変数に保存されます。ユーザがタスクを終了するとき、タスクコントローラは、タスクインスタンスのデータをもとに、プロセス変数の更新を行います。

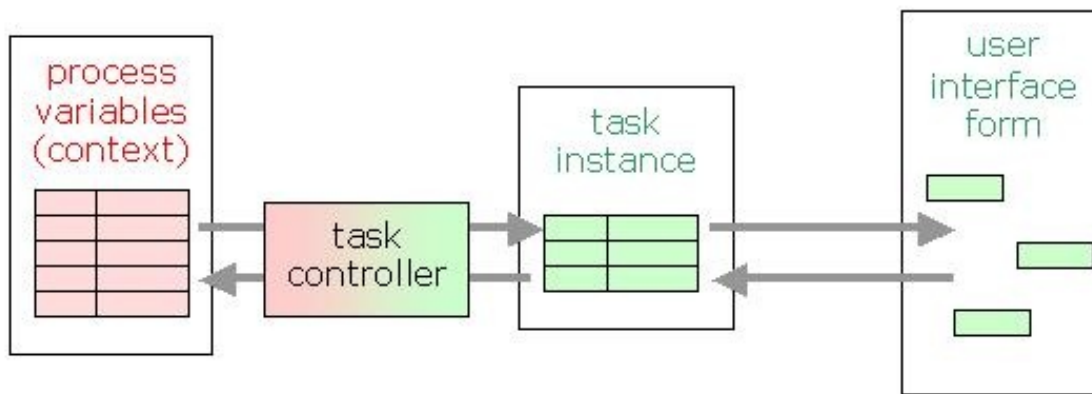


図8.2 タスクコントローラ

簡単なシナリオですと、プロセス変数とフォームパラメーターの間に1対1マッピングがあるとして、**task**要素でタスクコントローラを指定します。この場合、デフォルトのJBPMタスクコントローラを利用でき、タスク変数にプロセス変数をコピーする方法を記述した**variable**要素のリストを取得します。

次の例は、プロセス変数に基づいて、どのように別々のタスクインスタンス変数のコピーを作成するかを示しています。

```
<task name="clean ceiling">
  <controller>
    <variable name="a" access="read" mapped-name="x" />
    <variable name="b" access="read,write,required" mapped-name="y" />
    <variable name="c" access="read,write" />
  </controller>
</task>
```

name属性は、プロセス変数の名前をさしています。**mapped-name**は、任意設定で、タスクインスタンス変数の名前をさしています。**mapped-name**属性が省略された場合、**mapped-name**の名前をデフォルトにします。**mapped-name**は、webアプリケーションのタスクインスタンスにあるフィールドのラベルとしても利用できるということに注意してください。

access 属性を使い、タスクインスタンス作成時に変数をコピーするか、またタスクインスタンス終了時にプロセス変数を書き直すかどうか指定します(この情報を使い、ユーザーインターフェースが正しいフォーム制御を生成できるようにします)。**access** 属性は任意設定であり、デフォルトのアクセス権は **read,write** です。

A **task-node** は多くのタスクを持つことができ、**start-state**は1つのタスクを持つことができます。

プロセス変数とフォームパラメータ間の簡単な1対1のマッピングに制限が過剰となってきた場合、カスタムの **TaskControllerHandler** 実装を作成してください。以下がこのインターフェースです：

```
public interface TaskControllerHandler extends Serializable {
    void initializeTaskVariables(TaskInstance taskInstance, ContextInstance contextInstance, Token token);
    void submitTaskVariables(TaskInstance taskInstance, ContextInstance contextInstance, Token token);
}
```

このコード例はマップの設定方法を表しています。


```
<task name="clean ceiling">
  <controller class="com.yourcom.CleanCeilingTaskControllerHandler">
    -- here goes your task controller handler configuration --
  </controller>
</task>
```

8.6. スイムレーン

スイムレーンはプロセスロールであり、このメカニズムを使い、プロセス内の複数のタスクが同じアクターにより実行するよう指定します。あるスイムレーンに最初のタスクインスタンスが作成された後、同じスイムレーン内の以降の全タスク用にアクターが記憶されます。したがって、スイムレーンには **assignment** が1つ存在します。詳細は「[割り当て](#)」を参照してください。

スイムレーンで最初のタスクが作成されたとき、スイムレーンの **AssignmentHandler** が呼び出されます。**AssignmentHandler** に渡される **Assignable** な項目は、**SwimlaneInstance** になります。特定のスイムレーン内のタスクインスタンスで実行される全 **assignment** は伝播されます。特定のプロセスに関する知識を持つ人がタスクを取得するため、この動作はデフォルトとなっています。そのスイムレーンでそれ以降のタスクインスタンスはすべて、そのユーザーに自動的に割り当てられます。

8.7. 開始タスクのスイムレーン

スイムレーンは開始タスクとスイムレーンを関連付けることができます。このような関連付けを行うことでプロセスのイニシエーターをキャプチャーします。

タスクは、**start-state** で指定でき、スイムレーンと関連付けられます。新しいタスクインスタンスがタスクに割り当てられると、現在の認証されたアクターが **Authentication.getAuthenticatedActorId()** メソッド経由でキャプチャーされ、そのアクターは、開始タスクのスイムレーンに保存されます。

```
<process-definition>
  <swimlane name='initiator' />
  <start-state>
    <task swimlane='initiator' />
    <transition to='...' />
  </start-state>
  ...
</process-definition>
```

また、通常のメソッドを使い開始タスクへ変数を追加することで、このタスクに紐付いたフォームを定義します。「[タスクコントローラ](#)」を参照してください。

8.8. タスクイベント

アクションとタスクを関連付けることができます。標準のイベント型は4つあります。

1. **task-create**、タスクインスタンス作成時にトリガーされます。
2. **task-assign**、タスクインスタンス割り当て時にトリガーされます。このイベント上で実行されるアクションでは、前回のアクターに **executionContext.getTaskInstance().getPreviousActorId()** メソッドでアクセス可能である点に注意してください。

3. **task-start**、**TaskInstance.start()** メソッドが呼び出されるとトリガーされます。このオプション機能を使い、ユーザーが実際にこのタスクインスタンスにて作業を開始していることを示します。
4. **task-end**、**TaskInstance.end(...)** 呼び出されるとトリガーされます。これはタスクの終了をマークします。そのタスクが実行プロセスに関係している場合、この呼び出しは、実行プロセスの再開をトリガーするかもしれません。



注記

例外ハンドラーをタスクと紐付けることも可能です。詳細情報については、「[例外ハンドリング](#)」を参照してください。

8.9. タスクタイマー

タスクにタイマーを指定できます。「[タイマー](#)」を参照してください。

タスクタイマーの **cancel-event** をカスタマイズできることです。デフォルトでは、タスクタイマーは、タスク終了時にキャンセルされます。しかし、タイマー上の **cancel-event** 属性で **task-assign** か **task-start** にそれをカスタマイズすることができます。**cancel-event** は複数のイベントをサポートします。属性にてコンマ区切りのリストで指定することで、**cancel-event** 型を組み合わせます。

8.10. タスクインスタンスのカスタマイズ

タスクインスタンスのカスタマイズは、以下の手順に従います。

1. **TaskInstance** のサブクラスを作成します。
2. **org.jbpm.taskmgmt.TaskInstanceFactory** 実装を作成します。
3. **jbpm.cfg.xml** ファイルの **jbpm.task.instance.factory** 構成プロパティを完全修飾クラス名に設定することで、実装の設定を行います。
4. **TaskInstance** のサブクラスを利用している場合、(**extends="org.jbpm.taskmgmt.exe.TaskInstance"** を使い) そのサブクラスの **Hibernate** マッピングファイルを作成してください。
5. **hibernate.cfg.xml** のリストにマッピングファイルを追加します。

8.11. アイデンティティコンポーネント

ユーザ、グループとパーミッションの管理は、一般的に **アイデンティティ管理** として知られています。JBPM には任意のアイデンティティコンポーネントを含み、企業独自のデータストアと簡単に置き換えることができます。

JBPM アイデンティティ管理コンポーネントは、組織モデルのナレッジが含まれており、これを使いタスクを割り当てます。このモデルはユーザー、グループ、システム、これらの関係を説明します。オプションで権限やロールも含めることができます。

JBPM は、アクターを実際のプロセスへのパーティシパントとして定義してこれに対応しています。アクターは、**actorId** と呼ばれる ID で認識されます。JBPM は、**actorId** についてのナレッジだけを持ち、最大限の柔軟性を実現するため、**java.lang.String** として表現されます。そのため、組織モデルやデータ構造に関するいかなるナレッジも、JBPM コアエンジンの範囲外となります。

JBPM の拡張として、将来的に簡単なユーザーロールモデルを管理するコンポーネントを提供を予定しています。このユーザーとロール間の多対多の関係は、J2EE やサーブレット仕様で定義されているモデルと同じで、新しい開発の出発点となるかもしれません。

ユーザーロールモデルが、サーブレット、ejb、ポートレット仕様で使用されているように、ユーザーロールモデルは、タスク割り当てを行うには、十分強力とはいえないことに注意してください。そのモデルは、ユーザとロールの多対多関連です。これはプロセスに関連したユーザのチームと組織構成についての情報を含んでいません。

8.11.1. アイデンティティモデル

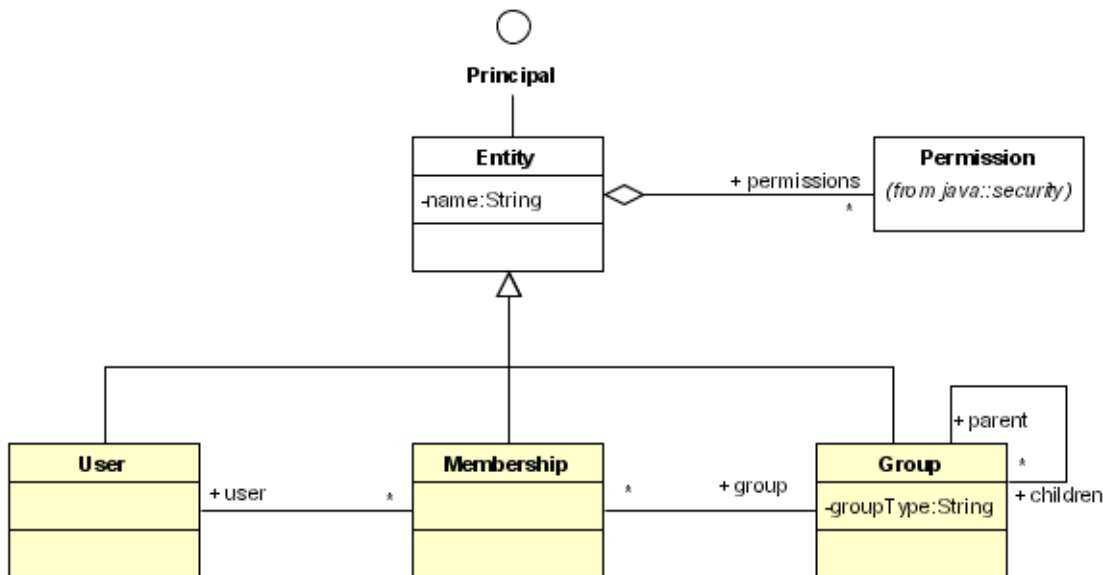


図8.3 アイデンティティモデルクラス図

黄色のクラスは、次に議論する Expression Assignment Handler に関連します。

User は、ユーザかサービスを表します。**Group** は、さまざまなユーザのグループです。**Group** は、チーム、ビジネスユニット、及び会社全体の関係をモデルするようにネストすることが可能です。**Group** は、階層グループを区別するために、タイプを持っています (例：髪の色グループ)。**Membership** は、ユーザとグループ間で多対多関連を表しています。メンバーシップは、会社でのポジションを表すのに利用できます。また、グループ内のユーザが行うロールを示すために、メンバーシップ名を使用することができます。

8.11.2. 割り当て式

アイデンティティコンポーネントは、アクター導出のために、タスクの割り当て中、式を評価する1つの実装を含んでいます。ここに、プロセス定義中での割り当て式の利用方法をお見せします。

```

<process-definition>
  <task-node name='a'>
    <task name='laundry'>
      <assignment expression='previous --> group(hierarchy) -->
member(boss)' />
    </task>
    <transition to='b' />
  </task-node>

```

<para>Syntax of the assignment expression is like this:</para>

```

    first-term --> next-term --> next-term --> ... --> next-term

where

first-term ::= previous |
              swimlane(swimlane-name) |
              variable(variable-name) |
              user(user-name) |
              group(group-name)

and

next-term ::= group(group-type) |
              member(role-name)
</programlisting>

```

8.11.2.1. 最初の条件 (First term)

表現は左から右に解決されていきます。**First-term**(最初の条件)は、アイデンティティモデルに **User** か **Group** を指定します。その後の条件は中間ユーザかグループから次の条件を算出します。

previousは、タスクが、現在の承認済みアクターに割り当てられていることを意味します。これは、プロセス中で前回のステップで動作したアクターということになります。

swimlane(swimlane-name)は、ユーザやグループが、指定されたスイムレーンインスタンスから取得されるということです。

variable(variable-name)は、ユーザやグループが指定された変数インスタンスから取得されるということです。変数インスタンスには、**java.lang.String** を含めることができます。その場合、そのユーザやグループがアイデンティティコンポーネントから取得されます。もしくは、変数インスタンスは、**User**や**Group**のオブジェクトを含みます。

user(user-name) は、特定のユーザがアイデンティティコンポーネントから取得されるということです。

group(group-name) は特定のグループが、アイデンティティコンポーネントから取得されるということです。

8.11.2.2. 次の条件(Next term)

group(group-type) は、ユーザーのグループを取得します。前回の条件は、**User**でなければなりません。そのユーザーの全メンバーシップの中で指定の**group-type** を用いてグループの検索を行います。

member(role-name) はグループ内で指定のロールを実行するユーザーを取得します。前回の条件は **Group**でなければなりません。この条件では、当グループへのメンバーシップを持つユーザーのうち、メンバーシップ名が指定の **role-name**に一致するものを検索します。

8.11.3. アイデンティティコンポーネントの削除

会社のユーザデータベースや LDAP システムなど組織情報の独自のデータソースを使用したい場合、JBPM アイデンティティコンポーネントを削除することができます。**hibernate.cfg.xml** より次の行を削除するだけで JBPM アイデンティティコンポーネントを削除することができます。

```
<mapping resource="org/jbpm/identity/User.hbm.xml"/>
<mapping resource="org/jbpm/identity/Group.hbm.xml"/>
<mapping resource="org/jbpm/identity/Membership.hbm.xml"/>
```

ExpressionAssignmentHandlerが、アイデンティティコンポーネントに依存しているためそのまま利用することはできません。 **ExpressionAssignmentHandler**を再利用して、利用中のユーザーデータストアにバインドしたい場合、 **ExpressionAssignmentHandler**を拡張して、**getExpressionSession**メソッドをオーバーライドできます。

```
protected ExpressionSession getExpressionSession(AssignmentContext
assignmentContext);
```

第9章 スケジューラー

本章を読んで、ビジネスプロセスマネージャのタイマーの役割について学びましょう。

タイマーはプロセスのイベントで作成することができます。タイマーを設定して、アクションの実行やイベントの遷移をトリガします。

9.1. タイマー

タイマーを指定する最も簡単な方法は、**タイマー要素**をノードに追加することです。追加する方法は次のサンプルコードを見てください。

```
<state name='catch crooks'>
  <timer name='reminder'
    duedate='3 business hours'
    repeat='10 business minutes'
    transition='time-out-transition' >
    <action class='the-remainder-action-class-name' />
  </timer>
  <transition name='time-out-transition' to='...' />
</state>
```

ノードに指定されたタイマーは、ノードの終了後には実行されません。遷移とアクションは任意で指定できます。タイマーが実行されると、次のイベントが順次発生します。

1. **timer** タイプのイベントが発生します。
2. アクションの指定がある場合、アクションが実行されます。
3. 指定された遷移でシグナルが実行を再開します。

すべてのタイマーには固有の名前が付けられなければなりません。**timer** 要素に名前が指定されていない場合、デフォルトでノード名がタイマー名になります。

タイマーアクションを使用してすべてのアクションエレメントをサポートします (**action** や **script** など)。

タイマーはアクションにより作成、キャンセルされます。2つの関連する **action-elements** は **create-timer** と **cancel-timer** です。前述のタイマー要素は、**node-enter** 上の **create-timer** アクションと **node-leave** 上の **cancel-timer** アクションを略したものになります。

9.2. スケジューラーデプロイメント

プロセス実行はタイマーを作成、キャンセルしタイマーストアに保存します。別の **timer runner** がこのストアをチェックし、各タイマーを実行します。

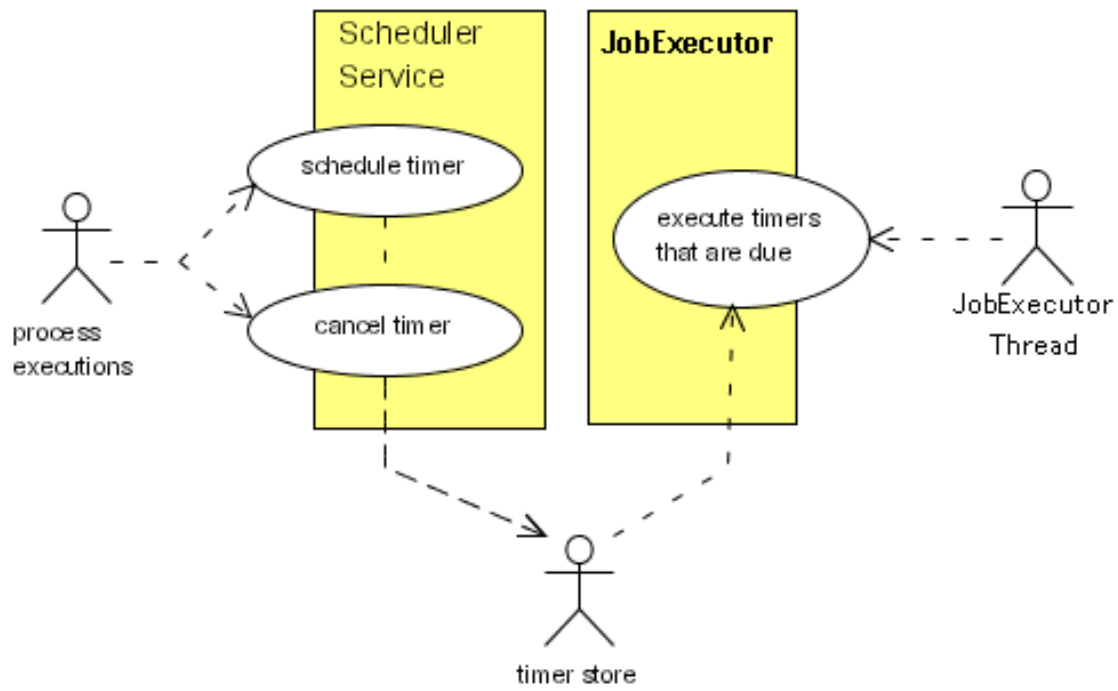


図9.1 スケジューラーコンポーネントの概要

第10章 非同期の続行

10.1. コンセプト

JBPM は *グラフ指向プログラミング(GOP)* に基づいています。基本的に、**GOP** は並列実行パスを扱える単純な状態マシンを指定できます。しかし、**GOP** で指定されている実行アルゴリズムでは、すべての状態遷移は、クライアントのスレッドでひとつの操作によって行われます。デフォルトでは、クライアントのスレッドにおいて状態遷移を行うことは、サーバー側のトランザクションに自然に適合するため、良いアプローチとなります。プロセス実行は、別の待機状態にひとつのトランザクション内で移動します。

状況によっては、開発者がプロセス定義のトランザクション境界を微調整したいことがあります。

jPDL では、プロセス実行が属性 **async="true"** で非同期に続行するよう指定できます。

async="true" は、イベントでトリガーされ、すべてのノードタイプとすべてのアクションタイプで指定できる場合のみサポートされます。

10.2. 例

通常、トークンがノードに入場した後に、そのノードが常に実行されます。そのため、ノードはクライアントのスレッドで実行されます。2つのサンプルを使用して非同期な続行を詳しく見ていきます。最初のサンプルは、3つのノードを持つプロセスの一部です。'a' ノードは待機状態で、'b' ノードは、自動ステップ、'c' ノードも待機状態です。下図の通り、このプロセスには非同期の動作がありません。

最初のフレームは、開始状況を示します。トークンは'a'ノードを参照します。つまり、実行パスは外部からのトリガーを待っています。そのトリガーは、シグナルをトークンに送ることによって起こるものでなければなりません。シグナルが到着時、トークンは'a'ノードから遷移越しに'b'ノードに渡されます。トークンが'b'ノードに到着後、'b'ノードは実行されます。'b'ノードは待機状態として振る舞わない自動ステップ(例えばeメール送信)だということを思い出してください。2個目のフレームは'b'ノードが実行されているときのスナップショットです。'b'ノードはプロセスでは自動ステップなので、'b'ノードの実行は、'c'ノードへ遷移するトークンの伝播も含みます。'c'ノードは待機状態なので、3個目のフレームはsignalメソッドから戻った後の最後の状況を示しています。

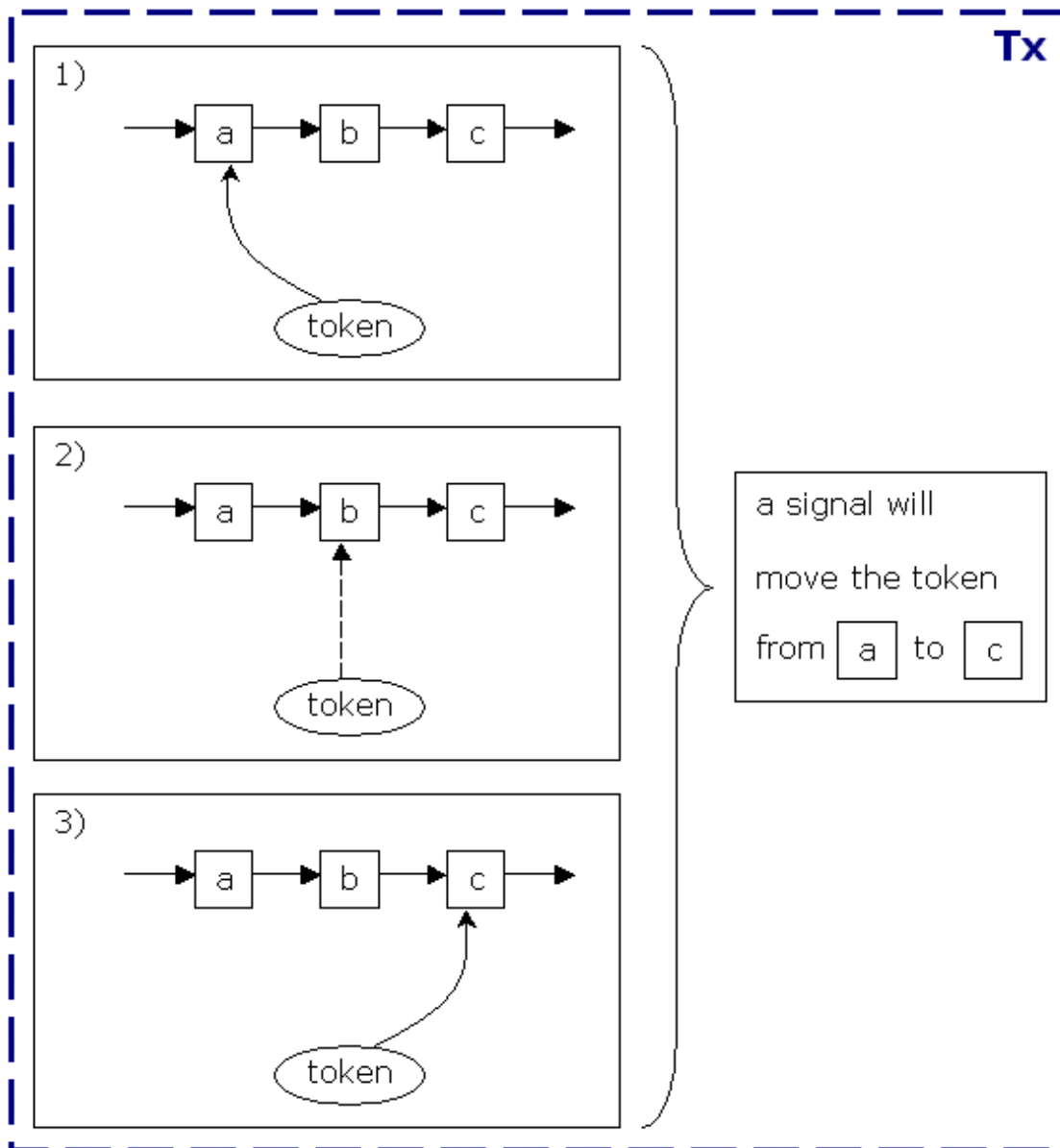


図10.1 サンプル 1: 非同期の続行がないプロセス

JBPM では「永続性」は必須ではありませんが、最も一般的にシグナルはトランザクション内で呼び出されます。トランザクションの更新を見てみましょう。最初に、トークンは 'c' ノードを示すよう更新されます。これらの更新は、JDBC 接続上の **GraphSession.saveProcessInstance** の呼び出しにより、**Hibernate** によって生成されます。2番目に自動化アクションがトランザクションリソースにアクセスし更新する場合、このような更新は同じトランザクションと組み合わせられるか同じトランザクションの一部となるはずで

2つ目のサンプルは、最初のサンプルが変形したもので、'b' ノードに非同期の続行を導入します。'a' ノードと 'c' ノードは、最初のサンプルと同様に待機状態として動作します。JPDLでは、**async="true"** 属性を設定するとノードが非同期としてマークされます。

async="true" を 'b' ノードに追加した結果、プロセス実行は、2つに分かれます。最初の部分は、'b' ノードが実行されるまで、実行していきます。2つ目の部分は、'b' ノードを実行し、'c' ノードの待機状態で止まります。

トランザクションもゆえに2つに分かれます。それぞれにひとつのトランザクションがあります。JBPMは、最初のトランザクションでは、'a' ノードを退場するのに、外部からのトリガー (**Token.signal** メソッドの呼び出し) を求めますが、2つ目のトランザクションは、自動的に呼び出されて実行されます。

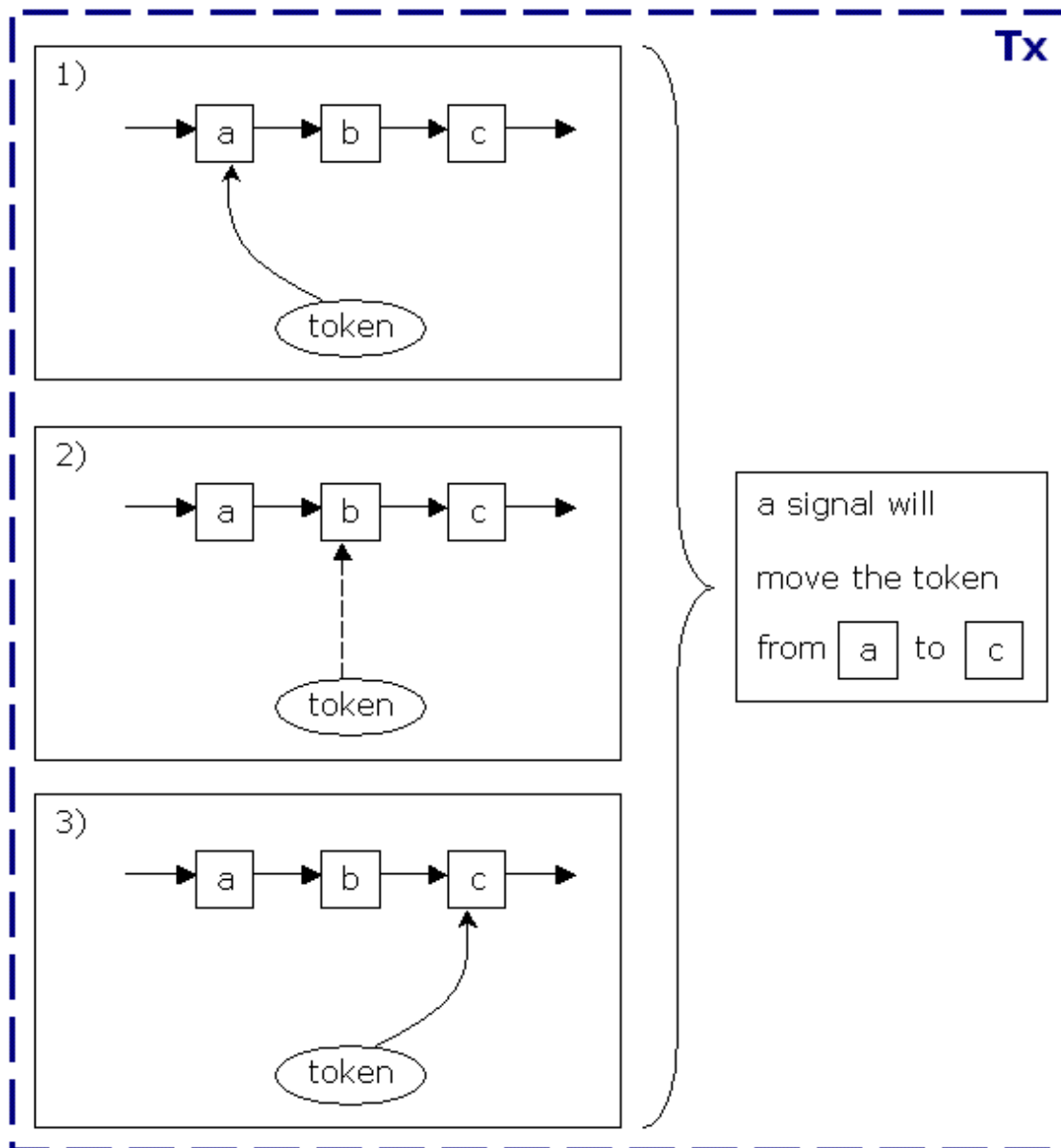


図10.2 非同期続行のプロセス

アクションも原則は似ています。属性 **async="true"** でマークされたアクションは、プロセス実行のスレッド外で実行されます。永続化の設定がされている場合（デフォルトです）、アクションは別々のトランザクションで実行されます。

JBPMでは、非同期続行は非同期メッセージングシステムの利用によって実現されます。プロセス実行が非同期で実行されるところに到着したときに、JBPMは、そのプロセスの中止、メッセージコマンドの作成、そして、コマンドエグゼキューターにメッセージコマンドを送ります。コマンドエグゼキューターはメッセージを受け取り次第、中断したプロセス実行を再開できる依存のないコンポーネントです。

JBPMは、JMSプロバイダー、もしくは組み込み非同期メッセージングシステムの利用を設定できます。組み込み非同期メッセージングシステムは、機能的に制限がありますが、JMSが利用できない環境でもこの機能をサポートします。

10.3. ジョブエグゼキューター

ジョブエグゼキューターは、プロセス実行を非同期に再開するコンポーネントです。非同期メッセージシステムを介してコマンドメッセージが到着するまで待機し、実行します。非同期続行に使用される2つのジョブは **ExecuteNodeJob** と **ExecuteActionJob** です。

これらのジョブメッセージは、プロセス実行により生成されます。プロセス実行中に、非同期で実行する必要がある各ノードまたはアクションに対して **Job (POJO)** が **MessageService** へ送信されます。メッセージサービスは **JbpmContext** と関連付けられ、送信する必要があるすべてのメッセージを収集します。

メッセージは **JbpmContext.close()** の一部として送信されます。このメソッドは関連付けられたすべてのサービスに対して **close()** 呼び出しをカスケード処理します。実際のサービスは **jbpm.cfg.xml** で設定できます。サービスの1つ **DbMessageService** はデフォルトで設定され、新しいジョブメッセージが利用可能であることをジョブエグゼキュータに通知します。

グラフ実行メカニズムは **MessageServiceFactory** インターフェースと **MessageService** インターフェースを使用してメッセージを送信します。これにより非同期メッセージサービスが設定可能になります (**jbpm.cfg.xml**)。Java EE 環境では、**DbMessageService** を **JmsMessageService** に置き換えて、アプリケーションサーバーの機能を活用できます。

ジョブエグゼキュータの挙動に関する要約は次の通りです。

ジョブはデータベースのレコードです。ジョブはオブジェクトであり、実行可能です。タイマーと非同期メッセージはジョブになります。非同期メッセージの場合、メッセージが挿入された時に **dueDate** が現在の時間に設定されます。ジョブエグゼキュータはジョブを実行する必要がありますが、次の2つの段階で実行されます。

- ジョブエグゼキュータスレッドはジョブを取得する必要がある
- ジョブエグゼキュータスレッドはジョブを取得する必要がある

ジョブの取得と実行は、別のトランザクションとして実行されます。ディスパッチャースレッドは、このノードの全エグゼキュータスレッドの代わりにデータベースからジョブを取得します。エグゼキュータスレッドがジョブを取得すると、その名前をジョブの所有者フィールドに追加します。各スレッドには、IP アドレスやシーケンス番号に従い一意名が設定されています。

スレッドはジョブの取得と実行の間に消失する可能性があります。このような状況をクリーンアップするため、1つのジョブエグゼキュータごとにロック時間をチェックする1つのロックモニタスレッドが存在します。30分以上ロックされているジョブは、他のジョブによって実行されるようロックモニタスレッドによってロック解除されます。

Hibernate のオプティミスティックロックが正常に動作するために必要な分離レベルは **REPEATABLE_READ** にセットする必要があります。 **REPEATABLE_READ** により、このクエリが1つの競合トランザクションで1つの行のみを更新することが保証されます。

```
update JBPM_JOB job
set job.version = 2
    job.lockOwner = '192.168.1.3:2'
where
    job.version = 1
```

Non-Repeatable Reads とは、トランザクションが以前に読み取ったデータを再び読み取り、トランザクションの以前の読み取り移行にコミットされた他のトランザクションによってデータが変更されていることを発見する特殊な状態を示します。

Non-Repeatable Reads は楽観ロックにとって問題です。分離レベル **READ_COMMITTED** は **Non-Repeatable Reads** の発生を許すため、十分ではありません。したがって、複数のジョブエグゼキュータスレッドを設定する場合は **REPEATABLE_READ** が必要です。

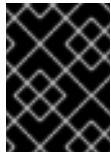
ジョブエグゼキュータ関連の設定プロパティは以下の通りです。

jbpmConfiguration

設定をリトリブする bean

name

ジョブエグゼキューター名



重要

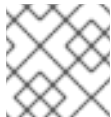
1 台のマシンに JBPM インスタンスが 1 つ以上、開始されている場合、この名前は各ノードに一意でなければなりません。

nbrOfThreads

開始されたエグゼキュータースレッドの数

idleInterval

保留されているジョブがない場合、ディスパッチャースレッドがジョブのキューを確認するまでに待機する時間

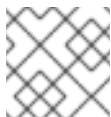


注記

キューにジョブが追加されると、ディスパッチャースレッドに自動通知されます。

retryInterval

実行時に問題があった場合のジョブ再試行の間隔。デフォルト値は 3 回です。



注記

再試行の最大回数は `jbpm.job.retries` で設定されています。

maxIdleInterval

`idleInterval` の最大期間

historyMaxSize

このプロパティは廃止されるため、特に影響はありません。

maxLockTime

`lock-monitor` スレッドがアンロックするまでジョブをロックできる最大時間

lockMonitorInterval

`lock-monitor` スレッドがロックされているジョブの確認をする際の休止間隔

lockBufferTime

このプロパティは廃止されるため、特に影響はありません。

10.4. JBPM 組み込み非同期メッセージング

JBPM の組み込み非同期メッセージを使用する場合、ジョブメッセージはデータベースに永続化することによって送信されます。このメッセージの永続化は、JBPM プロセス更新と同じトランザクションまたは JDBC 接続で実行できます。

そのジョブのメッセージは、**JBPM_JOB** テーブルに保存されます。

POJO コマンドエグゼキューター (**org.jbpm.msg.command.CommandExecutor**) は、データベースのテーブルからメッセージを読み取り、実行します。つまり、典型的な POJO コマンドエグゼキューターのトランザクションは以下のようになります。

1. 次のコマンドメッセージの読み込み
2. コマンドメッセージの実行
3. コマンドメッセージの削除

コマンドメッセージの実行処理が失敗した場合、トランザクションはロールバックされます。その後、新しいトランザクションは、開始され、エラーメッセージをデータベースのメッセージに追加します。コマンドエグゼキューターは、例外を含むすべてのメッセージを無視します。

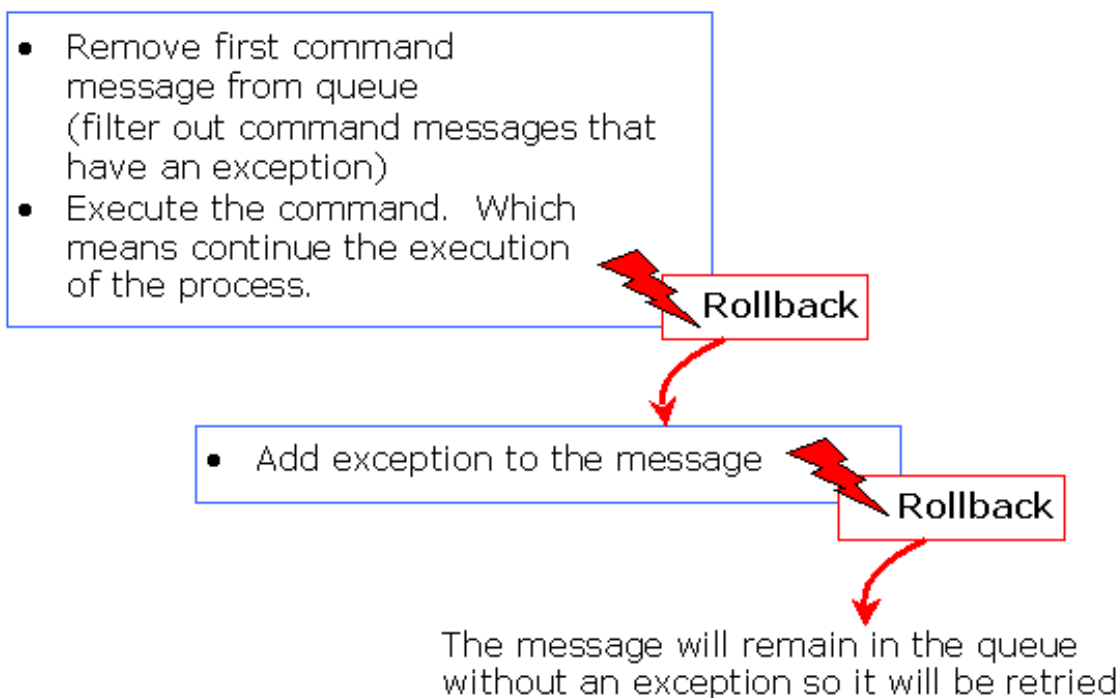


図10.3 POJO コマンドエグゼキュータートランザクション

例外をコマンドメッセージに追加するトランザクションが失敗した場合は、ロールバックされます。その場合、例外なしでメッセージはキューに残り、後で再試行されます。



重要

JBPM のビルトイン非同期メッセージングシステムはマルチノードロックをサポートしていません。そのため、POJO コマンドエグゼキューターを複数回デプロイすることはできず、同じデータベースを使用するよう設定することもできません。

第11章 ビジネスカレンダー

本章では、作業の期限やタイマーの計算に使用するビジネスプロセスマネージャのカレンダー機能について説明します。

ビジネスカレンダーは、基準日 (**base date**) に期間 (**duration**) を追加したり差し引いたりすることで計算を行います。基準日の指定がない場合は、デフォルトで現在の日付が使用されます。

11.1. DUE DATE (期限)

期限は期間と基準日によって構成されます。使用される公式は **duedate ::= [<basedate> +/-] <duration>** になります。

11.1.1. Duration (期間)

期間は、**duration ::= <quantity> [business] <unit>** を公式として使用し、絶対時間または営業時間のいずれかで指定されます。

上記の計算では、**<quantity>** は **Double.parseDouble(quantity)** で解析可能なテキストでなければなりません。**<unit>** は、**second**、**seconds**、**minute**、**minutes**、**hour**、**hours**、**day**、**days**、**week**、**weeks**、**month**、**months**、**year**、**years** のいずれかになります。任意の **business** フラグを追加すると、営業時間のみが期間で考慮されます。**business** を指定しないと、期間は絶対期間として解釈されます。

11.1.2. 基準日

基準日は、**basedate ::= <EL>** のように計算されます。

上記の公式では、**<EL>** は **Java Date** または **Calendar** オブジェクトへ解決する Java 表現言語になります。



警告

JbpmException エラーが発生するため、他のオブジェクトタイプの変数を参照しないようにしてください。

基準日は、簡単なタイマーの **duedate** 属性やタスクリマインダ、タスク内のタイマーなど複数の場所でサポートされますが、これらの要素の **repeat** 属性ではサポートされません。

11.1.3. 期限の例

以下は可能な使用例です。

```
<timer name="daysBeforeHoliday" duedate="5 business days">...</timer>
<timer name="pensionDate" duedate="#{dateOfBirth} + 65 years" >...</timer>
<timer name="pensionReminder" duedate="#{dateOfPension} - 1 year"
>...</timer>
<timer name="fireWorks" duedate="#{chineseNewYear} repeat="1 year"
```

```
>...</timer>
<reminder name="hitBoss" duedate="#{payRaiseDay} + 3 days" repeat="1 week"
/>
```

11.2. カレンダー設定

org/jbpm/calendar/jbpm.business.calendar.properties ファイルに営業時間を指定します (この設定ファイルをカスタマイズするには、変更したコピーをクラスパスのルートに置きます)。

次は **jbpm.business.calendar.properties** にあるデフォルトの営業時間の仕様になります。

```
hour.format=HH:mm
#weekday ::= [<daypart> [& <daypart>]*]
#daypart ::= <start-hour>-<to-hour>
#start-hour and to-hour must be in the hour.format
#dayparts have to be ordered
weekday.monday=    9:00-12:00 & 12:30-17:00
weekday.tuesday=   9:00-12:00 & 12:30-17:00
weekday.wednesday= 9:00-12:00 & 12:30-17:00
weekday.thursday=  9:00-12:00 & 12:30-17:00
weekday.friday=    9:00-12:00 & 12:30-17:00
weekday.saturday=
weekday.sunday=

day.format=dd/MM/yyyy
# holiday syntax: <holiday>
# holiday period syntax: <start-day>-<end-day>
# below are the belgian official holidays
holiday.1= 01/01/2005 # nieuwjaar
holiday.2= 27/3/2005  # pasen
holiday.3= 28/3/2005  # paasmaandag
holiday.4= 1/5/2005   # feest van de arbeid
holiday.5= 5/5/2005   # hemelvaart
holiday.6= 15/5/2005  # pinksteren
holiday.7= 16/5/2005  # pinkstermaandag
holiday.8= 21/7/2005  # my birthday
holiday.9= 15/8/2005  # moederkesdag
holiday.10= 1/11/2005 # allerheiligen
holiday.11= 11/11/2005 # wapenstilstand
holiday.12= 25/12/2005 # kerstmis

business.day.expressed.in.hours=      8
business.week.expressed.in.hours=    40
business.month.expressed.in.business.days= 21
business.year.expressed.in.business.days= 220
```

11.3. 使用例

異なる使用例は次の通りです。

```
<timer name="daysBeforeHoliday" duedate="5 business days">...</timer>

<timer name="pensionDate" duedate="#{dateOfBirth} + 65 years" >...</timer>
```

```

<timer name="pensionReminder" dueDate="#{dateOfPension} - 1 year" >...
</timer>

<timer name="fireWorks" dueDate="#{chineseNewYear} repeat="1 year" >...
</timer>

<reminder name="hitBoss" dueDate="#{payRaiseDay} + 3 days" repeat="1 week"
/>

```

```

hour.format=HH:mm
#weekday ::= [<daypart> [& <daypart>]*]
#daypart ::= <start-hour>-<to-hour>
#start-hour and to-hour must be in the hour.format
#dayparts have to be ordered
weekday.monday=    9:00-12:00 & 12:30-17:00
weekday.tuesday=   9:00-12:00 & 12:30-17:00
weekday.wednesday= 9:00-12:00 & 12:30-17:00
weekday.thursday=  9:00-12:00 & 12:30-17:00
weekday.friday=    9:00-12:00 & 12:30-17:00
weekday.saturday=
weekday.sunday=

day.format=dd/MM/yyyy
# holiday syntax: <holiday>
# holiday period syntax: <start-day>-<end-day>
# below are the belgian official holidays
holiday.1= 01/01/2005 # nieuwjaar
holiday.2= 27/3/2005  # pasen
holiday.3= 28/3/2005  # paasmaandag
holiday.4= 1/5/2005   # feest van de arbeid
holiday.5= 5/5/2005   # hemelvaart
holiday.6= 15/5/2005  # pinksteren
holiday.7= 16/5/2005  # pinkstermaandag
holiday.8= 21/7/2005  # my birthday
holiday.9= 15/8/2005  # moederkesdag
holiday.10= 1/11/2005 # allerheiligen
holiday.11= 11/11/2005 # wapenstilstand
holiday.12= 25/12/2005 # kerstmis

business.day.expressed.in.hours=      8
business.week.expressed.in.hours=     40
business.month.expressed.in.business.days= 21
business.year.expressed.in.business.days= 220

```

本章をお読みいただいたため、ビジネスカレンダーの仕組みをご理解いただけたと思います。

第12章 電子メールサポート

本章では、カスタマイズなしに JPD L で利用可能な 電子メールサポートについて説明します。本章を読んで、様々なメール機能の設定する方法について学びましょう。

12.1. JPD L でのメール

プロセスから電子メールが送信される時点を指定する方法は 4 つあります。各方法について順番に説明しましょう。

12.1.1. メールアクション

プロセスグラフで電子メールの送信をノードとして表示しない場合、メールアクションを使用します。



注記

メールアクションは通常のアクションを追加できるプロセスの場所ならどこにでも追加することができます。

```
<mail actors="#{president}" subject="readmylips" text="nomoretaxes" />
```

次のように、サブジェクト属性とテキスト属性を要素として指定できます。

```
<mail actors="#{president}" >
  <subject>readmylips</subject>
  <text>nomoretaxes</text>
</mail>
```

各フィールドには JSF のような表現を指定できます。

```
<mail
  to='#{initiator}'
  subject='websale'
  text='your websale of #{quantity} #{item} was approved' />
```

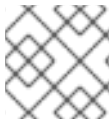


注記

表現に関する詳細は、[「式」](#) を参照ください。

`actors` と `to` の 2 つの属性が受信者を指定します。 `to` 属性はセミコロンで区切られた電子メールアドレスのリストに解決する必要があります。 `actors` 属性はセミコロンで区切られた `actorlds` のリストに解決する必要があります。これらの `actorlds` は電子メールアドレスに解決されます (詳細は [「アドレス解決」](#) を参照)。

```
<mail
  to='admin@mycompany.com'
  subject='urgent'
  text='the mailserver is down :-)' />
```

**注記**

受信者の指定方法については、「[メール受信者の指定](#)」を参照してください。

テンプレートを使用して電子メールを定義することができます。次のようにテンプレートプロパティを上書きします。

```
<mail template='sillystatement' actors="#{president}" />
```

**注記**

テンプレートの詳細については、「[電子メールテンプレート](#)」を参照してください。

12.1.2. メールノード

メールアクションと同様に、電子メール送信のアクションをノードとしてもモデリングすることができます。この場合、ランタイムの挙動は同じですが、プロセスグラフで電子メールがノードとして表示されます。

メールノードは **mail action** と全く同じ属性と要素をエレメントをサポートします (詳細は「[メールアクション](#)」を参照してください)。

```
<mail-node name="send email"
  to="#{president}"
  subject="readmylips"
  text="nomoretaxes">
  <transition to="the next node" />
</mail-node>
```

**重要**

メールノードが1つの退場遷移のみを持つようにしてください。

12.1.3. タスクが割り当てられた電子メール

タスクがアクターに割り当てられた時に通知電子メールを送信できます。以下のようにタスクに対して **notify="yes"** 属性を使用してください。

```
<task-node name='a'>
  <task name='laundry' swimlane="grandma" notify='yes' />
  <transition to='b' />
</task-node>
```

notify を **yes**、**true**、**on** のいずれかに設定して、タスクに割り当てられたアクターへビジネスプロセスマネージャーが電子メールを送信するようにします (電子メールはテンプレートを基とし、Web アプリケーションのタスクページへのリンクが含まれています)。

12.1.4. タスクリマインダ電子メール

電子メールをタスクリマインダーとして送信できます。JPDL の **reminder** 要素はタイマーを利用します。最も一般的な属性は **duedate** と **repeat** です。アクションを指定する必要はありません。

-

```
<task-node name='a'>
  <task name='laundry' swimlane="grandma" notify='yes'>
    <reminder duedate="2 business days" repeat="2 business hours"/>
  </task>
  <transition to='b' />
</task-node>
```

12.2. メールにおける表現

フィールド **to**、**recipients**、**subject**、**text** には、JSF に似た表現を指定することができます。(表現についての詳細は「[式](#)」を参照してください)。

変数 **swimlanes**、**process variables**、**transient variables** **beans** を表現に使用することができます。これらの変数は **jbpm.cfg.xml** ファイルより設定します。

表現は [アドレス解決機能](#) で組み合わせることができます (詳細は「[アドレス解決](#)」を参照してください)。

次の例は、**president** と呼ばれる **swimlane** が存在することを仮定しています。

```
<mail actors="#{president}"
      subject="readmylips"
      text="nomoretaxes" />
```

このコードは、特定のプロセス実行で **president** となる人へ電子メールを送信します。

12.3. メール受信者の指定

12.3.1. 複数の受信者

actors フィールドと **to** フィールドに複数の受信者を記載することができます。コロンかセミコロンで区切ってリストを作成します。

12.3.2. BCC アドレスへ電子メールを送信

ブラインドカーボンコピー(BCC) 受信者へメッセージを送信するには、プロセス定義で **bccActors** 属性か **bcc** 属性を使用します。

```
<mail to='#{initiator}'
      bcc='bcc@mycompany.com'
      subject='websale'
      text='your websale of #{quantity} #{item} was approved' />
```

BCC のメッセージを **jbpm.cfg.xml** で集中的に設定された場所へ送信する方法もあります。次の例を参考にしてみてください。

```
<jbpm-configuration>
  ...
  <string name="jbpm.mail.bcc.address" value="bcc@mycompany.com" />
</jbpm-configuration>
```

12.3.3. アドレス解決

ビジネスプロセスマネージャーではアクターは **actorIds** によって参照されます。これはプロセス参加者を識別する文字列です。アドレスリゾルバーは **actorIds** を電子メールアドレスに変換します。

アドレス解決を適用する場合は属性 **actors** を使用します。アドレスを直接追加する場合は、アドレス解決が適用されないため **to** 属性を使用します。

アドレスリゾルバが次のインターフェースを実装するようにしてください。

```
public interface AddressResolver extends Serializable {
    Object resolveAddress(String actorId);
}
```

アドレスリゾルバは、文字列、文字列のコレクション、文字列の配列のいずれかを返します (文字列は常に指定された **actorId** の電子メールアドレスを表します)。

アドレスリゾルバ実装が **Bean** であるようにしてください。この **Bean** は、次の例のように **jbpm.mail.address.resolver** という名前です。この名前を **jbpm.cfg.xml** ファイルに設定されなければなりません。

```
<jbpm-configuration>
  <bean name='jbpm.mail.address.resolver'
        class='org.jbpm.identity.mail.IdentityAddressResolver'
        singleton='true' />
</jbpm-configuration>
```

ビジネスプロセスマネージャーの **identity** コンポーネントにはアドレスリゾルバーが含まれています。このアドレスリゾルバーは指定 **actorId** のユーザーを検索します。ユーザーが存在する場合、ユーザーの電子メールアドレスが返されます。存在しない場合は **null** が返されます。



注記

アイデンティティコンポーネントの詳細は、「[アイデンティティコンポーネント](#)」を参照してください。

12.4. 電子メールテンプレート

processdefinition.xml ファイルを使用して電子メールを指定する代わりにテンプレートを使用することもできます。この場合でも、各フィールドは **processdefinition.xml** によって上書きされます。次のようにテンプレートを指定します。

```
<mail-templates>
  <variable name="BaseTaskListURL"
    value="http://localhost:8080/jbpm/task?id=" />

  <mail-template name='task-assign'>
    <actors>#{taskInstance.actorId}</actors>
    <subject>Task '#{taskInstance.name}'</subject>
    <text><![CDATA[Hi,
Task '#{taskInstance.name}' has been assigned to you.
Go for it: #{BaseTaskListURL}#{taskInstance.id}
Thanks.
---powered by JBoss jBPM---]]></text>
```

```

    </mail-template>

    <mail-template name='task-reminder'>
      <actors>#{taskInstance.actorId}</actors>
      <subject>Task '#{taskInstance.name}' !</subject>
      <text><![CDATA[Hey,
Don't forget about #{BaseTaskListURL}#{taskInstance.id}
Get going !
---powered by JBoss jBPM---]]></text>
    </mail-template>

  </mail-templates>

```

上記の通り、追加の変数をメールテンプレートに定義することができ、表現で使用できます。

次のように **jbpm.cfg.xml** ファイルよりテンプレートを含むリソースを設定します。

```

<jbpm-configuration>
  <string name="resource.mail.templates" value="jbpm.mail.templates.xml"
/>
</jbpm-configuration>

```

12.5. メールサーバーの設定

次のサンプルコードの通り、**jbpm.cfg.xml** ファイルの **jbpm.mail.smtp.host** プロパティを設定してメールサーバーを設定します。

```

<jbpm-configuration>
  <string name="jbpm.mail.smtp.host" value="localhost" />
</jbpm-configuration>

```

複数のプロパティを指定する必要がある場合は、以下のようにプロパティファイルへのリソース参照を指定します。

```

<jbpm-configuration>
  <string name='resource.mail.properties' value='jbpm.mail.properties' />
</jbpm-configuration>

```

12.6. 送信者アドレスの設定

From アドレスフィールドのデフォルト値は **jbpm@noreply** です。次のように、キー **jbpm.mail.from.address** を使って **jbpm.xfg.xml** ファイルで設定します。

```

<jbpm-configuration>
  <string name='jbpm.mail.from.address' value='jbpm@yourcompany.com' />
</jbpm-configuration>

```

12.7. 電子メールサポートのカスタマイズ

ビジネスプロセスマネージャのすべてのメールサポートは1つのクラス **org.jbpm.mail.Mail** に中央化されています。このクラスは **ActionHandler** 実装です。電子メールが **process XML** に指定されると、**mail** クラスに委譲されます。**mail** クラスから継承し、必要に応じて動作をカスタマイズす

ることができます。メール委譲に使用するクラスを設定するには、以下のように `jbpm.cfg.xml` で `jbpm.mail.class.name` 設定文字列を指定します。

```
<jbpm-configuration>
  <string name='jbpm.mail.class.name'
    value='com.your.specific.CustomMail' />
</jbpm-configuration>
```

カスタマイズされたメールクラスは解析中に読み取られます。アクションは設定された(またはデフォルトの)メールクラス名を参照するプロセスで設定されます。したがって、プロパティを変更した場合でも、既にデプロイされたすべてのプロセスは引き続き古いメールクラス名を参照します。変更するには、JBPM データベース宛の更新ステートメントを送信するだけです。

本章では、様々な電子メール設定の設定方法について詳しく説明しました。例を入念に学習すれば、ご自分の環境で設定を練習できるようになるはずです。

第13章 ロギング

本章を読んで、ビジネスプロセスマネージャが提供するロギング機能やロギング機能を活用できる様々な方法について学びましょう。

ロギングの目的は、プロセス実行の履歴を記録することです。各プロセス実行のランタイムデータが変更されると、変更はログに保存されます。



注記

本章で説明するプロセスロギングとソフトウェアロギングを混同しないようにしてください。ソフトウェアロギングは、ソフトウェアプログラムの実行を追跡します(通常、デバッグが目的です)。反対にプロセスロギングは、プロセスインスタンスの実行の追跡をします。

プロセスロギング情報を利用する多くの方法があります。最も明確な方法は、プロセス実行の参加者によるプロセス履歴のコンサルティングです。

他のユースケースとして、ビジネスアクティビティ監視(Business Activity Monitoring: BAM)があります。BAMは、ビジネスプロセスに関して役に立つ統計情報を見つけるために、プロセス実行ログのクエリや分析を行います。この情報は「真の」ビジネスプロセス管理を組織に実装する手掛かりとなります(真のビジネスプロセス管理とは、組織がプロセスを管理する方法、プロセスが情報技術にサポートされる方法、これらの2つをお互いに反復プロセスで向上させる方法などです)。

元の状態に戻すためプロセスログを使用することもできます。ログにはランタイム時の情報変更記録がすべて含まれているため、プロセスを前の状態に戻すために逆順で実行することができます。

13.1. ログの作成

ビジネスプロセスマネージャモジュールはプロセス実行を実行する時にログを作成しますが、ユーザーもプロセスログを挿入することができます(ログエントリは、**org.jbpm.logging.log.ProcessLog**を継承するJavaオブジェクトです)。プロセスログエントリは、**ProcessInstance**の任意拡張である**LoggingInstance**に追加されます。

ビジネスプロセスマネージャは、グラフ実行ログ、コンテキストログ、タスク管理ログなど様々な種類のログを生成します。**inheritance tree**をナビゲートできるため、**org.jbpm.logging.log.ProcessLog**から開始するとよいでしょう。

LoggingInstanceはすべてのログエントリを収集します。**ProcessInstance**が保存されると、データベースへフラッシュされます(**ProcessInstance**の**logs**フィールドはHibernateへマップされていません。これは、各トランザクションでデータベースより読み取られるログを回避するためです)。

各**ProcessInstance**は実行パスのコンテキストで作成されます。そのため、**ProcessLog**はインデックスシーケンスジェネレータともなるそのトークンを参照します(後続のトランザクションによって作成されたログは順次的なシーケンス番号が付けられるため、ログの読み出しに重要となります)。

このAPIメソッドを使用してプロセスログを追加します。

```
public class LoggingInstance extends ModuleInstance {
    ...
    public void addLog(ProcessLog processLog) {...}
    ...
}
```


これは情報ロギングの UML 図になります。

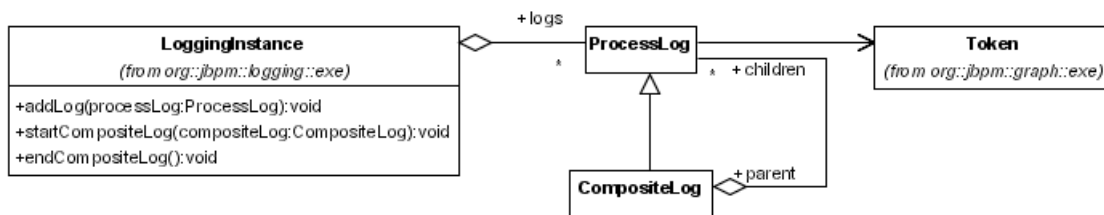


図13.1 JBPM ロギング情報クラス図

CompositeLog は特別なログエントリです。複数の子ログの親ログとなるため、階層構造を適用する方法を提案します。次のアプリケーションプログラミングインターフェースはログの挿入に使用されます。

```
public class LoggingInstance extends ModuleInstance {
    ...
    public void startCompositeLog(CompositeLog compositeLog) {...}
    public void endCompositeLog() {...}
    ...
}
```

階層構造の一貫性を保つため、**CompositeLog** は常に **try-finally-block** で呼び出されなければなりません。例は次の通りです。

```
startCompositeLog(new MyCompositeLog());
try {
    ...
} finally {
    endCompositeLog();
}
```

13.2. ログの設定

ログが重要でないデプロイメントの場合は、**jbpm.cfg.xml** 設定ファイルの **jbpm-context** セクションのロギング行を削除してください。

```
<service name='logging'
    factory='org.jbpm.logging.db.DbLoggingServiceFactory' />
```

ログをフィルタするには、**LoggingService** (**DbLoggingService** のサブクラス) のカスタム実装を記述します。その後、ロギングのカスタム **ServiceFactory** を作成し、**factory** 属性に指定します。

13.3. ログの読み出し

プロセスインスタンスのログは常にデータベースのクエリにて読み出さなければなりません。これには、**LoggingSession**による2つのメソッドがあります。

最初のメソッドは、プロセスインスタンスのすべてのログを読み出します。これらのログは、マップのトークンによってグループ化されます。このマップは、**ProcessLogs**のリストをプロセスインスタンスの全トークンに関連付けます。リストには**ProcessLogs**が作成順に記載されます。

```
public class LoggingSession {  
    ...  
    public Map findLogsByProcessInstance(long processInstanceId) {...}  
    ...  
}
```

2つ目のメソッドは、指定されたトークンのログを読み出します。リストには**ProcessLogs**が作成順に記載されます。

```
public class LoggingSession {  
    public List findLogsByToken(long tokenId) {...}  
    ...  
}
```

本章をお読みいただき、JBPMにおけるログインの仕組みや様々な使用方法について理解いただけたと思います。

第14章 JBPM プロセス定義言語

JBPM プロセス定義言語(JPDL)は、xml スキーマとすべてのプロセス定義に関するファイルをプロセスアーカイブにパッケージするためのメカニズムを指定します。

14.1. プロセスアーカイブ

プロセスアーカイブは zip ファイルです。プロセスアーカイブの中心的なファイルは **processdefinition.xml** となっています。このファイルに含まれる主な情報は、プロセスグラフです (アクションやタスクの情報も含まれています)。 **processdefinition.xml** には、アクションとタスクについての情報も含まれています。また、プロセスアーカイブにはタスクが必要とするクラスやユーザーインターフェース (UI) フォームなど、他のプロセス関連ファイルを格納することもできます。

14.1.1. プロセスアーカイブのデプロイ

process archiveは、以下の3つの方法でデプロイ可能です。

- **Process Designer Tool**
- **ant** タスク
- プログラム

Process Designer Toolでプロセスアーカイブをデプロイするには、以下の手順に従います (このプロセスは **starter's kit**で対応しています)。

1. プロセスアーカイブフォルダーを右クリックし、 **Deploy process archive** オプションを選択します。

starter's kit サーバーには **JBPM** アプリケーションが含まれており、このアプリケーションには **ProcessUploadServlet** と呼ばれるプロセスアーカイブをアップロードするサーブレットが存在します。このサーブレットはプロセスアーカイブをアップロードし、それをデフォルトの **JBPM** インスタンスにデプロイできます。

ant タスクでプロセスをデプロイを行うには、以下のコードを使います。

```
<target name="deploy-process">
  <taskdef name="deployproc" classname="org.jbpm.ant.DeployProcessTask">
    <classpath location="jbpm-jpdl.jar" />
  </taskdef>
  <deployproc process="build/myprocess.par" />
</target>
```

一度により多くのプロセスアーカイブをデプロイするには、ネストされた **fileset** 属性を使用します。このプロセス属性自体は任意です。 **ant** タスクの他の属性は以下の通りです。

表14.1 Ant 属性

属性	説明	必要性
----	----	-----

属性	説明	必要性
process	プロセスアーカイブへのパス	ネスト化されたりソースコレクション要素を利用していなければ YES
jbpmcfg	デフォルト値は、 jbpm.cfg.xml です。JBPM 設定ファイルは、データベースおよびマッピングファイルの JDBC 接続プロパティを含む Hibernate 設定ファイルの場所を指定できます (デフォルト値は hibernate.cfg.xml)。	No、 jbpm.cfg.xml がデフォルト
failonerror	false の場合、警告メッセージをログに残しますが、プロセス定義がデプロイに失敗してもビルドは停止されません。 True に設定されている場合、プロセス定義の1つのデプロイメントに失敗するとデプロイメントタスクが即座にエラーで失敗します。	No。True がデフォルト

プロセスアーカイブは、**org.jbpm.jpdl.par.ProcessArchiveDeployer** クラスの **parseXXX** の1つを使って、プログラムでデプロイできます。

14.1.2. プロセスバージョンニング

プロセスインスタンスは常に起動されたプロセス定義に対して実行します。ただし、**JBPM** では同じ名前の複数のプロセス定義をデータベースで共存させることができます。したがって通常、プロセスインスタンスは、処理時点で最新の有効なバージョンで開始して、その全生存期間は同じプロセス定義で実行し続けます。より新しいバージョンがデプロイされると、古いプロセスインスタンスが古いバージョンのプロセス定義で実行され続けているのと並行して、新しく作成されたインスタンスは最新のバージョンで開始されます。

このプロセスに **Java** クラスへの参照が含まれている場合、以下の2種類の方法のいずれかで **JBPM** ランタイム環境にて使えるようにできます。

- これらのクラスが **JBPM** クラスローダーで参照可能にすること

参照できるようにするには、プロセス定義のすべてがクラスファイルを参照するように **jbpm-[version].jar** の隣にある **.JAR** ファイルに委譲クラスを置きます。また、**Java** クラスはプロセスアーカイブにも含めることができます。プロセスアーカイブに委譲クラスを含めると (**jbpm** クラスローダーからは可視できません)、**JBPM** はプロセス定義内でこれらのクラスをバージョン管理します。



注記

プロセスのクラスローディングに関する詳細は、「[委譲](#)」を参照してください。

プロセスアーカイブがデプロイされると、**JBPM** データベースにプロセス定義を作成します。プロセス定義はプロセス定義名を基本にバージョンニングされます。名前がついたプロセスアーカイブがデプロイされると、デプロイヤーはバージョン番号を割り当てます。この番号を割り当てるには、デプロイヤーは同名のプロセス定義の最も高いバージョン番号をルックアップして1を加えます (名前のないプロセス定義は、バージョン番号は、いつも **-1** です)。

14.1.3. デプロイ済プロセス定義の変更

**警告**

デプロイ後にプロセス定義を変更するのは危険です。Red Hatはこのプロセスを推奨していません。プロセスインスタンスを新しい定義に移行してください。

このプロセスを実施する前に以下の項目を考慮してください。

- **org.jbpm.db.GraphSession** の **loadProcessDefinition** メソッド、**findProcessDefinition** メソッドでロードされるか、関係付けによりロードされるプロセス定義を更新する際の制限はありません。にもかかわらず、**setStartState(null)** などの呼び出しでプロセスが非常に簡単に混乱してしまいます。
- プロセス定義を変更すべきではないため、同梱の **Hibernate** 設定が定義クラスやコレクションの **nonstrict-read-write** キャッシュ戦略を指定します。この定義は、コミットされていない更新に関しても他のトランザクションで見えるようにできます。

14.1.4. プロセスインスタンスの移行

また、実行を新しいプロセス定義に変換してプロセス定義を変更する方法もあります。ビジネスプロセスの寿命は長いため、この操作は影響が大きいこと考慮してください。

**注記**

これは実験的な機能です。

プロセス定義データ、プロセスインスタンス（ランタイムデータ）と、ロギングデータの間に明確な違いがあります。この違いにより、JBPMデータベースで別のプロセス定義を（例えば、同じプロセスの新しいバージョンのデプロイによって）作成し、ランタイムの情報は新しいプロセス定義に変換します（ここでは、旧プロセスのトークンが新しいバージョンで削除されたノードを参照している場合、変換が行われます）。そのため、データベースには新しいデータだけが追加されます。しかし、ひとつのプロセス実行は2つのプロセスインスタンスオブジェクトに広がります。ツールの開発や統計計算を行う場合、問題が生じる可能性があります。

プロセスインスタンスを新バージョンに移行するには、以下のように **ChangeProcessInstanceVersionCommand** を実行してください。

```
new ChangeProcessInstanceVersionCommand()
    .processName("commute")
    .nodeNameMappingAdd("drive to destination", "ride bike to destination")
    .execute(jbpmContext);
```

14.2. 委譲

委譲メカニズムを使いプロセス実行にカスタムコードを含めます。

14.2.1. JBPMクラスローダー

JBPM クラスローダーは JBPM クラスをロードするクラスローダーです。クラスを JBPM クラスロー

ダーから見えるようにするには、それらを JAR ファイルに置き、その JAR ファイルを **jbpm-3.x.jar** の横に置きます。Web アプリケーションの場合には **jbpm-jpd1.jar** とともに **WEB-INF/lib** ディレクトリにカスタムの JAR ファイルを置きます。

14.2.2. プロセスクラスローダ

委譲クラスは、それぞれのプロセス定義のプロセスクラスローダーで読み込まれます (プロセスクラスローダーは親として JBPM クラスローダーをもつクラスローダーです。プロセスクラスローダーは、特定のプロセス定義のすべてのクラスを追加します)。クラスをプロセスアーカイブの **/classes** ディレクトリ内に置くことでクラスをプロセス定義に追加できます。これはプロセス定義に追加したいクラスにバージョンニングしたいときにだけ役に立つことに注意してください。バージョンニングが必要ない場合には、代わりに JBPM クラスローダーでクラスを利用可能にします。

リソース名がスラッシュで始まらない場合は、リソースはプロセスアーカイブの **/classes** ディレクトリからもロードされます。このディレクトリの外部にあるリソースをロードする場合は、スラッシュを 2つ (//) 先頭に指定します (たとえば、プロセスアーカイブファイルのルートにある **data.xml** をロードするには、**class.getResource("//data.xml")** を呼び出します)。

14.2.3. 委譲設定

委譲クラスは、プロセス実行内から呼ばれるユーザーコードを含みます。最も一般的なサンプルはアクションです。アクションの場合、**ActionHandler** インターフェースの実装は、プロセスのイベントに対して呼びだされます。委譲は、**processdefinition.xml** ファイルで指定されます。委譲を指定するときには、3つのデータを提供します。

1. クラス名(必須): 委譲クラスの完全修飾名
2. 設定タイプ (任意): 委譲オブジェクトのインスタンス化と設定方法を指定します。デフォルトでは、デフォルトコンストラクターが利用され、設定情報は無視されます。
3. 設定 (任意): 設定タイプに応じたフォーマットで記述された委譲オブジェクトの設定

以下は、全設定タイプの説明です。

14.2.3.1. config-type フィールド

これはデフォルト設定タイプです。**config-type**は、委譲クラスのオブジェクトを最初にインスタンス化し、それから設定で指定されている通りにオブジェクトのフィールドに値を設定します。設定は XML で、要素名がクラスのフィールド名に対応しなければなりません。要素の内容テキストは該当のフィールドにおかれます。また、必要で可能であれば、要素の内容テキストは、フィールドタイプに変換されます。

サポートされる型変換

- 文字列は省略されますが変換されません。
- **int**, **long**, **float**, **double**, ...などのプリミティブ型
- プリミティブ型の基礎ラッパークラス
- **list** と **set** と **collection**。その場合、xml コンテンツの各要素は **collection** の要素として扱われ、再帰的に変換しながらパースされます。要素の型が **java.lang.String** と異なる場合、完全修飾型名で **type** 属性を指定することによって示すことができます。例えば、以下のコードは **numbers** フィールドに文字列の **ArrayList** をインジェクトします:

```
<numbers>
  <element>one</element>
  <element>two</element>
  <element>three</element>
</numbers>
```

要素内のテキストはStringコンストラクターをもつ、いかなるオブジェクトにでも変換できます。他のString型でない他のものを利用する場合、このフィールドにて**element-type**で指定します。(この場合は**numbers**)

これがMapの別例です。

```
<numbers>
  <entry><key>one</key><value>1</value></entry>
  <entry><key>two</key><value>2</value></entry>
  <entry><key>three</key><value>3</value></entry>
</numbers>
```

- この場合、**field**の各要素には、子要素 **key** 1つと**value** のサブ要素 1つが必要です。これらは、変換ルールを利用して再帰的にパースされます。ちょうど **collections** と同じように、**type** 属性が指定されていないければ **java.lang.String** への変換が想定されます。
- **org.dom4j.Element**
- その他のタイプのために、**String** コンストラクタが利用されます

このクラスに着目してください。

```
public class MyAction implements ActionHandler {
  // access specifiers can be private, default, protected or public
  private String city;
  Integer rounds;
  ...
}
```

これは、このクラスに対して有効な設定です。

```
...
<action class="org.test.MyAction">
  <city>Atlanta</city>
  <rounds>5</rounds>
</action>
...
```

14.2.3.2. config-type Bean

これは、**config-type**と同様ですが、プロパティは**"setter"**メソッド経由で設定されます。ここではフィールドに直接設定され、同じ変換が適用されます。

14.2.3.3. config-type コンストラクター

このメソッドは、委譲のXML要素の内容を完全に取得し、テキストとして、委譲クラスのコンストラクタに渡します。

14.2.3.4. config-type configuration-property

最初に、デフォルトコンストラクタを利用すると、このメソッドは委譲のxml要素の内容を完全に取得し、テキストとして `void configure(String);` メソッドに渡します。

14.3. 式

JSP/JSF EL のような記述言語をサポートします。アクション、割り当て、決定条件では、`expression="#{myVar.handler[assignments].assign}"` のように記述することができます。



注記

この記述言語の基本については、<http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSPLIntro7.html> のチュートリアルを参照してください。

JPDLとJSFの記述言語は似ています。それは、JPDL ELはJSP ELに基づいていますが、対照的にJSFは `#{...}` を利用し、メソッドバインディングのサポートを含みます。

コンテキストによりますが、このプロセス変数、もしくはタスクインスタンス変数は、次の暗黙オブジェクトと共に、開始変数として使うことができます。

- `taskInstance (org.jbpm.taskmgmt.exe.TaskInstance)`
- `processInstance (org.jbpm.graph.exe.ProcessInstance)`
- `processDefinition (org.jbpm.graph.def.ProcessDefinition)`
- `token (org.jbpm.graph.exe.Token)`
- `taskMgmtInstance (org.jbpm.taskmgmt.exe.TaskMgmtInstance)`
- `contextInstance (org.jbpm.context.exe.ContextInstance)`

この機能は、JBoss SEAM 環境で使うと効果を発揮します (<http://www.jboss.com/products/seam>)。JBPM と SEAMを統合することで全バックエンドbeanや Enterprise Java Bean などはプロセス定義内からアクセスできるようになります。

14.4. JPDL XML スキーマ

JPDLスキーマはプロセスアーカイブにある `processdefinition.xml` ファイルで利用されるスキーマです。

14.4.1. バリデーション

JPDL の XML ドキュメントを解析する際、2つの条件に該当すると JBPM は本スキーマに対してこのドキュメントを検証します。

1. スキーマは XML ドキュメントで参照されます。

```
<process-definition xmlns="urn:jbpm.org:jpd1-3.2">
  ...
</process-definition>
```

2. Xerces パーサーがクラスパス上にあります。



注記

jPDLスキーマは`${jbpn.home}/src/java.jbpn/org.jbpn/jpdl/xml/jpdl-3.2.xsd`か、<http://jbpm.org/jpdl-3.2.xsd>にあります。

14.4.2. プロセス定義

表14.2 プロセス定義スキーマ

名前	種類	多重度	説明
name	属性	任意	プロセス名
swimlane	要素	[0..*]	プロセスで利用されるスイムレーン。スイムレーンはプロセス役割を表してタスク割り当てで利用されます。
start-state	要素	[0..1]	プロセスの開始状態。開始状態のないプロセスは有効ですが、実行することができないことに注意してください。
{end-state state node task-node process-state super-state fork join decision}	要素	[0..*]	プロセス定義のノード。ノードのないプロセスは有効ですが、実行することができないことに注意してください。
event	要素	[0..*]	アクションのコンテナとしてサービス提供
{action script create-timer cancel-timer}	要素	[0..*]	グローバルに定義されたアクションでイベントと遷移から参照可能。これらのアクションは名前を指定することで参照する必要があることに注意してください。
task	要素	[0..*]	アクションなどで使用できるグローバルに定義されたタスク
exception-handler	要素	[0..*]	このプロセス定義で委譲クラスによってスローされたすべてのエラーに適用される例外ハンドラのリスト

14.4.3. node

表14.3 Node スキーマ

名前	種類	多重度	説明
{action script create-timer cancel-timer}	要素	1	このノードの動作を表すカスタムアクション
一般的なノード要素			「 一般的なノード要素 」

14.4.4. 一般的なノード要素

表14.4 共通ノードスキーマ

名前	種類	多重度	説明
name	属性	必須	ノードの名前
async	属性	{ true false }, falseがデフォルト値	true に設定されている場合、このノードは非同期で実行されます。 10章 非同期の続行 も参照してください。
transition	要素	[0..*]	退場遷移。ノードを退場する遷移は一意の名前を持たなければいけません。名前がなくても許可される退場遷移は最大1つです。最初に指定されている遷移はデフォルト遷移と呼ばれます。デフォルト遷移は、遷移名を指定せずにノードが退場する時に使用されます。
event	要素	[0..*]	サポートされているイベントタイプ: {node-enter node-leave}
exception-handler	要素	[0..*]	プロセスノードから委譲クラスによってスローされたすべての例外に適用される例外ハンドラのリスト
timer	要素	[0..*]	このノードで実行の時間を監視するタイマーを指定します。

14.4.5. start-state

表14.5 開始状態スキーマ

名前	種類	多重度	説明
name	属性	任意	ノードの名前
task	要素	[0..1]	このプロセスに対して新しいインスタンスを起動するタスクまたはプロセスイニシエータをキャプチャーするタスク。「 開始タスクのスイムレーン 」を参照

名前	種類	多重度	説明
event	要素	[0..*]	サポートされているイベントタイプ: {node-leave}
transition	要素	[0..*]	退場遷移。ノードを退場する各遷移は一意の名前を持つ必要があります。
exception-handler	要素	[0..*]	プロセスノードから委譲クラスによってスローされたすべての例外に適用される例外ハンドラのリスト

14.4.6. end-state

表14.6 終了状態スキーマ

名前	種類	多重度	説明
name	属性	必須	終了状態の名前
end-complete-process	属性	任意	デフォルトでは、 end-complete-process は false であるため、この end-state を終了するトークンのみが終了されます。このトークンが最後に終了する子トークンである場合、親トークンも再帰的に終了されます。このプロパティを true に設定すると、全体のプロセスインスタンスが終了されます。
event	要素	[0..*]	サポートされているイベントタイプ: {node-enter}
exception-handler	要素	[0..*]	プロセスノードから委譲クラスによってスローされたすべての例外に適用される例外ハンドラのリスト

14.4.7. state

表14.7 状態スキーマ

名前	種類	多重度	説明
一般的なノード要素			「 一般的なノード要素 」を参照

14.4.8. task-node

表14.8 タスクノードスキーマ

名前	種類	多重度	説明
----	----	-----	----

名前	種類	多重度	説明
signal	属性	任意	{unsynchronized never first first-wait last last-wait} 、デフォルト値は last です。プロセス実行継続のタスク完了にどのように影響を与えるか指定します。
create-tasks	属性	任意	{yes no true false} 、デフォルト値は true です。ランタイム時に作成する必要があるタスクを決定しなければならない場合は、 false に設定できます。この場合は、アクションを node-enter に追加し、アクションにタスクを作成して、 create-tasks を false に設定します。
end-tasks	属性	任意	{yes no true false} 、デフォルト値は false です。 remove-tasks が node-leave で true に設定されている場合、開いているタスクはすべて終了されます。
task	要素	[0..*]	このタスクノードに到着したときに、作られるタスク
一般的なノード要素			「 一般的なノード要素 」を参照

14.4.9. process-state

表14.9 プロセス状態スキーマ

名前	種類	多重度	説明
sub-process	要素	1	バージョンと binding="late" が与えられると、JBPM はバージョン属性を無視し最新のバージョンを使います。
variable	要素	[0..*]	データを起動時にスーパープロセスからサブプロセスにどのようにコピーするかとデータをサブプロセスの完了時にサブプロセスからスーパープロセスにどのようにコピーするかを指定します。
一般的なノード要素			「 一般的なノード要素 」を参照

14.4.10. super-state

表14.10 親状態 (superstate) スキーマ

名前	種類	多重度	説明
{end-state state node task-node process-state super-state fork join decision}	要素	[0..*]	super-state のノード。super-state はネスト化できません。
一般的なノード要素			「 一般的なノード要素 」を参照

14.4.11. fork

表14.11 Fork スキーマ

名前	種類	多重度	説明
一般的なノード要素			「 一般的なノード要素 」を参照

14.4.12. join

表14.12 Join スキーマ

名前	種類	多重度	説明
一般的なノード要素			「 一般的なノード要素 」を参照

14.4.13. decision

表14.13 Decision スキーマ

名前	種類	多重度	説明
handler	要素	遷移上の 'handler' のエレメント、又は条件を指定する必要があります。	org.jbpm.jpdl.Def.DecisionHandler 実装の名前
遷移の条件	決定を退場する遷移の属性または要素テキスト		遷移にはすべてガード条件が存在します。決定ノードは、条件付きの退出遷移を検証し、条件が true の最初の遷移を選択します。条件を満たすものがない場合、デフォルトの遷移が選択されます。デフォルトの遷移は、無条件の遷移がある場合は最初のものを、そうでない場合は最初の条件付き遷移となっています。遷移は文書順に考慮されます。

名前	種類	多重度	説明
一般的なノード要素			「 一般的なノード要素 」を参照

14.4.14. event

表14.14 イベントスキーマ

名前	種類	多重度	説明
type	属性	必須	これはイベントタイプで、イベントが置かれた要素に対して相対的に表されます。
{action script create-timer cancel-timer}	要素	[0..*]	このイベントで実行すべきアクションのリスト

14.4.15. transition

表14.15 遷移スキーマ

名前	種類	多重度	説明
name	属性	任意	遷移の名前。ノードを退場する各遷移は一意の名前を持つ必要があることに注意してください。
to	属性	必須	宛先ノードの階層名。階層名の詳細については、「 階層名 」を参照してください。
condition	属性または要素テキスト	任意	ガード条件表現。これらの条件属性(または子要素)を決定ノードで使用、あるいはランタイム時にトークンで利用可能な遷移を計算するために使用できます。決定ノードを退出する遷移でのみ条件設定が可能です。
{action script create-timer cancel-timer}	要素	[0..*]	この遷移の発生時に実行するアクション。遷移のアクションをイベントに配置する必要がないことに注意してください(遷移は1つしかないためです)。
exception-handler	要素	[0..*]	プロセスノードから委譲クラスによってスローされたすべての例外に適用される例外ハンドラのリスト

14.4.16. action

表14.16 アクションスキーマ

名前	種類	多重度	説明
----	----	-----	----

名前	種類	多重度	説明
name	属性	任意	アクションの名前。アクションに名前が指定されている場合はプロセス定義から名前を検索できます。これは実行時アクションとアクションを一度だけ宣言する場合に便利です。
class	属性	参照名または式のいずれか	org.jbpm.graph.def.ActionHandler インターフェースを実装するクラスの完全修飾クラス名
ref-name	属性	this または class	参照されたアクション名。参照アクションが指定された場合はこのアクションの内容が処理されません。
expression	属性	this、class、ref-name のいずれか	メソッドを解決するjPDL表現。「式」も参照してください。
accept-propagated-events	属性	任意	オプションは {yes no true false} です。デフォルト値は yes true です。 false に設定された場合、アクションはこのアクションの要素でトリガーされたイベントでのみ実行されます。詳細については、「 イベントを渡す 」を参照してください。
config-type	属性	任意	{field bean constructor configuration-property} 。action-object の構築方法やこの要素の内容を action-object の設定情報として使用する方法を指定します。
async	属性	{true false}	' async="true" は、イベントでトリガーされた場合のみ action でサポートされます。デフォルト値は false であり、 action は実行スレッドで実行されます。 true に設定された場合は、コマンドエグゼキューターにメッセージが送信され、そのコンポーネントが別のトランザクションで非同期でアクションを実行します。
	{content}	任意	アクションの内容は、カスタムアクション実装の設定情報として使用できます。これにより、再利用可能な委譲クラスを作成できます。委譲設定の詳細については、「 委譲設定 」を参照してください。

14.4.17. script

表14.17 スクリプトスキーマ

名前	種類	多重度	説明
name	属性	任意	script-action の名前。アクションに名前が指定されている場合は、名前をプロセス定義から検索できます。これは、ランタイム時アクションの場合やアクションを一度だけ宣言する場合に便利です。
accept-propagated-events	属性	optional [0..*]	{yes no true false}。デフォルト値は yes true です。 false に設定された場合、アクションはこのアクションの要素でトリガされたイベントでのみ実行されます。詳細については、「 イベントを渡す 」を参照してください。
expression	要素	[0..1]	bean-shell スクリプト。変数要素を指定しないと、スクリプト要素の内容として表現を記述できます (表現エレメントタグは省略)。変数および／あるいは表現要素とあわせて「スクリプト要素の内容として表現」を利用する場合、スクリプト要素の内容は無視されます。 <pre>public void read(Element scriptElement, JpdlXmlReader jpdlReader) { if (scriptElement.isTextOnly()) { expression = scriptElement.getText(); } else { this.variableAccesses = new HashSet(jpdlReader.readVariableAccesses(scriptElement)); expression = scriptElement.element("expression").getText(); } }</pre>
variable	要素	[0..*]	スクリプトの変数。 in 変数が指定されていない場合は現在のトークンのすべての変数がスクリプト評価にロードされます。スクリプト評価にロードする変数の数を制限する場合は、 in 変数を使用します。

14.4.18. expression

表14.18 表現スキーマ

名前	種類	多重度	説明
	{content}		Bean シェルスクリプト

14.4.19. variable

表14.19 変数スキーマ

名前	種類	多重度	説明
name	属性	必須	プロセス変数名
access	属性	任意	デフォルト値は read, write です。これはアクセス指定子のコンマ区切りリストです。これまで使用されたアクセス指定子は read 、 write 、 required のみです。 「 required 」はタスク変数をプロセス変数に送信する場合のみ適切です。
mapped-name	属性	任意	デフォルトでは変数名に設定されます。変数名がマップされた名前を指定します。mapped-nameの意味はこの要素が使用されるコンテキストに依存します。スクリプトの場合は script-variable-name になります。タスクコントローラの場合はタスクフォームパラメータのラベルになり、 process-state の場合は sub-process で使用された変数名になります。

14.4.20. handler

表14.20 ハンドラスキーマ

名前	種類	多重度	説明
expression	属性	this または class	JPDL 式。返された結果は toString() メソッドにより文字列に変換されます。変換された文字列は退場遷移のいずれかに一致します。「式」も参照してください。
class	属性	this または ref-name	org.jbpm.graph.node.DecisionHandler インターフェースを実装するクラスの完全修飾クラス名
config-type	属性	任意	{field bean constructor configuration-property}。action-object の構築方法やこのエレメントの内容を action-object の設定情報として使用する方法を指定します。
	{content}	任意	ハンドラの内容はカスタムハンドラ実装の設定情報として使用できます。これにより、再利用可能な移譲クラスを作成できます。移譲設定の詳細については、「 委譲設定 」を参照してください。

14.4.21. timer

表14.21 タイマースキーマ

名前	種類	多重度	説明
name	属性	任意	タイマーの名前。名前を指定しない場合は、閉じるノードの名前が取得されます。各タイマーは一意的な名前を持つことに注意してください。
duedate	属性	必須	タイマーの作成からタイマーの実行までの時間(営業時間で示すこともできます)。構文については「 Duration (期間) 」を参照してください。
repeat	属性	任意	{duration 'yes' 'true'}。タイマーが期日に実行された後は、'repeat' はノードが退場するまでの繰り返しでタイマーを実行する間隔をオプションで指定します。 yes または true が指定された場合は、 repeat に期日と同じ期間が使用されます。構文については、「 Duration (期間) 」を参照してください。
transition	属性	任意	タイマーイベントが発生しアクションを実行した後にタイマーが実行されたときに取得される transition-name
cancel-event	属性	任意	この属性はタスクのタイマーでのみ使用されます。タイマーをキャンセルするイベントを指定します。デフォルトは、 task-end イベントですが、 task-assign や task-start を設定することもできます。属性にてカンマ区切りリストを指定すると、 cancel-event タイプを組み合わせることができます。
{action script create-timer cancel-timer}	要素	[0..1]	このタイマーが起動したときに実行されるアクション

14.4.22. create-timer

表14.22 Create Timer スキーマ

名前	種類	多重度	説明
name	属性	任意	タイマーの名前。この名前は cancel-timer アクションでタイマーをキャンセルするのに使用できません。
duedate	属性	必須	タイマーの作成からタイマーの実行までの時間(営業時間で示すこともできます)。構文については「 Duration (期間) 」を参照してください。

名前	種類	多重度	説明
repeat	属性	任意	{duration 'yes' 'true'}。タイマーが期日に実行された後は、'repeat' はノードが退場するまでの繰り返しでタイマーを実行する間隔をオプションで指定します。 yes または true が指定された場合は、 repeat に期日と同じ期間が使用されます。構文については、「 Duration（期間） 」を参照してください。
transition	属性	任意	タイマーイベントが発生し、アクションを実行した後にタイマーが実行されたときに取得する transition-name

14.4.23. cancel-timer

表14.23 Cancel Timer スキーマ

名前	種類	多重度	説明
name	属性	任意	キャンセルされるタイマーの名前。

14.4.24. task

表14.24 タスクスキーマ

名前	種類	多重度	説明
name	属性	任意	タスクの名前。名前が付けられたタスクは、 TaskMgmtDefinition によって参照および検索できます。
blocking	属性	任意	{yes no true false}、デフォルトはfalseです。もしtrueにセットされた場合、タスクが終了していない際にはノードを退場することはできません。もしfalse（デフォルト）でセットされた場合、トークンのシグナルは実行を続行し、ノードを退場することができます。blockingは普通、ユーザインターフェースによって行われるものなのでデフォルトとしてfalseにしています。
signalling	属性	任意	{yes no true false}、デフォルトはtrueです。シグナリングがfalseに設定されている場合は、このタスクはトークンの続行をトリガーする機能を持ちません。
duedate	属性	任意	11章ビジネスカレンダー で説明された絶対または営業時間で示された期間。

名前	種類	多重度	説明
swimlane	属性	任意	スイムレーンの参照。タスクにスイムレーンを指定した場合、割り当ては無視されます。
priority	属性	任意	{highest, high, normal, low, lowest}のいずれか。または、優先順位として整数を指定できます(最大=1、最小=5)。
assignment	要素	任意	タスクが作成された時に、アクターにタスクを割り当てる委譲を記述します。
event	要素	[0..*]	サポートされているイベントタイプは{task-create task-start task-assign task-end}。task-assignに対して特別に非永続化プロパティpreviousActorIdをTaskInstanceに追加しました。
exception-handler	要素	[0..*]	プロセスノードでスローされた移譲クラスによってスローされたすべての例外に適用される例外ハンドラのリスト
timer	要素	[0..*]	このタスクで実行の継続時間を監視するタイマーを指定します。タスクタイマーに対して特別にcancel-eventを指定できます。デフォルトでは、cancel-eventはtask-endですが、task-assignやtask-startなどにカスタマイズできます。
controller	要素	[0..1]	プロセス変数をどのようにタスクフォームパラメーターに変換するかを指定します。タスクフォームパラメーターはタスクフォームをユーザーにレンダリングするためにユーザーインターフェースによって使用されます。

14.4.25. スイムレーン

表14.25 スイムレーンスキーマ

名前	種類	多重度	説明
name	属性	必須	スイムレーンの名前。スイムレーンはTaskMgmtDefinitionを使用して参照および検索できます。
assignment	要素	[1..1]	スイムレーンの割り当てを指定します。割り当ては、このスイムレーンで最初のタスクインスタンスが作成されたときに実行されます。

14.4.26. 割り当て

表14.26 割り当てスキーマ

名前	種類	多重度	説明
expression	属性	任意	歴史的な経緯から、この属性表現は JPDL 式を参照しません。ただし、JBPM アイデンティティコンポーネントの割り当て表現となります。JBPM アイデンティティコンポーネント表現の記述方法の詳細については、「 割り当て式 」を参照してください。この実装は <code>jbpm</code> アイデンティティコンポーネントの依存関係を持つことに注意してください。
actor-id	属性	任意	<code>actorId</code> 。 <code>pooled-actors</code> と組み合わせて使用できます。 <code>actor-id</code> は表現として解決されます。したがって、 <code>actor-id="bobthebuilder"</code> のような固定 <code>actorId</code> を参照できます。または、タスクインスタンス変数 <code>"myVar"</code> で <code>getActorId</code> メソッドを呼び出す <code>actor-id="myVar.actorId"</code> のような文字列を返すプロパティまたはメソッドを参照できます。
pooled-actors	属性	任意	コンマ区切りの <code>actorId</code> のリスト。 <code>actor-id</code> と組み合わせて使用できます。固定のプール <code>actorId</code> のセットは <code>pooled-actors="chicagobulls, pointersisters"</code> のように指定することができます。 <code>pooled-actors</code> は、表現として解決されます。そのため、 <code>String[]</code> 、コレクション、プールアクターのコンマ区切りのリストを返さなければならぬプロパティやメソッドも参照できます。
class	属性	任意	<code>org.jbpm.taskmgmt.def.AssignmentHandler</code> 実装の完全修飾クラス名
config-type	属性	任意	<code>{field bean constructor configuration-property}</code> 。 <code>assignment-handler-object</code> の構築方法やこのエレメントの内容を <code>assignment-handler-object</code> の設定情報として使用する方法を指定します。
	<code>{content}</code>	任意	<code>assignment-element</code> の内容は <code>AssignmentHandler</code> 実装の設定情報として使用できます。これにより、再利用可能な委譲クラスを作成できます。委譲設定の詳細については、「 委譲設定 」を参照してください。

14.4.27. Controller

表14.27 コントローラスキーマ

名前	種類	多重度	説明
class	属性	任意	<code>org.jbpm.taskmgmt.def.TaskControllerHandler</code> 実装の完全修飾クラス名

名前	種類	多重度	説明
config-type	属性	任意	{field bean constructor configuration-property}。 assignment-handler-object の構築方法やこの要素の内容を assignment-handler-object の設定情報として使用する方法を指定します。
	{content}		どちらの場合もコントローラーの内容は指定されたタスクコントローラーハンドラーの設定になります(クラス属性が指定されている場合)。タスクコントローラーハンドラーが指定されていない場合は、内容をvariable要素のリストにしなければなりません。
variable	要素	[0..*]	タスクコントローラーがクラス属性によって指定されていない場合、controller要素の内容は変数のリストである必要があります。

14.4.28. sub-process

表14.28 サブプロセススキーマ

名前	種類	多重度	説明
name	属性	必須	呼び出す sub-process の名前。EL 式を使用できますが、 String との検証が必要です。
version	属性	任意	呼び出す sub-process のバージョン。version が指定されない場合、 process-state は指定プロセスの最新版を使用します。
binding	属性	任意	Sub-process が解決されるタイミングを定義します。オプションは{early late}。デフォルトは、解決は early でデプロイメント時に行う設定です。binding が late として定義されている場合、 process-state は各実行時に指定プロセスの最新版を解決します。Late バインディングは、固定のバージョンとの組み合わせでは意味がないため、 binding="late" の場合は version 属性は無視されます。

14.4.29. condition

表14.29 条件スキーマ

名前	種類	多重度	説明
----	----	-----	----

名前	種類	多重度	説明
	このオプションは {content} で、後方互換性を保証するために、 expression 属性では条件を入力することもできます。ただし、この属性はバージョン3.2で削除されています。	必須	condition 要素の内容は Boolean を評価する JPDL 表現です。表現が true に解決された最初の遷移 (processdefinition.xml の順番通り) を取ります。 true に解決される条件がない場合、デフォルトの退場遷移 (最初のもの) が取得されます。

14.4.30. exception-handler

表14.30 例外ハンドラスキーマ

名前	種類	多重度	説明
exception-class	属性	任意	これは、 Java の「スロー可能な」クラスの完全修飾名を指定しており、この例外ハンドラと一致するはずです。この属性が指定されないと、すべての例外 (java.lang.Throwable) に一致します。
action	要素	[1..*]	この例外ハンドラーによってエラー処理されるときに実行するアクションのリスト

第15章 WORKFLOW のテスト駆動開発 (TDD)

15.1. WORKFLOW のテスト駆動開発の紹介

プロセス指向プログラムの開発は、他のソフトウェアの開発と何ら変わりはないので、簡単にプロセス定義をテストできるべきだと考えます。この章では、JUnit の利用し拡張なしにカスタムのプロセス定義を単体テストする方法を見ていきます。

開発サイクルはできる限り短く保つべきです。(できれば中間の構築ステップを踏まず)ソフトウェアのソースコードへの変更をすべてすぐ検証してください。以下の例では、この方法を用いて JBPM プロセスを開発、テストする方法をお見せします。

プロセス定義の単体テストのほとんどは、実行ベースです。各シナリオは、1つの JUnit テストメソッドで実行され、外部トリガー(シグナル)でプロセス実行に送り込んで、各シグナルの後にプロセスが期待された状態にあるかを検証します。

これは、そのようなテストのサンプルをグラフ表示したものです。オークションプロセスの簡単なバージョンを取り上げます。

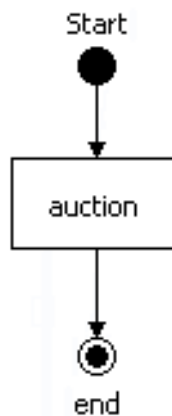


図15.1 オークションテストプロセス

次に、メインシナリオを実行するテストを記述してみましょう。

```

public class AuctionTest extends TestCase {

    // parse the process definition
    static ProcessDefinition auctionProcess =
        ProcessDefinition.parseParResource("org/jbpm/tdd/auction.par");

    // get the nodes for easy asserting
    static StartState start = auctionProcess.getStartState();
    static State auction = (State) auctionProcess.getNode("auction");
    static EndState end = (EndState) auctionProcess.getNode("end");

    // the process instance
    ProcessInstance processInstance;

    // the main path of execution
    Token token;

    public void setUp() {

```

```

        // create a new process instance for the given process definition
        processInstance = new ProcessInstance(auctionProcess);

        // the main path of execution is the root token
        token = processInstance.getRootToken();
    }

    public void testMainScenario() {
        // after process instance creation, the main path of
        // execution is positioned in the start state.
        assertSame(start, token.getNode());

        token.signal();

        // after the signal, the main path of execution has
        // moved to the auction state
        assertSame(auction, token.getNode());

        token.signal();

        // after the signal, the main path of execution has
        // moved to the end state and the process has ended
        assertSame(end, token.getNode());
        assertTrue(processInstance.hasEnded());
    }
}

```

15.2. XMLソース

実行シナリオを書き始める前に、**ProcessDefinition**を作成する必要があります。

ProcessDefinition オブジェクトを取得する最も簡単な方法は、XML を構文解析することです。

ProcessDefinition.parse とタイプして、コード補完機能を起動させてください。構文解析を行うさまざまなメソッドが表示されます。**ProcessDefinition** オブジェクトに構文解析可能な基本的記述方法は3つあります。

15.2.1. プロセスアーカイブの構文解析

プロセスアーカイブとは、**processdefinition.xml** というプロセスの XML ファイルが含まれている zip ファイルです。**JBPM プロセスデザイナー** プラグインはプロセスアーカイブを読み書きします。

```

static ProcessDefinition auctionProcess =
    ProcessDefinition.parseParResource("org/jbpm/tdd/auction.par");

```

15.2.2. XML ファイルの構文解析

手作業で **processdefinition.xml** ファイルを記述する場合 **Jpd1XmlReader** を利用し、**ant** スクリプトを使い、できた ZIP ファイルをパッケージします。

```

static ProcessDefinition auctionProcess =
    ProcessDefinition.parseXmlResource("org/jbpm/tdd/auction.xml");

```

15.2.3. XML String の構文解析

シンプルなストリングからのインラインの単体テストでXML を構文解析する方法：

```
static ProcessDefinition auctionProcess =  
    ProcessDefinition.parseXmlString(  
        "<process-definition>" +  
        "  <start-state name='start'>" +  
        "    <transition to='auction'/>" +  
        "  </start-state>" +  
        "    <state name='auction'>" +  
        "      <transition to='end'/>" +  
        "    </state>" +  
        "  <end-state name='end'/>" +  
        "</process-definition>");
```

付録A GNU LESSER GENERAL PUBLIC LICENSE 2.1

GNU LESSER GENERAL PUBLIC LICENSE Version 2.1, February 1999

Copyright (C) 1991, 1999 Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL. It also counts
as the successor of the GNU Library Public License, version 2, hence
the version number 2.1.]

Preamble

The licenses for most software are designed to take away your
freedom to share and change it. By contrast, the GNU General Public
Licenses are intended to guarantee your freedom to share and change
free software--to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some
specially designated software packages--typically libraries--of the
Free Software Foundation and other authors who decide to use it. You
can use it too, but we suggest you first think carefully about whether
this license or the ordinary General Public License is the better
strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use,
not price. Our General Public Licenses are designed to make sure that
you have the freedom to distribute copies of free software (and charge
for this service if you wish); that you receive source code or can get
it if you want it; that you can change the software and use pieces of
it in new free programs; and that you are informed that you can do
these things.

To protect your rights, we need to make restrictions that forbid
distributors to deny you these rights or to ask you to surrender these
rights. These restrictions translate to certain responsibilities for
you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis
or for a fee, you must give the recipients all the rights that we gave
you. You must make sure that they, too, receive or can get the source
code. If you link other code with the library, you must provide
complete object files to the recipients, so that they can relink them
with the library after making changes to the library and recompiling
it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the
library, and (2) we offer you this license, which gives you legal
permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that
there is no warranty for the free library. Also, if the library is
modified by someone else and passed on, the recipients should know

that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a

"work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

GNU LESSER GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- a) The modified work must itself be a software library.
 - b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.
 - c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.
 - d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.
- (For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the

Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

- a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)
- b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.
- c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.
- d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.
- e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library

facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

- a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.
- b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME

THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the library's name and a brief idea of what it does.>

Copyright (C) <year> <name of author>

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the

library `Frob' (a library for tweaking knobs) written by James Random Hacker.

<signature of Ty Coon>, 1 April 1990
Ty Coon, President of Vice

That's all there is to it!

付録B 改訂履歴

改訂 5.2.0-1.402 Rebuild with Publican 4.0.0	Fri Oct 25 2013	Rüdiger Landmann
改訂 5.2.0-1.33 Rebuild for Publican 3.0	2012-07-25	Anthony Towns
改訂 5.2.0-0 SOA 5.2 向けの更新	Wed Jun 29 2011	David Le Sage
改訂 5.1.0-0 SOA 5.1 向けの更新	Fri Feb 18 2011	David Le Sage
改訂 5.0.2-0 SOA 5.0.2 向けの更新	Wed May 26 2010	David Le Sage
改訂 5.0.1-0 SOA 5.0.1 向けの更新	Tue Apr 20 2010	David Le Sage
改訂 5.0.0-0 作成	Sat Jan 30 2010	David Le Sage