JBoss Enterprise Application Platform 5

Hibernate Entity Manager リファレンスガイド

JBoss Enterprise Application Platform 5での使用向け
エディッション 5.1.2

Last Updated: 2017-10-26

JBoss Enterprise Application Platform 5 Hibernate Entity Manager リファレンスガイド

JBoss Enterprise Application Platform 5での使用向け
エディッション 5.1.2

Red Hat Documentation Group

法律上の通知

概要

JBoss Enterprise Application Platform 5およびパッチリリース向けのHibernate Entity Manager リファレンスガイド

目次

第1章 アーキテクチャ

1.1. 定義

EJB3 は Java EE 5.0 プラットフォームの一部です。EJB3 の永続化は EJB3 コンテナと特定のコンテナの外部で実行されるスタンドアロン J2SE アプリケーションで利用可能です。両方の環境では以下のプログラミングインタフェースとアーチファクトが利用できます。

`EntityManagerFactory`

エンティティマネージャーファクトリは、エンティティマネージャーインスタンスを提供し、すべてのインスタンスは同じデータベースに接続して特定の実装で定義されたのと同じデフォルト設定を使用します。複数のデータストアにアクセスするために複数のエンティティマネージャーファクトリを準備できます。このインタフェースはネイティブの Hibernate の **`SessionFactory`** に似ています。

`EntityManager`

特定の作業単位のデータベースにアクセスするには **`EntityManager`** API が使用されます。この API は、永続エンティティインスタンスを作成および削除し、プライマリキー ID によりエンティティを見つけ、すべてのエンティティに対して問い合わせるために使用されます。このインスタンスは Hibernate の **`Session`** に似ています。

永続コンテキスト

永続コンテキストは、永続エンティティ ID に対して一意のエンティティインスタンスがあるエンティティインスタンスのセットです。永続コンテキスト内で、エンティティインスタンスとライフサイクルは特定のエンティティマネージャーで管理されます。このコンテキストのスコープはトランザクションまたは拡張された作業単位のいずれかです。

永続性単位

該当するエンティティマネージャーにより管理できるエンティティタイプのセットは、永続性単位で定義されます。永続性単位はアプリケーションに関連するか、またはアプリケーションによりグループ化されたすべてのクラスセット (単一のデータストアに対するマッピングに配置する必要がある) を定義します。

コンテナにより管理されたエンティティマネージャー

ライフサイクルがコンテナにより管理されたエンティティマネージャー

アプリケーションにより管理されたエンティティマネージャー

ライフサイクルがアプリケーションにより管理されたエンティティマネージャー

JTA エンティティマネージャー

JTA トランザクションに関連するエンティティマネージャー

リソースローカルエンティティマネージャー

リソーストランザクションを使用するエンティティマネージャー (非 JTA トランザクション)

1.2. EJB コンテナ環境

1.2.1. コンテナにより管理されたエンティティマネージャー

Java EE 環境で最も一般的で幅広く使用されているエンティティマネージャーはコンテナにより管理されたエンティティマネージャーです。このモードでは、コンテナはエンティティマネージャーのオープンとクローズを行います (これはアプリケーションに対して透過的です)。また、トランザクション境界も担当します。コンテナにより管理されたエンティティマネージャーは依存関係挿入または JNDI ルックアップを使用してアプリケーションで取得されます。コンテナにより管理されたエンティティマネージャーでは JTA トランザクションを使用する必要があります。

1.2.2. アプリケーションにより管理されたエンティティマネージャー

アプリケーションにより管理されたエンティティマネージャーを使用すると、アプリケーションコードでエンティティマネージャーを制御できます。このエンティティマネージャは **`EntityManagerFactory`** API を使用して取得されます。アプリケーションにより管理されたエンティティマネージャーは、現在の JTA トランザクション (JTA エンティティマネージャー) に関係付けることができます。または、トランザクションは **`EntityTransaction`** API (リソースローカルエンティティマネージャー) を使用して制御できます。リソースローカルエンティティマネージャートランザクションは直接リソーストランザクションに対してマップされます (つまり、Hibernate の場合は JDBC トランザクション)。エンティティマネージャータイプ (JTA またはリソースローカル) は設定時 (エンティティマネージャーファクトリの設定時) に定義されます。

1.2.3. 永続コンテキストスコープ

エンティティマネージャーは永続コンテキストと対話する API です。トランザクション境界に永続コンテキストをバインドすることと複数のトランザクションで永続コンテキストを利用可能にすることの 2 つの一般的な方針があります。

最も一般的なケースは、現在のトランザクションスコープに永続コンテキストスコープをバインドすることです。これは、JTA トランザクションが使用されている場合のみ可能です。永続コンテキストは JTA トランザクションライフサイクルに関連付けられます。エンティティマネージャーが呼び出されるときに、永続コンテキストが現在の JTA トランザクションに関連付けられていない場合は、永続コンテキストもオープンされます。それ以外の場合は、関連付けられた永続コンテキストが使用されます。永続コンテキストは、JTA トランザクションが完了したときに終了します。つまり、JTA トランザクションの間は、アプリケーションが同じ永続コンテキストの管理対象エンティティを処理できます。エンティティマネージャーの永続コンテキストを EJB メソッドコールで渡す必要はありません。ただし、エンティティマネージャーが必要な場合は、常に依存関係挿入またはルックアップを使用してください。

また、拡張された永続コンテキストも使用できます。これは、コンテナにより管理されたエンティティマネージャーを使用する場合に、ステートフルセッション Bean と組み合わせることができます。永続コンテキストは、エンティティマネージャーが依存関係挿入または JNDI ルックアップから取得され、**`Remove`** ステートフルセッション Bean メソッドの完了後にコンテナがクローズするまで保持されます。これは、「長い」作業単位パターンを実装するのに最適なメカニズムです。たとえば、単一の作業単位として複数のユーザー対話サイクルを取り扱う場合 (完全に完了する必要があるウィザードダイアログなど) は、通常アプリケーションユーザーの観点からこれを作業単位としてモデル化し、拡張された永続コンテキストとして実装します。このパターンの詳細については、Hibernate リファレンスマニュアルまたは『Hibernate In Action』を参照してください。JBoss Seam は対話および作業単位の表記法について JSF と EJB3 をリンクするフレームワークです。アプリケーションにより管理されたエンティティマネージャーの場合、永続コンテキストはエンティティマネージャーが作成されたときに作成され、エンティティマネージャーがクローズされるまで保持されます。拡張された永続コンテキストでは、トランザクション外部で実行されたすべての変更操作 (persist、merge、remove) が、永続コンテキストがトランザクションにアタッチされるまでキューに格納されます。トランザクションは、通常ユーザープロセス終了時に実行され、プロセス全体をコミットしたり、ロールバックしたりできます。アプリケーションにより管理されたエンティティマネージャーは拡張された永続コンテキストのみをサポートします。

リソースローカルエンティティマネージャーまたは

`EntityManagerFactory.createEntityManager()` (アプリケーションにより管理) で作成されたエンティティマネージャーは、永続コンテキストと 1 対 1 関係を持ちます。他の状況では、**永続コンテキスト伝播**が行われます。

1.2.4. 永続コンテキスト伝播

永続コンテキスト伝播は、コンテナにより管理されたエンティティマネージャーに対して行われます。

トランザクションスコープコンテナにより管理されたエンティティマネージャー (Java EE 環境の一般的なケース) で、JTA トランザクション伝播が永続コンテキストリソース伝播と同じです。つまり、該当する JTA トランザクション内で取得された、コンテナにより管理されたトランザクションスコープエンティティマネージャーはすべて同じ永続コンテキストを共有します。Hibernate の用語では、これはすべてのマネージャーが同じセッションを共有することを意味します。

重要: 永続コンテキストは異なる JTA トランザクション間または同じエンティティマネージャファクトリ経由のエンティティマネージャー間で共有されません。拡張された永続コンテキストを使用する場合は、コンテキスト伝播に対していくつかの重要な例外があります。

- ステートレスセッション Bean、メッセージ駆動 Bean、またはトランザクションスコープ永続コンテキストを持つステートフルセッション Bean が、同じ JTA トランザクションの拡張永続コンテキストを持つステートフルセッション Bean を呼び出す場合は、IllegalStateException がスローされます。

- 拡張された永続コンテキストを持つステートフルセッション Bean がステートレスセッション Bean または同じ JTA トランザクションのトランザクションスコープ永続コンテキストを持つステートフルセッション Bean を呼び出す場合は、永続コンテキストが伝播されます。

- 拡張された永続コンテキストを持つステートフルセッション Bean が異なる JTA トランザクションコンテキストのステートレスまたはステートフルセッション Bean を呼び出す場合、永続コンテキストは伝播されません。

- 拡張された永続コンテキストを持つステートフルセッション Bean が拡張された永続コンテキストを持つ別のステートフルセッション Bean をインスタンス化する場合、拡張された永続コンテキストは 2 つ目のステートフルセッション Bean により継承されます。2 つ目のステートフルセッション Bean が最初のものとは異なるトランザクションコンテキストで呼び出された場合は、IllegalStateException がスローされます。

- 拡張された永続コンテキストを持つステートフルセッション Bean が、同じトランザクションの異なる拡張永続コンテキストを持つステートフルセッション Bean を呼び出す場合は、IllegalStateException がスローされます。

1.3. JAVA SE 環境

Java SE 環境では、アプリケーションにより管理された拡張コンテキストエンティティマネージャーのみが利用できます。**`EntityManagerFactory`** API を使用してエンティティマネージャーを取得できます。リソースローカルエンティティマネージャーのみが利用可能です。つまり、JTA トランザクションおよび永続コンテキスト伝播は Java SE ではサポートされません (永続コンテキストは、たとえば、Hibernate コミュニティで人気があるスレッドローカルセッションパターンを使用して永続コンテキストを手動で伝播する必要があります)。

拡張コンテキストは、エンティティマネージャーが取得されたときに永続コンテキストが作成され (**`EntityManagerFactory.createEntityManager(...)`** の使用)、エンティティマネージャーがクローズされたときに永続コンテキストがクローズされることを意味します。この場合、多くのリソースローカルトランザクションは同じ永続コンテキストを共有します。

第2章 設定と構成設定

2.1. 設定

EJB 3.0 / JPA 対応の Hibernate EntityManager は、Hibernate core および Hibernate Annotations上に構築されています。各モジュールと互換のあるバージョンを利用する必要がありますので、hibernate.org のダウンロードセクションにある互換性マトリックスを参照してください。以下のライブラリはご利用中のクラスパス (hibernate3.jar、hibernate-annotations.jar、hibernate-commons-annotations.jar、hibernate-entitymanager.jarおよび (ejb-persistence.jarを含む) 各パッケージ用のサードパーティライブラリ) に置く必要があります。

2.2. 構成とブートストラップ

2.2.1. パッケージング

アプリケーションサーバーとスタンドアロンアプリケーション内のエンティティマネージャの設定は永続アーカイブに存在します。永続アーカイブは、**META-INF** フォルダにある **persistence.xml** ファイルを定義する JAR ファイルです。アーカイブに含まれる適切にアノテートされたすべてのクラス (**@Entity** アノテーションを持ちます)、アーカイブに含まれるアノテートされたすべてのパッケージ、および Hibernate hbm.xml ファイルが永続ユニット設定に追加されます。したがって、デフォルトでは、persistence.xml が最低要件になります。

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
   version="1.0">
   <persistence-unit name="sample">
      <jta-data-source>java:/DefaultDS</jta-data-source>
      <properties>
         <property name="hibernate.dialect"
value="org.hibernate.dialect.HSQLDialect"/>
         <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
      </properties>
   </persistence-unit>
</persistence>
```

以下に **persistence.xml** ファイルの完全な例を示します。

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
   version="1.0">
   <persistence-unit name="manager1" transaction-type="JTA">
      <provider>org.hibernate.ejb.HibernatePersistence</provider>
      <jta-data-source>java:/DefaultDS</jta-data-source>
      <mapping-file>ormap.xml</mapping-file>
      <jar-file>MyApp.jar</jar-file>
      <class>org.acme.Employee</class>
      <class>org.acme.Person</class>
      <class>org.acme.Address</class>
      <properties>
```

```
         <property name="hibernate.dialect"
value="org.hibernate.dialect.HSQLDialect"/>
         <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
      </properties>
   </persistence-unit>
</persistence>
```

name

(属性) 各エンティティマネージャは名前を持つ必要があります。

transaction-type

(属性) 使用されたトランザクションタイプ。JTA または RESOURCE_LOCAL (デフォルトで JavaEE 環境の JTA、JavaSE 環境の RESOURCE_LOCAL に設定されます) のいずれかになります。jta-datasource が使用された場合、デフォルト値は JTA になり、non-jta-datasource が使用された場合は RESOURCE_LOCAL が使用されます。

provider

プロバイダは EJB Persistence プロバイダの完全修飾クラス名です。複数の EJB3 実装を使用しない場合は、これを定義する必要はありません。これは、EJB Persistence の複数のベンダー実装を使用する場合に必要です。

jta-data-source、non-jta-data-source

これは、javax.sql.DataSource が存在する場所の JNDI 名です。JNDI 対応 Datasource なしで実行する場合は、JDBC 接続を Hibernate 固有のプロパティで指定する必要があります (以下参照)。

mapping-file

Class 要素は、マップする EJB3 準拠の XML マッピングファイルを指定します。このファイルは classpath に含まれる必要があります。EJB3 仕様ごとに、Hibernate EntityManager は **META_INF/orm.xml** で指定された jar ファイルにあるマッピングファイルをロードしようとします。当然、明示的なマッピングファイルもロードされます。実際には、マッピングファイルエレメントの任意の XML ファイルを提供できます (hbm ファイルまたは EJB3 配備記述子のいずれか)。

jar-file

jar-file エレメントは分析する jar を指定します。適切にアノテートされたすべてのクラス、アノテートされたパッケージ、およびこの jar ファイルに含まれるすべての hbm.xml ファイルが、永続ユニット設定に追加されます。このエレメントは主に Java EE 環境で使用されます。Java SE でのこれの使用は移植不可と見なされます。この場合は、絶対 url が必要です。または、ディレクトリを参照できます (これは、テスト環境で特に役に立ちます。persistence.xml ファイルはドメインモデルと同じルートディレクトリまたは jar に存在しません)。

```
         <jar-file>file:/home/turin/work/local/lab8/build/classes</jar-
file>
```

exclude-unlisted-classes

アノテートされたクラスに対して主な jar ファイルをチェックしないでください。明示的なクラスだけが永続ユニットの一部となります。

class

class 要素はマップする完全修飾クラス名を指定します。デフォルトでは、適切にアノテートされたすべてのクラスとアーカイブ内にあるすべての hbm.xml ファイルが永続ユニット設定に追加されま

す。ただし、class 要素を使用して一部の外部エンティティを追加できます。仕様の拡張として、パッケージ名を **<class>** 要素 (**<class>org.hibernate.eg</class>** など) に追加できます。**<class>** 要素でパッケージを指定すると、アノテートされたクラスのみが含まれます。

properties

ベンダー固有プロパティを指定するには、property 要素が使用されます。property 要素では、Hibernate 固有の設定を定義します。また、JDBC 接続情報も指定する必要があります。

EJB3 仕様ではスキーマ検証が必要になるため、**persistence** 要素の文法定義を定義してください。systemId が **persistence_1_0.xsd** で終わる場合、Hibernate entityManager は hibernate-entitymanager.jar に組み込まれたバージョンを使用します。インターネットアクセスは実行されません。

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
   version="1.0">
```

2.2.2. ブートストラップ

EJB3 仕様は **EntityManagerFactory** と **EntityManager** にアクセスするブートストラップ手順を定義します。ブートストラップクラスは **javax.persistence.Persistence** などです。

```
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("manager1");
//or
Map configOverrides = new HashMap();
configOverrides.put("hibernate.hbm2ddl.auto", "create-drop");
EntityManagerFactory programmaticEmf =
    Persistence.createEntityManagerFactory("manager1", configOverrides);
```

最初のバージョンはマップが空白な 2 つ目のバージョンと同等です。マップバージョンは persistence.xml ファイルで定義されたすべてのプロパティよりも優先されるオーバーライドセットです。マップで使用できる EJB3 プロパティが2つ存在します。

- 使用されるプロバイダクラスを定義する javax.persistence.provider
- 使用されるトランザクションタイプを定義する javax.persistence.transactionType (JTA または RESOURCE_LOCAL のいずれか)
- JNDI で JTA データソース名を定義する javax.persistence.jtaDataSource
- JNDI の JTA データソース名を定義する javax.persistence.nonJtaDataSource

Persistence.createEntityManagerFactory() が呼び出されると、永続実装が **ClassLoader.getResource("META-INF/persistence.xml")** メソッドを使用してクラスパスを検索し、**META-INF/persistence.xml** ファイルを探します。実際には、**Persistence** クラスはクラスパスで利用可能なすべての永続プロバイダを参照し、エンティティマネージャファクトリ **manager1** の作成を行うかどうかをそれぞれの永続プロバイダに尋ねます。各プロバイダから利用可能なリソースのリストから、永続実装は **persistence.xml** の名前がコマンドラインで指定された名前

と一致するエンティティマネージャを検索します (プロバイダ **element** は現在の永続プロバイダに一致する必要があります)。現在の名前を持つ persistence.xml が見つからない場合や予期された永続プロバイダが見つからない場合は、**PersistenceException** が発生します。

Hibernate システムレベル設定以外に、Hibernate で利用可能なすべてのプロパティを persistence.xml ファイルの **properties** 要素で設定できます。または、マップのオーバーライドとして、**createEntityManagerFactory()** に渡します。完全なリストについては、Hibernate リファレンスドキュメンテーションを参照してください。ただし、EJB3 プロバイダでは 2 つのプロパティだけが存在します。

表2.1 Hibernate Entity Manager 固有のプロパティ

プロパティ名	定義
hibernate.ejb.classcache.<classname>	キャッシュなしに対するクラス Default のクラスキャッシュ方針 [カンマキャッシュリージョン]、fully.qualified.classname に対するデフォルトのリージョンキャッシュ (hibernate.ejb.classcache.com.acme.Cat read-write または hibernate.ejb.classcache.com.acme.Cat read-write、MyRegion).
hibernate.ejb.collectioncache.<collectionrole>	キャッシュなしに対するコレクションキャッシュ方針 [カンマキャッシュリージョン]、fully.qualified.classname.role に対するデフォルトのリージョンキャッシュ (hibernate.ejb.classcache.com.acme.Cat read-write または hibernate.ejb.classcache.com.acme.Cat read-write、MyRegion)。
hibernate.ejb.cfgfile	Hibernate を設定するために使用する XML 設定ファイル (**/hibernate.cfg.xml** など)。
hibernate.archive. autodetection	.par アーカイブの解析中に Hibernate Entity Manager により自動検出されたエレメントを決定します (デフォルトで **class,hbm** に設定されます)。
hibernate.ejb. interceptor	オプションの Hibernate インターセプタ。インターセプタインスタンスはすべての **Session** インスタンスにより共有されます。このインターセプタは **org.hibernate.Interceptor** を実装する必要があり、no-arg コンストラクタを持ちます。このプロパティを **hibernate.ejb.interceptor.session_scoped** と組み合わせることはできません。
hibernate.ejb.interceptor.session_scoped	オプションの Hibernate インターセプタ。インターセプタインスタンスは該当する **Session** インスタンスに固有です (したがって、非スレッドセーフの場合があります)。このインターセプタは **org.hibernate.Interceptor** を実装し、no-arg コンストラクタを持つ必要があります。このプロパティを **hibernate.ejb.interceptor** と組み合わせることはできません。
hibernate.ejb.naming_strategy	オプションの命名方針。使用されるデフォルトの命名方針は **EJB3NamingStrategy** です。また、**DefaultComponentSafeNamingStrategy** を使用することもできます。

プロパティ名	定義
hibernate.ejb.event. <eventtype>	該当するイベントタイプのイベントリスナーリスト。イベントリスナーのリストはカンマで区切られた完全修飾クラス名リストです (たとえば、hibernate.ejb.event.pre-load com.acme.SecurityListener, com.acme.AuditListener)。
hibernate.ejb. use_class_enhancer	デプロイメント時にアプリケーションサーバークラス拡張を使用するかどうか (デフォルトでは false に設定されます)
hibernate.ejb. discard_pc_on_close	true の場合は、永続コンテキストが破棄されます (clear() が呼び出されたとき)。それ以外の場合は、トランザクションが完了するまで永続コンテキストが有効なままになります。すべてのオブジェクトは管理され、すべての変更はデータベースと同期されます (デフォルトでは false に設定されます。つまり、トランザクションの完了を待機します)。

同じ設定で XML **<class>** 宣言と **hibernate.ejb.cfgfile** を同時に使用できないことに注意してください (競合が発生する可能性があります)。**persistence.xml** に設定されたプロパティは **hibernate.cfg.xml** に設定されたプロパティをオーバーライドします。

注記

hibernate.transaction.factory_class をオーバーライドしないでください。Hibernate EntityManager は EntityManager タイプに応じて適切なトランザクションファクトリを自動的に設定します (**JTA** と **RESOURSE_LOCAL**)。Java EE 環境を使用する場合は、**hibernate.transaction.manager_lookup_class** を設定できます。

J2SE 環境の典型的な設定は以下のとおりです。

```
<persistence>
   <persistence-unit name="manager1" transaction-type="RESOURCE_LOCAL">
      <class>org.hibernate.ejb.test.Cat</class>
      <class>org.hibernate.ejb.test.Distributor</class>
      <class>org.hibernate.ejb.test.Item</class>
      <properties>
         <property name="hibernate.dialect"
value="org.hibernate.dialect.HSQLDialect"/>
         <property name="hibernate.connection.driver_class"
value="org.hsqldb.jdbcDriver"/>
         <property name="hibernate.connection.username" value="sa"/>
         <property name="hibernate.connection.password" value=""/>
         <property name="hibernate.connection.url" value="jdbc:hsqldb:."/>
         <property name="hibernate.max_fetch_depth" value="3"/>

         <!-- cache configuration -->
         <property
name="hibernate.ejb.classcache.org.hibernate.ejb.test.Item" value="read-
write"/>
         <property
name="hibernate.ejb.collectioncache.org.hibernate.ejb.test.Item.distributo
rs" value="read-write, RegionName"/>
```

```
         <!-- alternatively to <class> and <property> declarations, you
can use a regular hibernate.cfg.xml file -->
         <!-- property name="hibernate.ejb.cfgfile"
value="/org/hibernate/ejb/test/hibernate.cfg.xml"/ -->
      </properties>
   </persistence-unit>
</persistence>
```

プログラムによる設定を簡単にするために、Hibernate Entity Manager は商用の API を提供します。この API は **`Configuration`** API に非常に似ており、同じコンセプトを共有します (**`Ejb3Configuration`**)。この使用方法の詳細については、JavaDoc とHibernate リファレンスガイドを参照してください。

```
Ejb3Configuration cfg = new Ejb3Configuration();
EntityManagerFactory emf =
  cfg.addProperties( properties ) //add some properties
     .setInterceptor( myInterceptorImpl ) // set an interceptor
     .addAnnotatedClass( MyAnnotatedClass.class ) //add a class to be
mapped
     .addClass( NonAnnotatedClass.class ) //add an hbm.xml file using the
Hibernate convention
     .addResource( "mypath/MyOtherCLass.hbm.xml" ) //add an hbm.xml file
     .addResource( "mypath/orm.xml" ) //add an EJB3 deployment descriptor
     .configure("/mypath/hibernate.cfg.xml") //add a regular
hibernate.cfg.xml
     .buildEntityManagerFactory(); //Create the entity manager factory
```

2.3. イベントリスナー

Hibernate Entity Manager はすべての EJB3 セマンティクスを実装するために Hibernate コアを拡張する必要があります。これは、Hibernate のイベントリスナーシステムにより実現されます。イベントシステムを使用する場合は、いくつかの EJB3 セマンティクスをオーバーライドする可能性があるので注意してください。安全な方法は、以下に示されたリストにイベントリスナーを追加することです。

表2.2 Hibernate Entity Manager デフォルトイベントリスナー

イベント	リスナー
flush	org.hibernate.ejb.event.EJB3FlushEventListener
auto-flush	org.hibernate.ejb.event.EJB3AutoFlushEventListener
delete	org.hibernate.ejb.event.EJB3DeleteEventListener
flush-entity	org.hibernate.ejb.event.EJB3FlushEntityEventListener
merge	org.hibernate.ejb.event.EJB3MergeEventListener
create	org.hibernate.ejb.event.EJB3PersistEventListener
create-onflush	org.hibernate.ejb.event.EJB3PersistOnFlushEventListener

イベント	リスナー
save	org.hibernate.ejb.event.EJB3SaveEventListener
save-update	org.hibernate.ejb.event.EJB3SaveOrUpdateEventListener
pre-insert	org.hibernate.secure.JACCPreInsertEventListener, org.hibernate.valitator.event.ValidateEventListener
pre-insert	org.hibernate.secure.JACCPreUpdateEventListener, org.hibernate.valitator.event.ValidateEventListener
pre-delete	org.hibernate.secure.JACCPreDeleteEventListener
pre-load	org.hibernate.secure.JACCPreLoadEventListener
post-delete	org.hibernate.ejb.event.EJB3PostDeleteEventListener
post-insert	org.hibernate.ejb.event.EJB3PostInsertEventListener
post-load	org.hibernate.ejb.event.EJB3PostLoadEventListener
post-update	org.hibernate.ejb.event.EJB3PostUpdateEventListener

セキュリティが有効でない場合は、JACC*EventListeners が削除されることに注意してください。

イベントリスナーは、プロパティ (「構成とブートストラップ」 を参照) または **`Ejb3Configuration.getEventListeners()`** API を使用して設定できます。

2.4. JAVA SE 環境での ENTITYMANAGER の取得

エンティティマネージャファクトリは、変更不可の設定ホルダーと見なす必要があります。単一のデータソースを参照し、定義された一連のエンティティをマップするよう定義されます。これは、**`EntityManager`** を作成および管理するエンティティポイントです。**`Persistence`** クラスはエンティティマネージャファクトリを作成するブートストラップクラスです。

```
// Use persistence.xml configuration
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("manager1");
EntityManager em = emf.createEntityManager(); // Retrieve an application
managed entity manager
// Work with the EM
em.close();
...
emf.close(); //close at application end
```

エンティティマネージャファクトリは通常アプリケーション初期化時に作成され、アプリケーション終了時に閉じられます。この作成はコストがかかるプロセスです。Hibernate の使用経験があるユーザーにとっては、エンティティマネージャファクトリはセッションファクトリに非常に似ています。実際に

は、エンティティマネージャファクトリはセッションファクトリの上部のラッパーです。entityManagerFactory への呼出は、スレッドセーフです。

EntityManagerFactory により、拡張されたエンティティマネージャを取得できます。拡張されたエンティティマネージャはエンティティマネージャのライフタイムの間、同じ永続コンテキストを保持します。つまり、エンティティは 2 つのトランザクション間で管理されます (間に entityManager.clear() を呼び出さない限り)。エンティティマネージャは Hibernate セッション上部の小さなラッパーと見なすことができます。

2.5. その他

Hibernate Entity Manager は、カスタマイズなしにHibernate Validator が構成されており、ご自身でイベントをオーバーライドする必要がありません。ドメインモデルにHibernate Validator アノテーションを使わない場合、パフォーマンスコストは発生しません。Hibernate Validatorの詳細情報については、Hibernate Annotations リファレンスガイドを参照してください。

第3章 オブジェクトの使用

3.1. エンティティステータス

Hibernate の場合と同様に (対応する用語は括弧で囲まれています)、エンティティインスタンスは以下のいずれかのステータスを持ちます。

- 新規 (一時): **new** 演算子を使用してエンティティがインスタンス化された場合にエンティティは新規になります。エンティティは永続コンテキストに関連付けられません。データベースには永続表現がなく、ID 値が割り当てられません。
- 管理対象 (永続): 管理対象エンティティインスタンスは、永続コンテキストに現在関連付けられている永続 ID を持つインスタンスです。
- デタッチ済み: エンティティインスタンスは、永続コンテキストに関連付けられなくなった永続 ID を持つインスタンスです。通常、これは永続コンテキストが閉じられたか、インスタンスがコンテキストから除外されたが理由です。
- 削除済み: 削除済みエンティティインスタンスは、永続コンテキストに関連付けられ、データベースから削除するようスケジュールされた永続 ID を持つインスタンスです。

EntityManager API を使用すると、オブジェクトをロードおよび保存するためにエンティティのステータスを変更できます。SQL ステートメントの管理ではなくオブジェクトステータス管理について考える場合に、EJB3 での永続化を理解しやすくなります。

3.2. オブジェクトの永続化

新しいエンティティインスタンスを作成したら (一般的な **new** 演算子を使用)、エンティティインスタンスのステータスは **new** になります。これは、エンティティマネージャに関連付けることによって永続化できます。

```
DomesticCat fritz = new DomesticCat();
fritz.setColor(Color.GINGER);
fritz.setSex('M');
fritz.setName("Fritz");
em.persist(fritz);
```

DomesticCat エンティティタイプが生成された ID を持つ場合、値は **persist()** が呼び出されたときにインスタンスに関連付けられます。ID が自動的に生成されない場合は、**persist()** が呼び出される前にアプリケーションにより割り当てられた (通常は自然な) キー値をインスタンスに設定する必要があります。

3.3. オブジェクトのロード

エンティティマネージャの **find()** メソッドを使用して ID 値別にエンティティインスタンスをロードします。

```
cat = em.find(Cat.class, catId);

// You may need to wrap the primitive identifiers
long catId = 1234;
em.find( Cat.class, new Long(catId) );
```

場合によっては、オブジェクトステータスをロードせずに、参照だけしたい (プロキシ) 場合があります。この参照は、**getReference()** メソッドを使用して取得できます。これは、親をロードせずに親に子をリンクする場合に特に役に立ちます。

```
child = new Child();
child.SetName("Henry");
Parent parent = em.getReference(Parent.class, parentId); //no query to the
DB
child.setParent(parent);
em.persist(child);
```

em.refresh() 操作を使用してエンティティインスタンスとそのコレクションをいつでもリロードできます。これは、エンティティの一部のプロパティを初期化するためにデータベーストリガが使用される場合に役に立ちます。関係のカスケードスタイルとして **REFRESH** を指定しない限り、エンティティインスタンスとそのコレクションのみが更新されます。

```
em.persist(cat);
em.flush(); // force the SQL insert and triggers to run
em.refresh(cat); //re-read the state (after the trigger executes)
```

3.4. オブジェクトのクエリ

探しているオブジェクトの 識別子の値を知らない場合は、クエリが必要です。Hibernate EntityManager 実装は、使いやすく強力なオブジェクト指向クエリ言語 (EJB3-QL) をサポートします (EJB3-QL は HQL の影響を受けています。あるいは HQL は EJB3-QL の影響を受けています)。両方のクエリ言語はデータベース間で移植可能であり、(テーブル名とカラム名の代わりに) エンティティ名とプロパティ名を 識別子として使用します。また、クエリをデータベースのネイティブ SQL で記述することもできます (Java ビジネスオブジェクトへの結果セットの変換に対する EJB3のオプションのサポートあり)。

3.4.1. クエリの実行

EJB3QL クエリと SQL クエリは **javax.persistence.Query** のインスタンスにより表されます。このインターフェースは、パラメータバインディング、結果セット処理、およびクエリの実行に関するメソッドを提供します。クエリは、常に現在のエンティティマネージャを使用して作成されます。

クエリは、通常 **getResultList()** を呼び出すことによって実行されます。このメソッドはクエリの結果となるインスタンスをメモリに完全にロードします。クエリにより取得されたエンティティインスタンスは、永続の状態になります。クエリが単一オブジェクトだけを返すことがわかっている場合は、**getSingleResult()** メソッドを使用することにより作業を短縮できます。

3.4.1.1. プロジェクション

プロジェクションが使用された場合、EJB3QL クエリはオブジェクトの組を返すことができます。各組はオブジェクトアレイとして返されます。

```
Iterator<Cat[]> kittensAndMothers =
  em.createQuery("select kitten, mother from Cat kitten join kitten.mother
mother").getResultList().iterator();
  while (kittensAndMothers.hasNext()) {
     Cat[] tuple = kittensAndMothers.next();
     Cat kitten = tuple[0];
     Cat mother = tuple[1];
```

```
  }
    ....
}
```

3.4.1.2. スカラー結果

クエリは、select 句でエンティティエイリアスの代わりにエンティティの特定のプロパティを指定できます。SQL 集計関数も呼び出すことができます。返された非トランザクションオブジェクトまたは集計結果は、「スカラー」結果と見なされ、永続状態のエンティティではありません (つまり、「読み取り専用」と見なされます)。

```
Iterator<Object[]> results = em.createQuery(
        "select cat.color, min(cat.birthdate), count(cat) from Cat cat " +
        "group by cat.color")
        .getResultList()
        .iterator();

while ( results.hasNext() ) {
    Object[] row = results.next();
    Color type = (Color) row[0];
    Date oldest = (Date) row[1];
    Integer count = (Integer) row[2];
    .....
}
```

3.4.1.3. バインドパラメータ

名前付きおよび位置クエリパラメータの両方がサポートされます。**Query** API は引数をバインドするメソッドをいくつか提供します。EJB3 仕様では、位置パラメータが 1 から数えられます。名前付きパラメータはクエリ文字列の **:paramname** という形式の ID です。(名前付きパラメータは堅牢であり、理解しやすいため)名前付きパラメータが推奨されます。

```
// Named parameter (preferred)
Query q = em.createQuery("select cat from DomesticCat cat where cat.name =
:name");
q.setParameter("name", "Fritz");
List cats = q.getResultList();

// Positional parameter
Query q = em.createQuery("select cat from DomesticCat cat where cat.name =
?1");
q.setParameter(1, "Izi");
List cats = q.getResultList();

// Named parameter list
List names = new ArrayList();
names.add("Izi");
names.add("Fritz");
Query q = em.createQuery("select cat from DomesticCat cat where cat.name
in (:namesList)");
q.setParameter("namesList", names);
List cats = q.getResultList();
```

3.4.1.4. Pagination

結果セットに境界 (取得したい行の最大数または取得したい最初の行) を指定する必要がある場合は、以下のメソッドを使用してください。

```
Query q = em.createQuery("select cat from DomesticCat cat");
q.setFirstResult(20);
q.setMaxResults(10);
List cats = q.getResultList(); //return cats from the 20th position to 29th
```

Hibernate は、この制限クエリを DBMS のネイティブ SQL に変換する方法を知っています。

3.4.1.5. 名前付きクエリの外部化

アノテーションを使用して名前付きクエリを定義することもできます。

```
@javax.persistence.NamedQuery(name="eg.DomesticCat.by.name.and.minimum.wei
ght",
  query="select cat from eg.DomesticCat as cat  where cat.name = ?1 and
cat.weight > ?2")
```

パラメータは、実行される前に名前付きクエリにプログラムによりバインドされます。

```
Query q =
em.createNamedQuery("eg.DomesticCat.by.name.and.minimum.weight");
q.setParameter(1, name);
q.setParameter(2, minWeight);
List cats = q.getResultList();
```

実際のプログラムコードは使用されるクエリ言語とは独立し、XML マッピングファイルに配置することによりメタデータのネイティブ SQL クエリを定義したり、Hibernate のネイティブ機能を使用したりできます。

3.4.1.6. ネイティブクエリ

`createNativeQuery()` を使用して SQL でクエリを表記できます。Hibernate では、JDBC 結果セットとビジネスオブジェクトのマッピングが考慮されます。**`@SqlResultSetMapping`** (SQL 結果セットマッピングのマップ方法については、Hibernate Annotations リファレンスドキュメンテーションを参照) またはエンティティマッピング (クエリ結果のカラム名がエンティティマッピングで宣言された名前と同じである場合。このメカニズムが動作するにはすべてのエンティティカラムが返される必要があります)。

```
@SqlResultSetMapping(name="getItem", entities =
        @EntityResult(entityClass=org.hibernate.ejb.test.Item.class,
fields= {
            @FieldResult(name="name", column="itemname"),
            @FieldResult(name="descr", column="itemdescription")
        })
)

Query q = em.createNativeQuery("select name as itemname, descr as
itemdescription from Item", "getItem");
item = (Item) q.getSingleResult(); //from a resultset
```

```
Query q = em.createNativeQuery("select * from Item", Item.class);
item = (Item) q.getSingleResult(); //from a class columns names match the
mapping
```

注記

名前付きクエリのスカラーサポートに関する詳細については、Hibernate Annotations ドキュメンテーションを参照してください。

3.4.1.7. クエリヒント

クエリヒント (通常はパフォーマンス最適化が目的) は実装固有です。ヒントは **`query.setHint(String name, Object value)`** メソッドまたは **`@Named(Native)Query(hints)`** アノテーションを使用して宣言されます。これらは SQL クエリヒントでないことに注意してください。Hibernate JEB3 実装は以下のクエリヒントを提供します。

表3.1 Hibernate クエリヒント

ヒント	定義
org.hibernate.timeout	秒単位のクエリタイムアウト (new Integer(10) など)
org.hibernate.fetchSize	ラウンドトリップごとに JDBC ドライバにより フェッチされる行数 (new Integer(50) など)
org.hibernate.comment	SQL クエリにコメントを追加します。DBA の場合に役に立ちます (new String("fetch all orders in 1 statement") など)
org.hibernate.cacheable	クエリがキャッシュ可能であるかどうか (new Boolean(true) など)。デフォルト値は false
org.hibernate.cacheMode	このクエリのキャッシュモードをオーバーライドします (CacheMode.REFRESH など)
org.hibernate.cacheRegion	このクエリのキャッシュリージョン (new String("regionName") など)
org.hibernate.readOnly	このクエリにより取得されたエンティティは読み取り専用モードでロードされ、Hibernate によりエンティティがダーティチェックされないか、変更が永続化されません (new Boolean(true) など)。デフォルト値は false です
org.hibernate.flushMode	このクエリに使用されるフラッシュモード
org.hibernate.cacheMode	このクエリに使用されるキャッシュモード

値オブジェクトはネイティブタイプまたは文字列同等を受けいれます (**CaheMode.REFRESH** or 「**REFRESH**」 など)。詳細については、Hibernate リファレンスドキュメンテーションを参照してください。

3.5. 永続オブジェクト

トランザクション管理インスタンス (つまり、エンティティマネージャによりロード、保存、作成、または問い合わされたオブジェクト) は、アプリケーションにより操作され、永続ステータスの変更は、エンティティマネージャがフラッシュされるときに保持されます (この章の後半で説明)。変更を保持するために特定のメソッドを呼び出す必要はありません。エンティティインスタンスのステータスを更新する簡単な方法は **find()** を使用し、直接操作することです (永続コンテキストはオープンのまま)。

```
Cat cat = em.find( Cat.class, new Long(69) );
cat.setName("PK");
em.flush();  // changes to cat are automatically detected and persisted
```

このプログラミングモデルは SQL SELECT (オブジェクトをロードする) と SQL UPDATE (更新されたステータスを保持する) が同じセッションで必要であるため、非効率です。したがって、Hibernate はデタッチ済みのインスタンスを使用する別の方法を提供します。

3.6. デタッチ済みのオブジェクトの変更

多くのアプリケーションは、 1 つのトランザクションで 1 つのオブジェクトを取得し、操作のためにプレゼンテーション層に送信し、新しいトランザクションで変更を保存する必要があります。ユーザーがトランザクションとトランザクションの間に考える場合は、待機時間が非常に長くなることがあります。高同時実行環境でこのような方法を使用するアプリケーションは、通常「長い」単位の作業を分離するために、バージョン管理されたデータを使用します。

EJB3 仕様では、**EntityManager.merge()** メソッドを使用してデタッチ済みのインスタンスに行われる変更の永続化を提供することにより、この開発モデルがサポートされます。

```
// in the first entity manager
Cat cat = firstEntityManager.find(Cat.class, catId);
Cat potentialMate = new Cat();
firstEntityManager.persist(potentialMate);

// in a higher layer of the application
cat.setMate(potentialMate);

// later, in a new entity manager
secondEntityManager.merge(cat);  // update cat
secondEntityManager.merge(mate); // update mate
```

merge() メソッドは、永続コンテキストのステータスを考慮せずにデタッチ済みのインスタンスに行われた変更を対応する管理対象インスタンスにマージします。つまり、マージされたオブジェクトステータスは、永続エンティティステータスをオーバーライドします (すでに存在する場合)。アプリケーションは、該当するデタッチ済みのインスタンスから到達可能なデタッチ済みのインスタンスに対して **merge()** を個別に使用します (ステータスを保持したい場合)。これは、推移の永続化を使用して関連するエンティティとコレクションにカスケードできます (「遷移の永続化」 を参照)。

3.7. 自動ステータス検出

マージ操作はデタッチ済みのインスタンスのマージにより挿入または更新が必要になるかどうかを自動的に検出します。つまり、新しいインスタンス (およびデタッチ済みのインスタンス) を **merge()** に渡すことについて心配する必要はありません。この作業はエンティティマネージャが行います。

```
// In the first entity manager
Cat cat = firstEntityManager.find(Cat.class, catID);

// In a higher layer of the application, detached
Cat mate = new Cat();
cat.setMate(mate);

// Later, in a new entity manager
secondEntityManager.merge(cat);   // update existing state
secondEntityManager.merge(mate);  // save the new instance
```

新しいユーザーは **merge()** の使用方法とセマンティクスについて混乱するかもしれません。別の新しいエンティティマネージャの 1 つのエンティティマネージャでロードされたオブジェクトステータスを使用しない限り、**merge()** を使用しないでください。一部のアプリケーションはこのメソッドを使用しません。

通常、**merge()** は次のシナリオで使用されます。

- アプリケーションは最初のエンティティマネージャでオブジェクトをロードします
- オブジェクトはプレゼンテーション層に渡されます
- いくつかの変更がオブジェクトに行われます
- オブジェクトがビジネスロジック層に渡されます
- アプリケーションはセカンダリエンティティマネージャで **merge()** を呼び出すことによりこれらの変更を永続化します

merge() の適切なセマンティックは以下のとおりです。

- 永続コンテキストで現在関連付けられたのと同じ ID を持つ管理対象インスタンスが存在する場合は、該当するオブジェクトのステータスを管理対象インスタンスにコピーします。
- 永続コンテキストに現在関連付けれた管理対象インスタンスが存在しない場合は、データベースからロードするか、新しい管理対象インスタンスを作成します。
- 管理対象インスタンスが返されます
- 該当するインスタンスが永続コンテキストに関連付けられてない場合、コンテキストはデタッチ済みのままになり、通常は破棄されます

注記

EJB3 でのマージはネイティブ Hibernate での **saveOrUpdateCopy()** メソッドに似ています。ただし、**saveOrUpdate()** メソッドとは同じではありません。該当するインスタンスは永続コンテキストに再びアタッチされず、管理対象インスタンスが **merge()** メソッドにより返されます。

3.8. 管理対象オブジェクトの削除

EntityManager.remove() はデータベースからオブジェクトステータスを削除します。アプリケーションは削除されたオブジェクトの参照を引き続き保持する場合があります。**remove()** は永続インスタンスを再度新しくする (一時的) と考えることができます。インスタンスはデタッチされず、マージにより挿入が行われます。

3.9. 永続コンテキストのフラッシュ

3.9.1. トランザクション内

場合によっては、エンティティマネージャはメモリに保持されたオブジェクトのステータスでデータストアを同期するのに必要な SQL DML ステートメントを実行します。このプロセスであるフラッシュは以下の時点でデフォルトで実行されます (これは Hibernate 固有であり、仕様では定義されません)。

- クエリ実行前*
- **javax.persistence.EntityTransaction.commit()*** から
- **EntityManager.flush()** が呼び出されたとき

(*) トランザクションがアクティブな場合

SQL ステートメントは以下の順で発行されます。

- すべてのエンティティ挿入 (**EntityManager.persist()** を使用して対応するオブジェクトが保存されたのと同じ順序)
- すべてのエンティティ更新
- すべてのコレクション削除
- すべてのコレクションエレメント削除、更新、および挿入
- すべてのコレクション挿入
- すべてのエンティティ削除 (**EntityManager.remove()** を使用して対応するオブジェクトが削除されたのと同じ順序)

(例外: アプリケーションにより割り当てられた ID を使用するエンティティインスタンスが保存時に挿入されます)

flush() を明示的に使用する場合を除き、エンティティマネージャが JDBC コールを実行するタイミングに関して絶対的な保証はありません。実行順序のみ保証されます。ただし、Hibernate は **Query.getResultList()**/**Query.getSingleResult()** が無効なデータを返さず、アクティブなトランザクションで実行された場合に間違ったデータを返さないことを保証します。

フラッシュの頻度が少なくなるようにデフォルトの動作を変更できます。エンティティマネージャの **FlushModeType** は 2 つの異なるモードを定義します (コミット時のフラッシュまたは **flush()** を明示的に呼び出さない場合の説明されたルーチンを使用した自動フラッシュ)。

```
em = emf.createEntityManager();
em.getTransaction().begin();
em.setFlushMode(FlushModeType.COMMIT); // allow queries to return stale
state

Cat izi = em.find(Cat.class, id);
```

```
izi.setName(iznizi);

// might return stale data
em.createQuery("from Cat as cat left outer join cat.kittens
kitten").getResultList();

// change to izi is not flushed!
...
em.getTransaction().commit(); // flush occurs
```

フラッシュ中に、例外が発生することがあります (例：DML 操作が制約を違反した場合)。

Hibernate は EJB3 仕様で説明されたものよりも多くのフラッシュモードを提供します。詳細については、Hibernate Core リファレンスドキュメンテーションを参照してください。

3.9.2. トランザクション外

EXTENDED 永続コンテキストでは、エンティティマネージャのすべての読み取り専用操作をトランザクション外で実行できます (**find()**、**getReference()**、**refresh()**、および読み取りクエリ)。一部の変更操作をトランザクション外で実行できますが、これらの操作は永続コンテキストがトランザクションに参加するまでキューに格納されます。これは、**persist()**、**merge()**、**remove()** の場合に該当します。一部の操作はトランザクション外で呼び出すことができません (**flush()**、**lock()**、および更新/削除クエリ)。

3.10. 遷移の永続化

個々のオブジェクトを保存、削除、または再アタッチすることは非常に面倒です (特に、関連付けられたオブジェクトのグラフを扱う場合)。典型的なケースは親子関係です。以下の例を考えてください。

親子関係の子が値タイプ (たとえば、アドレスまたは文字列のコレクション) である場合、ライフサイクルは親に依存しステータス変更の便利な「カスケード」に対して何も行う必要がありません。親が永続化された場合、値タイプの子オブジェクトも永続化されます。親が削除された場合は、子が削除されます。これは、コレクションからの子の削除などの操作にも該当します。Hibernate はこれを検出し、値タイプオブジェクトが参照を共有できるため、データベースから子を削除します。

親および子オブジェクトがエンティティであり、値タイプでない同じシナリオを考えてください (たとえば、カテゴリとアイテム、または親猫と子猫)。エンティティは独自のライフサイクルを持ち、共有された参照をサポートし (したがって、コレクションからのエンティティの削除は、エンティティを削除できることを意味しません)、デフォルトであるエンティティから他の関連エンティティへのステータスのカスケードが行われません。EJB3 仕様では、到達可能性による永続性は必要ありません。Hibernate で最初に示されたように遷移の永続化の柔軟なモデルがサポートされます。

エンティティマネージャの各基本操作 (**persist()**、**merge()**、**remove()**、**refresh()**) に対して、対応するカスケードスタイルが存在します。カスケードスタイルの名前はそれぞれ PERSIST、MERGE、REMOVE、REFRESH です。関連エンティティ (またはエンティティのコレクション) に対して操作をカスケードする場合は、関係アノテーションにそれを示す必要があります。

```
@OneToOne(cascade=CascadeType.PERSIST)
```

カスケードオブジェクトは組み合わせることができます。

```
@OneToOne(cascade= { CascadeType.PERSIST, CascadeType.REMOVE,
CascadeType.REFRESH } )
```

CascadeType.ALL を使用してすべての操作を特定の関係に対してカスケードするよう指定することもできます。デフォルトでは操作がカスケードされないことに注意してください。

Hibernate は、ネイティブのカスケードオプションを提供します。詳細については、Hibernate Annotations マニュアルと Hibernate リファレンスガイドを参照してください。

推奨事項:

- 通常、**@ManyToOne** 関係または **@ManyToMany** 関係でカスケードを有効にすることは適切ではありません。多くの場合、カスケードは、**@OneToOne** 関係および **@OneToMany** 関係で役に立ちます。
- 子オブジェクトのライフスパンが親オブジェクトのライフスパンにより制限を受ける場合は、**CascadeType.ALL** と **org.hibernate.annotations.CascadeType.DELETE_ORPHAN** を指定して親を完全ライフサイクルオブジェクトにしてください (孤立削除のセマンティクスについては、Hibernate リファレンスガイドを参照)。
- それ以外の場合は、カスケードをまったく必要としません。ただし、同じトランザクションで親と子を処理することが多く、入力を省略したい場合は、**cascade={PERSIST, MERGE}** の使用を考えてください。これらのオプションは多対多の関係に適切です。

3.11. ロック

EJB3 のデフォルトのロックシステムは、多くの場合、楽観的ロックに基づいています (つまり、バージョンカラムを使用して並行性の問題をチェックします)。JEB3 では、並行性の保証を強化するために追加のメカニズムが定義されています。**lock(Object entity)** メソッドを使用して、該当するエンティティ (**LOCK** がカスケードされる場合は関連するエンティティを含む) に対してロックを適用できます。必要な並行性の保証に応じて、ロックモードを選択します。

- **LockMode.READ** は該当するエンティティのダーティな読み取りと繰り返し不可能な読み取りを防ぎます。
- **LockMode.WRITE** は該当するエンティティのダーティな読み取りと繰り返し不可能な読み取りを防ぎ、バージョン番号 (存在する場合) を増加させます。

第4章 トランザクションおよび並行性

Hibernate Entity Manager と並行性制御に関する最も重要な点は、非常に理解しやすいことです。Hibernate Entity Manager はロック動作を追加せずに JDBC 接続と JTA リソースを直接使用します。ユーザーは JDBC、ANSI、使用しているデータベース管理システムのトランザクション隔離の仕様について調べてみることが強く推奨されます。Hibernate Entity Manager は自動バージョン機能のみを追加しますが、メモリ内でオブジェクトをロックしたり、データベーストランザクションの隔離レベルを変更したりしません。基本的に、データベースリソースで直接 JDBC (または JTA/CMT) を使用するように Hibernate Entity Manager を使用してください。

Hibernate の並行性制御に関して、最初に **`EntityManagerFactory`** と **`EntityManager`** の粒度とデータベーストランザクションおよび長い作業単位について説明します。

この章では、特に明示的な記述がない限り、エンティティマネージャと永続コンテキストの概念はほとんど同じです。一方は API およびプログラミングオブジェクトであり、もう一方はスコープの定義です。ただし、重要な違いがあることに注意してください。永続コンテキストは通常、Java EE の JTA トランザクションにバインドされ、拡張されたエンティティマネージャを使用しない限り、永続コンテキストはトランザクション境界 (トランザクションスコープ) で開始および終了します。詳細については、「永続コンテキストスコープ」 を参照してください。

4.1. エンティティマネージャおよびトランザクションスコープ

`EntityManagerFactory` はすべてのアプリケーションスレッドで共有することを目的とした、作成にコストがかかるスレッドセーフオブジェクトです。これは通常、アプリケーション起動時に 1 度だけ作成されます。

`EntityManager` は、単一ビジネスプロセスと単一作業単位に対して一度だけ使用し、破棄すべきコストがかからない非スレッドセーフオブジェクトです。**`EntityManager`** は、必要でない限り JDBC **`Connection`** (または **`Datasource`**) を取得しません。したがって、特定の要求に対応するためにデータアクセスが必要であることがわからない場合であっても **`EntityManager`** を安全にオープンまたはクローズできます (これは、要求の傍受を使用して以下のいくつかのパターンを実装する場合に重要になります)。

ここでは、データベーストランザクションについても考える必要があります。データベースのロックの競合を削減するために、データベーストランザクションはできるだけ短くする必要があります。データベーストランザクションが長いと、同時処理ロードが高いとアプリケーションが対応できなくなります。

作業単位のスコープとは何ですか? 単一の Hibernate **`EntityManager`** で複数のデータベーストランザクションに対応できますか、またはこれはスコープの 1 対 1 関係ですか? いつ **`Session`** をオープンおよびクローズすべきですか? データベーストランザクション境界をどのように設定しますか?

4.1.1. 作業単位

最初に **entitymanager-per-operation** アンチパターンを使用しないでください。つまり、単一スレッドで各単一データベースコールに対して **`EntityManager`** をオープンおよびクローズしないでください。当然、これはデータベーストランザクションにも当てはまります。アプリケーションのデータベースコールは、計画された順序で行われます。これらのコールはアトミックな作業単位にグループ分けされます (これは、各単一 SQL ステートメント後の自動コミットがアプリケーションで役に立たないことを意味します。このモードはアドホック SQL コンソール作業で使用することを目的としています)。

マルチユーザークライアント/サーバーアプリケーションの最も一般的なパターンは **entitymanager-per-request** です。このモデルでは、クライアントからの要求はサーバー (EJB3 永続レイヤーが実行される) に送信され、新しい **`EntityManager`** がオープンされ、すべてのデータベース操作がこの作業単位で実行されます。作業が完了すると (および、クライアントの応答が準備されると)、永続コンテキス

トとエンティティマネージャがフラッシュされ、クローズされます。また、クライアント要求を処理するために単一のデータベーストランザクションも使用します。2 つの関係は 1 対 1 であり、このモデルは多くのアプリケーションに完全に適合します。

これは、Java EE 環境のデフォルトの EJB3 永続モデル (JTA バインドされた、トランザクションスコープ対象の永続コンテキスト) です。挿入 (または検索) されたエンティティマネージャは特定の JTA トランザクションの同じ永続コンテキストを共有します。EJB3 の利点は、これについて考慮しなくてもよいことであり、エンティティマネージャと、完全に直交するセッション Bean 上のトランザクションスコープの境界からデータアクセスを参照します。

難点は、EJB3 コンテナ外部の本 (および他の) 動作の実装です。**`EntityManager`** とリソースローカルトランザクションを適切に開始および終了する必要があるだけでなく、これらはデータアクセス操作に対してアクセス可能である必要があります。作業単位の境界は、理想的には要求が非 EJB3 コンテナサーバーに届いたときに (応答が送信される前 (つまり、スタンドアロンサーブレットコンテナを使用する場合は **`ServletFilter`**)) 実行されるインターセプタを使用して実装されます。**`ThreadLocal`** 変数を使用して要求を処理するスレッドに **`EntityManager`** をバインドすることをお薦めします。これにより、このスレッドで実行されるすべてのコードでアクセスが容易になります (静的変数へのアクセスと同様)。選択したデータベーストランザクション境界メカニズムに応じて、トランザクションコンテキストを **`ThreadLocal`** 変数に保持することもできます。この実装パターンは Hibernate コミュニティでは **ThreadLocal Session** および **Open Session in View** として知られています。Hibernate リファレンスドキュメンテーションに記載された **`HibernateUtil`** を簡単に拡張し、このパターンを実装できます。外部ソフトウェアは必要ありません (これは実際には重要なことではありません)。当然、使用している環境でインターセプタを実装し、設定する方法を見つける必要があります。ヒントと例については、Hibernate Web サイトを参照してください。もう一度述べますが、最初の選択肢は EJB3 コンテナ (理想的には JBoss アプリケーションサーバーなどの軽量でモジュール形式のもの) です。

4.1.2. 長い作業単位

entitymanager-per-request パターンは作業単位を設計する場合にのみ役に立つコンセプトです。多くのビジネスプロセスはデータベースアクセスでインターリーブされたユーザーとの全体の対話を必要します。Web とエンタープライズアプリケーションでは、データベーストランザクションで要求間の待機時間が長いユーザー対話に対応できません。以下の例を考えてください。

- ダイアログの第一画面が開きます。ユーザーが参照するデータは特定の **`EntityManager`** とリソースローカルトランザクションでロードされます。ユーザーはデタッチされたオブジェクトを自由に変更できます。

- ユーザーが5 分後に "Save" をクリックすると、この変更が永続的になります。また、ユーザーはこの情報を編集する唯一のユーザーとなるため、変更の競合が発生しません。

ユーザーの観点からこの作業単位は長く実行されている**アプリケーショントランザクション**と呼ばれます。アプリケーションでこれを実装するには多くの方法があります。

最初のナイーブな実装では、ユーザーが考えている間に、同時の変更を防ぎ、分離と原子性を確保するためにデータベースでロックが保持された状態で **`EntityManager`** とデータベーストランザクションがオープンのままに場合があります。当然これは、ロックの競合によりアプリケーションが複数の同時ユーザーに対応できなくなるため、アンチパターンであり、ペシミスティックなアプローチです。

当然、アプリケーショントランザクションを実装するには複数のデータベーストランザクションを使用する必要があります。この場合、ビジネスプロセスの分離の維持について、一部はアプリケーション層の責任になります。単一アプリケーショントランザクションには、通常複数のデータベーストランザクションが関係します。これは、これらのいずれかのデータベーストランザクション (少なくとも 1 つ) が更新データを保持し、他のすべてが単にデータを読み取る場合 (複数の要求/応答サイクルにわたるウィザード形式のダイアログなど) にのみアトミックになります。これは、思ったよりも簡単に実装できます (特に、EJB3 エンティティマネージャと永続コンテキスト機能を使用する場合)。

- **自動バージョン機能** - エンティティマネージャはユーザーのために自動オプティミスティック並行性制御を実行できます。ユーザーが考えている間に同時の変更が行われたかどうかを自動的に検出できます (通常は、最終リソースローカルトランザクションでのデータの更新時にタイムスタンプのバージョン番号を比較します)。

- **デタッチ済みエンティティ (Detached Entities)** - すでに説明された **entity-per-request** パターンを使用する場合は、ユーザーの考えている間にロードされたすべてのインスタンスがデタッチ済みの状態になります。エンティティマネージャでは、デタッチ済み (変更済み) の状態をマージし、変更を永続化します。このパターンは **entitymanager-per-request-with-detached-entities** と呼ばれます。自動バージョン機能は同時の変更を分離するために使用されます。

- **拡張エンティティマネージャ** - Hibernate Entity Manager は、2 つのクライアントコール間に基礎となる JDBC 接続から接続解除したり、新しいクライアント要求が発生したときに再接続したりできます。このパターンは **entitymanager-per-application-transaction** と呼ばれ、マージが不必要になります。拡張永続コンテキストはトランザクション外に行われた変更 (永続化、マージ、削除) を収集および保持します。アクティブなトランザクション内部で行われた次のクライアントコール (通常は、ユーザー対話の最後の操作) はキューに格納されたすべての変更を実行します。自動バージョン機能は同時の変更を分離するために使用されます。

entitymanager-per-request-with-detached-objects と **entitymanager-per-application-transaction** には利点と欠点があります。これらについては、この章の後半のオプティミスティック並行性制御のコンテキストで説明します。

4.1.3. オブジェクト ID の考慮

アプリケーションは 2 つの異なる永続コンテキストの同じ永続ステータスに同時にアクセスできます。ただし、管理対象クラスのインスタンスは 2 つの永続コンテキスト間で共有されません。したがって、ID には 2 つの異なる表記法があります。

データベース ID

```
foo.getId().equals( bar.getId() )
```

JVM ID

```
foo==bar
```

特定の永続コンテキスト (つまり、`EntityManager` のスコープ内) にアタッチされたオブジェクトの場合は、2 つの表記法が同じになり、データベース ID の JVM ID が Hibernate Entity Manager によって保証されます。ただし、アプリケーションが 2 つの異なる永続コンテキストの「同じ」(永続 ID) ビジネスオブジェクトに同時にアクセスできる一方で、2 つのインスタンスは実際には異なります (JVM ID)。競合はフラッシュ/コミット時に自動バージョン機能とオプティミスティックアプローチを使用して解決されます。

このアプローチでは Hibernate とデータベースで並行性について考える必要があります。単一スレッドの作業単位で ID を保証する場合はコストがかかるロックや他の同期手段を必要としないため、最適なスケーラビリティが提供されます。`EntityManager` ごとの単一スレッドを使用し続ける限り、アプリケーションはビジネスオブジェクトを同期する必要がありません。永続コンテキスト内で、アプリケーションはエンティティを比較するために `==` を安全に使用できます。

ただし、永続コンテキスト外で `==` を使用するアプリケーションは、予期しない結果をもたらす場合があります。これは、予期しない場合 (2 つのデタッチ済みインスタンスを同じ `Set` に配置した場合など) に起こることがあります。両方は同じデータベース ID を持つことができます (つまり、同じ行を表す) が、JVM ID は定義により、デタッチ済みの状態のインスタンスに対して保証されません。開発者は永続クラスの `equals()` メソッドと `hashCode()` メソッドをオーバーライドし、オブジェクトの同一性を表す独自の表記法を実装する必要があります。1 つ注意点があります。同一性を実装するためにデー

タベース ID を使用しないでください。ビジネスキーと、一意の通常は変更不可の属性の組み合わせを使用してください。データベース ID は一時エンティティが永続的になた場合に変わります (**persist()** 操作のコントラクトを参照)。一時インスタンス (通常は、デタッチ済みのインスタンスとともに使用) が **Set** で保持された場合は、ハッシュコードを変更すると、 **Set** のコントラクトが破棄されます。適切なビジネスキーの属性はデータベースプライマリキーほど安定的である必要がなく、オブジェクトが同じ **Set** に含まれる場合にのみ、安定性を保証する必要があります。この問題の詳細については、Hibernate Web サイトを参照してください。また、これは Hibernate の問題ではなく、単に Java オブジェクト ID と同一性の実装方法の問題であることに注意してください。

4.1.4. 一般的な同一性制御の問題

アンチパターン **entitymanager-per-user-session** または **entitymanager-per-application** (当然、このルールの例外はほとんどありません。たとえば、デスクトップアプリケーションでは、永続コンテキストを手動でフラッシュして entitymanager-per-application を使用できます)。以下のいくつかの問題には推奨するパターンが存在します。設計を決定する前に結果について理解してください。

- エンティティマネージャはスレッドセーフではありません。**EntityManager** インスタンスが共有される場合は、同時に動作するもの (HTTP 要求、セッション Bean、Swing ワーカーなど) により競合が発生します。Hibernate **EntityManager** を **HttpSession** (後で説明) に保持する場合は、Http セッションへのアクセスの同期を考慮してください。考慮しないと、リロードを早くクリックしたユーザーが同時に実行されている 2 つのスレッドで同じ **EntityManager** を使用することがあります。他の非スレッドセーフのセッションスコープオブジェクトの場合はこれが該当する可能性が非常に高くなります。

- Entity Manager によりスローされた例外は、データベーストランザクションをロールバックし、**EntityManager** をすぐにクローズする (詳細は後に説明) 必要があることを意味します。**EntityManager** がアプリケーションにバインドされた場合は、アプリケーションを停止する必要があります。データベーストランザクションをロールバックしても、ビジネスオブジェクトはトランザクションの開始時の状態に戻りません。つまり、データベースの状態とビジネスオブジェクトは同期されません。例外は復元可能ではなく、ロールバック後に作業単位をやり直す必要があるため、通常これは問題ではありません。

- 永続コンテキストは、管理対象状態の各オブジェクトをキャッシュします (Hibernate によりダーティ状態が監視およびチェックされる)。つまり、永続コンテキストは、長い間オープンにしたり、ロードするデータが多すぎたりする場合に、**OutOfMemoryException** を取得するまで継続して肥大化します。この問題の 1 つの解決法は、永続コンテキストの定期的なフラッシュによるバッチ処理です。ただし、大量のデータ操作が必要な場合にはデータベースのストアドプロシージャを使用することを考慮する必要があります。この問題の複数の解決法は 6 章*バッチ処理* に示されています。ユーザーセッションの間、永続コンテキストをオープンにすると、無効なデータとなる可能性が高くなります (この問題は把握し、適切に処理する必要があります)。

4.2. データベーストランザクション境界

データベース (またはシステム) トランザクション境界は、常に必要です。データベーストランザクションの外部ではデータベースとの通信が起こりません (これにより、自動コミットモードに慣れている多くの開発者が混乱することがあります)。常に、明確なトランザクション境界を (読み取り専用の操作に対しても) 使用してください。これは、分離レベルとデータベース機能によっては必要でないことがあります。ただし、トランザクションの境界を常に明示的に設定する場合、欠点は存在しません。**EXTENDED** 永続コンテキストの変更を保持する必要がある場合は、トランザクション外で操作を実行する必要があります。

EJB3 アプリケーションは非管理 (つまり、スタンドアロンで単純な Web または Swing アプリケーション) および管理 J2EE 環境で実行できます。非管理環境では、通常 **EntityManagerFactory** が独自のデータベース接続プールを担当します。アプリケーション開発者はトランザクション境界を手動で設定

する必要があります (つまり、データベーストランザクション自体を開始、コミット、またはロールバックします)。管理環境は、通常コンテナ管理トランザクションを提供します (トランザクションアセンブリは EJB セッション Bean のアノテーションを使用して定義されます)。トランザクション境界のプログラミングは不必要になり、**EntityManager** のフラッシュも自動的に実行されます。

通常、作業単位の終了には 4 つの異なるフェーズが関係します。

- (リソースローカルまたは JTA) トランザクションをコミットします (これによりエンティティマネージャと永続コンテキストが自動的にフラッシュされます)。
- エンティティマネージャを終了します (アプリケーション管理エンティティマネージャを使用している場合)。
- 例外を処理します。

トランザクション境界と管理および非管理環境での例外処理について詳しく説明します。

4.2.1. 非管理環境

EJB3永続レイヤーが非管理環境で実行される場合、データベース接続は通常目に付かない Hibernate のプールメカニズムによって処理されます。一般的なエンティティマネージャとトランザクション処理イディオムは以下のとおりです。

```
// Non-managed environment idiom
EntityManager em = emf.createEntityManager();
EntityTransaction tx = null;
try {
    tx = em.getTransaction();
    tx.begin();

    // do some work
    ...

    tx.commit();
}
catch (RuntimeException e) {
    if ( tx != null && tx.isActive() ) tx.rollback();
    throw e; // or display error message
}
finally {
    em.close();
}
```

EntityManager に対して明示的に **flush()** を実行する必要はありません。**commit()** を呼び出すと、自動的に同期が実行されます。

close() を呼び出すと、**EntityManager** が終了します。**close()** の主な問題はリソースのリリースです。常に終了し、保証された最終ブロック外部で実行しないでください。

多くの場合、通常のアプリケーションではビジネスコードにこのイディオムが使用されません。重大な (システム) 例外は常に「上部」で補足する必要があります。つまり、エンティティマネージャコール (永続レイヤー) を実行するコードと **RuntimeException** を処理 (および通常はクリーンアップを実行し、終了します) するコードは異なるレイヤーに属します。これは、設計を行う上で問題となります。できるだけ J2EE/EJB コンテナサービスを使用してください。例外処理については、この章の後半で説明します。

4.2.1.1. EntityTransaction

JTA 環境では、環境のトランザクションと対話するのに特別な API を必要としません。トランザクション宣言または JTA API を使用します。

RESOURCE_LOCAL エンティティマネージャを使用する場合は、**EntityTransaction** API を使用してトランザクション境界を設定する必要があります。**EntityTransaction** は、**entityManager.getTransaction()** を使用して取得できます。この **EntityTransaction** API は通常の **begin()** メソッド、**commit()** メソッド、**rollback()** メソッド、および **isActive()** メソッドを提供します。また、トランザクションをロールバックとマークすることもできます (つまり、トランザクションを強制的にロールバックできます)。これは、JTA 操作 **setRollbackOnly()** に非常に似ています。**commit()** 操作が失敗した場合やトランザクションが **setRollbackOnly()** とマークされた場合は、**commit()** メソッドがトランザクションをロールバックしようとし、**javax.transaction.RollbackException** を発生させます。

JTA エンティティマネージャでは、**entityManager.getTransaction()** コールが許可されません。

4.2.2. JTA の使用

永続レイヤーがアプリケーションサーバーで実行される場合 (たとえば、EJB3 セッション Bean の背後で)、エンティティマネージャにより内部的に取得された各データソース接続は自動的にグローバルな JTA トランザクションの一部になります。Hibernate はこの統合に対して 2 つの方針を提供します。

Bean 管理トランザクション (BMT) を使用する場合、コードは以下のようになります。

```
// BMT idiom
@Resource public UserTransaction utx;
@Resource public EntityManagerFactory factory;

public void doBusiness() {
    EntityManager em = factory.createEntityManager();
    try {

    // do some work
    ...

    utx.commit();
}
catch (RuntimeException e) {
    if (utx != null) utx.rollback();
    throw e; // or display error message
}
finally {
    em.close();
}
```

EJB3 コンテナ内の CMT (Container Managed Transactions) を使用して、トランザクション境界は、プログラムではなく、セッション Bean アノテーションまたは配備記述子で設定されます。**EntityManager** はトランザクション完了時に自動的にフラッシュされます (**EntityManager** を挿入または検索した場合は、自動的に終了します)。**EntityManager** の使用中に例外が発生した場合は、例外を補足しないときにトランザクションロールバックが自動的に実行されます。**EntityManager** 例外は **RuntimeException** であるため、EJB 仕様 (システム例外とアプリケーション例外) ごとにトランザクションがロールバックされます。

Hibernate EntityManager で **hibernate.transaction.factory_class** を定義することが重要です (つまり、この値をオーバーライドしない)。**org.hibernate.transaction.manager_lookup_class** も設定することに注意してください。

CMT 環境を使用する場合は、コードの異なる部分で同じエンティティマネージャーを使用することもできます。通常は、非管理環境で、**ThreadLocal** 変数を使用してエンティティマネージャーを保持しますが、単一の EJB 要求は異なるスレッド (たとえば、別のセッション Bean を呼び出すセッション) で実行できます。EJB3 コンテナはユーザーのために永続コンテキストを伝播します。投入または検索を使用して、EJB3 コンテナは JTA コンテキスト (存在する場合) にバインドされた同じ永続コンテキストでエンティティマネージャーを返すか、新しいエンティティマネージャを作成し、バインドします (「永続コンテキスト伝播」 を参照)。

CMT および EJB3コンテナに使用するエンティティマネージャ/トランザクション管理イディオムは以下のように短縮されました。

```
//CMT idiom through injection
@PersistenceContext(name="sample") EntityManager em;
```

つまり、管理対象環境で行うべきことは、**EntityManager** を挿入し、データアクセスを行い、他のことはコンテナに任せることです。トランザクション境界はセッション Bean のアノテーションとデプロイメント記述子で宣言して設定されます。エンティティマネージャと永続コンテキストのライフサイクルはコンテナによって完全に管理されます。

特定の Hibernate ネイティブ API を使用する場合は、**after_statement** 接続リリースモードに注意する必要があります。JTA 仕様の制限により、Hibernate が終了していない **ScrollableResults** や **scroll()** または **iterate()** によって返された **Iterator** インスタンスを自動的にクリーンアップすることはできません。**ScrollableResults.close()** または **Hibernate.close(Iterator)** を **finally** ブロックから明示的に呼び出すことにより基礎となるデータベースカーソルをリリースする必要があります (当然、ほとんどのアプリケーションでは CMT コードから **scroll()** または **iterate()** を使用することを簡単に回避できます)。

4.2.3. 例外処理

EntityManager が例外 (**SQLException** を含む) をスローした場合は、データベーストランザクションをすぐにロールバックし、**EntityManager.close()** を呼び出し (**createEntityManager()** が呼び出された場合)、**EntityManager** インスタンスを破棄する必要があります。**EntityManager** の一部のメソッドでは、永続コンテキストが整合性がある状態に保たれません。エンティティマネージャによりスローされた例外は復元可能として処理できません。**finally** ブロックで **close()** を呼び出し、**EntityManager** が終了するようにしてください。コンテナが管理するエンティティマネージャはユーザーのためにこれを行います。RuntimeException がコンテナに伝播するようにします。

Hibernate エンティティマネージャは、通常 Hibernate コア例外をカプセル化する例外を発生させます。**EntityManager** API により発生する一般的な例外は以下のとおりです。

- IllegalArgumentException: 引数は許可されないか、認識されないか、または不正な形式 (または類似のこと) になります。
- EntityNotFoundException: エンティティが期待されていましたが、要件に一致するものがありません。
- TransactionRequiredException: この操作はトランザクションに含まれる必要があります。
- IllegalStateException: エンティティマネージャが間違った方法で使用されています。

Hibernate 永続レイヤで発生することがあるほとんどのエラーをラップする **HibernateException** は未チェックの例外です。また、Hibernate は **HibernateException** でない他の未チェックの例外をスローすることもできます。これらは復元不可であり、適切なアクションをとる必要があります。

Hibernate は **JDBCException** のデータベースとの対話中にスローされた **SQLException** をラップします。実際には、Hibernate は例外を **JDBCException** の意味のあるサブクラスに変換しようとします。基礎となる **SQLException** は常に **JDBCException.getCause()** から利用できます。Hibernate は **SessionFactory** に接続された **SQLExceptionConverter** を使用して **SQLException** を適切な **JDBCException** サブクラスに変換します。デフォルトでは、**SQLExceptionConverter** は設定されたダイアログにより定義されます。ただし、カスタムの実装を接続することもできます (詳細については、**SQLExceptionConverterFactory** クラスの javadoc を参照)。標準的な **JDBCException** サブタイプは以下のとおりです。

- **JDBCConnectionException** - 基礎となる JDBC 通信でのエラーを示します。
- **SQLGrammarException** - 発行された SQL の文法または構文エラーを示します。
- **ConstraintViolationException** - 何らかの整合性制約違反を示します。
- **LockAcquisitionException** - 要求された操作を実行するのに必要なロックレベルを示します。
- **GenericJDBCException** - 他のどのカテゴリにも属さない一般的な例外。

4.3. EXTENDED 永続コンテキスト

アプリケーションにより管理されたすべてのエンティティマネージャとそのように定義された、コンテナにより管理された永続コンテキストは **EXTENDED** です。つまり、永続コンテキストタイプはトランザクションライフサイクルの範囲を超えます。トランザクションの範囲外で実行された操作に何が起こったのかを理解する必要があります。

EXTENDED 永続コンテキストにおいて、トランザクション外でエンティティマネージャのすべての読み取り専用操作を実行できます (**find()**、**getReference()**、**refresh()**、および読み取りクエリ)。一部の変更操作はトランザクション外で実行できますが、これらの操作は永続コンテキストがトランザクションに参加するまでキューに格納されます。これは、**persist()**、**merge()**、**remove()** の場合に該当します。一部の操作 (**flush()**、**lock()**、および更新/削除クエリ) はトランザクション外で呼び出すことができません。

4.3.1. コンテナにより管理されたエンティティマネージャ

コンテナにより管理されたエンティティマネージャで EXTENDED 永続コンテキストを使用する場合は、永続コンテキストのライフサイクルがステートフルセッション Bean のライフサイクルにバインドされます。また、エンティティマネージャがトランザクション外で作成された場合は、変更操作 (persist、merge、remove) が永続コンテキストでキューに格納され、データベースに対して実行されません。

トランザクションに関連する、またはトランザクションを開始するステートフルセッション Bean のメソッドが後で呼び出されたとき、エンティティマネージャはトランザクションに参加します。永続コンテキストを同期するために、キューに格納されたすべての操作が実行されます。

これは、**entitymanager-per-conversation** パターンを実装する場合に最適です。ステートフルセッション Bean は対話実装を表します。すべての中間対話はトランザクションに関係しないメソッドで処理されます。対話の終了は **JTA** トランザクション内部で処理されます。したがって、キューに格

納されたすべての操作はデータベースに対して実行され、コミットされます。アプリケーション内部の対話の表記法に興味がある場合は、JBoss Seam を調べてください。Jboss Seam は対話とエンティティマネージャのコンセプトを強調し、EJB3 と JSF をともにバインドします。

4.3.2. アプリケーションにより管理されたエンティティマネージャ

アプリケーションにより管理されたエンティティマネージャは常に **EXTENDED** です。トランザクション内部でエンティティマネージャを作成する場合、エンティティマネージャは現在のトランザクションに自動的に参加します。エンティティマネージャがトランザクションの外部で作成された場合は、エンティティマネージャが変更操作をキューに格納します。

- **JTA** エンティティマネージャに対して JTA トランザクションがアクティブな場合は、**`entityManager.joinTransaction()`** が呼び出されます。
- **RESOURCE_LOCAL** エンティティマネージャに対しては **`entityManager.getTransaction().begin()`** が呼び出されます。

エンティティマネージャはトランザクションに参加し、永続コンテキストを同期するためにキューに格納されたすべての操作が実行されます。

JTA トランザクションが関係しない場合、**`entityManager.joinTransaction()`** を呼び出すことは不正ではありません。

4.4. オプティミスティック並行性制御

高い並行性とスケーラビリティに対応する唯一の方法は、バージョン機能によるオプティミスティック並行性制御です。バージョンチェックはバージョン番号またはタイムスタンプを使用して競合する更新を検出します (および更新を失わないようにします)。Hibernate はオプティミスティック並行性を使用するアプリケーションコードを記述するのに 3 つの方法を提供します。示される使用ケースは長いアプリケーショントランザクションのコンテキストですが、バージョンチェックには、単一データベーストランザクションでの更新の損失を防ぐ利点があります。

4.4.1. アプリケーションバージョンチェック

永続メカニズムをあまり使用しない実装の場合、データベースとの各対話は新しい **`EntityManager`** で行われ、開発者はすべての永続インスタンスを操作する前にデータベースからそれらのインスタンスをリロードする必要があります。この方法では、アプリケーションが独自のバージョンチェックを実行してアプリケーショントランザクションの分離を保証します。この方法はデータベースアクセスの点で最も非効率です。これは EJB2 エンティティに最も類似した方法です。

```
// foo is an instance loaded by a previous entity manager
em = factory.createEntityManager();
EntityTransaction t = em.getTransaction();
t.begin();
int oldVersion = foo.getVersion();
Foo dbFoo = em.find( foo.getClass(), foo.getKey() ); // load the current
state
if ( dbFoo.getVersion()!=foo.getVersion() )
    throw new StaleObjectStateException("Message", oldVersion);
dbFoo.setProperty("bar");
t.commit();
em.close();
```

version プロパティは **@Version** を使用してマップされ、エンティティがダーティ状態の場合にエンティティマネージャはフラッシュ中にバージョンをインクリメントします。

当然、低データ並行性環境を使用し、バージョンチェックを必要としない場合は、この方法を使用し、バージョンチェックを省略できます。この場合は、**last commit wins** が長いアプリケーショントランザクションのデフォルト方針になります。アプリケーションのユーザーはエラーメッセージなしで更新を失ったり、競合する変更をマージしたりすることがあるため、混乱する場合があります。

手動によるバージョンチェックは非常に規模が小さい状況でのみ現実的であり、ほとんどのアプリケーションには適用されません。多くの場合、単一インスタンスだけでなく変更されたオブジェクトの完全なグラフもチェックする必要があります。Hibernate はデタッチされたインスタンスまたは拡張されたエンティティマネージャを使用して自動バージョンチェックを提供し、設計パラダイムとして永続コンテキストを提供します。

4.4.2. 拡張されたエンティティマネージャと自動バージョン機能

単一の永続コンテキストはアプリケーショントランザクション全体に使用されます。エンティティマネージャはフラッシュ時にインスタンスバージョンをチェックし、同時の変更が検出された場合に例外をスローします。開発者は、例外を捕捉し、処理する必要があります (一般的なオプションは、ユーザーが変更をマージするか、無効でないデータでビジネスプロセスを再開することです)。

EXTENDED 永続コンテキストで、アクティブなトランザクション外部で実行されたすべての操作がキューに格納されます。**EXTENDED** 永続コンテキストはアクティブなトランザクションで実行されたとき (最悪の場合はコミット時) にフラッシュされます。

Entity Manager は、ユーザーとの対話の待機中に基礎となる JDBC 接続から切断されます。アプリケーションにより管理された拡張エンティティマネージャでは、これはトランザクション完了時に自動的に行われます。コンテナにより管理された拡張エンティティマネージャ (つまり、**@PersistenceContext(EXTENDED)** でアノテートされた SFSB) を保持するステートフルセッション Bean では、これは透過的に行われます。この方法は、データベースアクセスの点で最も効率的です。アプリケーションではバージョンチェックやデタッチされたインスタンスのマージについて心配する必要がなく、各データベーストランザクションでインスタンスをリロードする必要もありません。オープンおよびクローズされた接続の数について心配する場合は、パフォーマンスの影響がないよう接続プロバイダを接続プールにする必要があることに注意してください。以下の例は、非管理環境のイディオムを示しています。

```
// foo is an instance loaded earlier by the extended entity manager
em.getTransaction().begin(); // new connection to data store is obtained
and tx started
foo.setProperty("bar");
em.getTransaction().commit();  // End tx, flush and check version,
disconnect
```

foo オブジェクトはロードされた **persistence context** を認識します。**getTransaction.begin();** を使用してエンティティマネージャは新しい接続を取得し、永続コンテキストを再開します。メソッド **getTransaction().commit()** は、チェックバージョンをフラッシュするだけでなく JDBC 接続からエンティティマネージャを切断し、接続をプールに返します。

ユーザーが考える間に永続コンテキストが大きすぎて保存できず、保存する場所がわからない場合、このパターンは問題となります。たとえば、**HttpSession** はできるだけ小さくする必要があります。永続コンテキストは (必須の) 一次キャッシュであり、ロードされたすべてのオブジェクトを含むため、この方針は少ない要求/応答サイクルに対してのみ使用できます。これは、永続コンテキストが無効なデータを持つため、推奨されます。

要求時に拡張エンティティマネージャを保存する場所はユーザーが決めることができます。EJB3 コン

テナ内部で、上述したようにステートフルセッション Bean を使用します。**HttpSession** に保存するために Web レイヤーには送信しないでください (または異なる層に対してシリアル化しないでください)。非管理 2 層環境では、**HttpSession** が保存に適した場所である場合があります。

4.4.3. デタッチされたオブジェクトと自動バージョン機能

このパラダイムでは、データストアとの各対話が新しい永続コンテキストで行われます。ただし、データベースとの各対話に対して同じ永続インスタンスが再利用されます。アプリケーションは最初に別の永続コンテキストにロードされたデタッチ済みのインスタンスのステータスを操作し、**EntityManager.merge()** を使用して変更をマージします。

```
// foo is an instance loaded by a non-extended entity manager
foo.setProperty("bar");
entityManager = factory.createEntityManager();
entityManager.getTransaction().begin();
managedFoo = entityManager.merge(foo);  // discard foo and from now on use
managedFoo
entityManager.getTransaction().commit();
entityManager.close();
```

再び、エンティティマネージャはフラッシュ時にインスタンスバージョンをチェックして、更新が競合する場合は例外をスローします。

第5章 エンティティリスナーおよびコールバックメソッド

5.1. 定義

多くの場合、アプリケーションが永続メカニズム内部で発生した特定のイベントに反応する点は役に立ちます。これにより、特定の種類の汎用機能を実装したり、組み込み機能を拡張したりできるようになります。EJB3 仕様は、このために 2 つの関連するメカニズムを提供します。

エンティティのメソッドは特定のエンティティライフサイクルイベントの通知を受け取るコールバックメソッドとして指定できます。コールバックメソッドは、コールバックアノテーションによりアノテートされます。また、エンティティクラス内部で直接定義されたコールバックメソッドの代わりに使用するエンティティリスナークラスを定義できます。エンティティリスナーは引数なしのコンストラクタを持つステートレスクラスです。エンティティリスナーは、**@EntityListeners** アノテーションでエンティティクラスをアノテートすることにより定義されます。

```
@Entity
@EntityListeners(class=Audit.value)
public class Cat {
    @Id private Integer id;
    private String name;
    private Date dateOfBirth;
    @Transient private int age;
    private Date lastUpdate;
    //getters and setters

    /**
     * Set my transient property at load time based on a calculation,
     * note that a native Hibernate formula mapping is better for this
purpose.
     */
    @PostLoad
    public void calculateAge() {
        Calendar birth = new GregorianCalendar();
        birth.setTime(dateOfBirth);
        Calendar now = new GregorianCalendar();
        now.setTime( new Date() );
        int adjust = 0;
        if ( now.get(Calendar.DAY_OF_YEAR) -
birth.get(Calendar.DAY_OF_YEAR) < 0) {
            adjust = -1;
        }
        age = now.get(Calendar.YEAR) - birth.get(Calendar.YEAR) + adjust;
    }
}

public class LastUpdateListener {
    /**
     * automatic property set before any database persistence
     */
    @PreUpdate
    @PrePersist
    public void setLastUpdate(Cat o) {
        o.setLastUpdate( new Date() );
    }
}
```

同じコールバックメソッドまたはエンティティリスナーメソッドは複数のコールバックアノテーションでアノテートできます。該当するエンティティに対して、2 つのメソッドを同じコールバックアノテーションによりアノテートすることはできません (コールバックメソッドであるか、エンティティリスナーメソッドであるかは関係ありません)。コールバックメソッドは引数がないメソッド (戻り値タイプがなく任意の名前) です。エンティティリスナーはシグネチャ **`void <METHOD>(Object)`** (Object は実際のエンティティタイプ) です (Hibernate Entity Manager ではこの制限が緩和され、**`java.lang.Object`** タイプの **`Object`** を使用できます (複数のエンティティでリスナーを共有できます)。

コールバックメソッドは **`RuntimeException`** を発生させます。現在のトランザクション (存在する場合) はロールバックする必要があります。以下のコールバックが定義されます。

表5.1 コールバック

タイプ	定義
@PrePersist	エンティティマネージャの永続化操作が実際に実行される、またはカスケードされる前に実行されます。このコールは永続化操作と同期されます。
@PreRemove	エンティティマネージャの削除操作が実際に実行される、またはカスケードされる前に実行されます。このコールは削除操作と同期されます。
@PostPersist	エンティティマネージャの永続化操作が実際に実行される、またはカスケードされる前に実行されます。このコールはデータベースの INSERT の実行後に行われます。
@PostRemove	エンティティマネージャの削除操作が実際に実行される、またはカスケードされる前に実行されます。このコールはデータベースの削除操作と同期されます。
@PreUpdate	データベースの UPDATE 操作が実行される前に実行されます。
@PostUpdate	データベースの UPDATE 操作が実行された後に実行されます。
@PostLoad	エンティティが現在の永続化コンテキストにロードされた後、またはエンティティ更新された後に実行されます。

コールバックメソッドは **`EntityManager`** メソッドまたは **`Query`** メソッドを呼び出さないようにする必要があります。

重要

このメソッドが例外を発生する場合があるため、フラッシュ中に遅延コレクションやプロキシを初期化できません。

Hibernate イベントで遅延コレクションにアクセスする際、遅延ロードされたコレクションに対応する CollectionEntry を PersistenceContext に追加します。フラッシュ時は、コレクションが処理されず、また、無視するように設定もされていないため、Hibernate は追加したコレクション値にアクセスしクラッシュします。

以下の回避策は、別のセッションを作成し、元のセッションがフラッシュメソッドを呼び出す前に遅延コレクションを初期化します。

```
SessionImplementor si = (SessionImplementor)
(event.getSession());
Session anotherSession =
si.getFactory().openSession(si.getJDBCContext().connection());
Object obj = anotherSession.get(
event.getEntity().getClass(),event.getId());
if(obj instanceof Parent){
   Parent parent = (Parent)obj;
   Iterator it = parent.getChildren().iterator();
   while(it.hasNext()){
       Child child = (Child)it.next();
   }
}
anotherSession.close()
```

5.2. コールバックおよびリスナーの継承

エンティティごとに複数のエンティティリスナーを階層の異なるレベルで定義できます。また、複数のコールバックを階層の異なるレベルで定義することもできます。ただし、同じエンティティまたは同じエンティティリスナーの 2 つのリスナーを定義することはできません。

イベントが発生すると、リスナーは次の順序で実行されます。

- **@EntityListeners** (アレイの順序の該当するエンティティまたはスーパークラス)
- スーパークラスのエンティティリスナー (最も高いものが最初)
- エンティティのエンティティリスナー
- スーパークラスのコールバック (最も高いものが最初)
- エンティティのコールバック

@ExcludeSuperclassListeners を使用することにより、エンティティリスナーの継承を停止できます。すべてのスーパークラス **@EntityListeners** が無視されます。

5.3. XML 定義

EJB3 仕様では、EJB3 配備記述子を使用してアノテーションオーバーライドを行えます。役に立つ追加機能であるデフォルトのイベントリスナーも存在します。

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
orm_1_0.xsd"
                 version="1.0"
        >
    <persistence-unit-metadata>
        <persistence-unit-defaults>
            <entity-listeners>
                <entity-listener
class="org.hibernate.ejb.test.pack.defaultpar.IncrementListener">
                    <pre-persist method-name="increment"/>
                </entity-listener>
            </entity-listeners>
        </persistence-unit-defaults>
    </persistence-unit-metadata>
    <package>org.hibernate.ejb.test.pack.defaultpar</package>
    <entity class="ApplicationServer">
        <entity-listeners>
            <entity-listener class="OtherIncrementListener">
                <pre-persist method-name="increment"/>
            </entity-listener>
        </entity-listeners>


        <pre-persist method-name="calculate"/>
    </entity>
</entity-mappings>
```

該当するエンティティのエンティティリスナーをオーバーライドできます。エンティティリスナーは該当するクラスに対応し、1 つまたは複数のイベントにより該当するメソッドコールが呼び出されます。また、コールバックを定義するためにエンティティ自体に対するイベントを定義することもできます。

該当する永続化単位のマップされたすべてのエンティティのエンティティリスナースタック上で最初に適用するデフォルトのエンティティリスナーを定義できます。エンティティがデフォルトリスナーを継承しないようにする場合は、@ExcludeDefaultListeners (または、<exclude-default-listeners/>) を使用できます。

第6章 バッチ処理

バッチ処理は従来、完全オブジェクト/関係マッピングで実現することは困難でした。ORM で重要なのはオブジェクトステータス管理であり、オブジェクトステータスがメモリ内で利用可能であることを暗黙的に示しています。ただし、Hibernate は Hibernate リファレンスガイドで説明された、バッチ処理を最適化する機能をいくつか持ちます。ただし、JEB3 永続化は少し異なります。

6.1. 一括更新/削除

すでに説明したように、自動的かつ透過的なオブジェクト/関係マッピングは、オブジェクトステータスの管理に関係します。これは、オブジェクトステータスがメモリで利用可能であることを暗黙的に示します。したがって、データベースのデータを直接更新または削除した場合 (SQL の **UPDATE** と **DELETE** を使用) に、データベースはメモリ内ステータスに影響を及ぼしません。ただし、Hibernate は、EJB-QL (*7章EJB-QL: オブジェクトクエリ言語*) で実行される一括 SQL スタイルの **UPDATE** および **DELETE** ステートメント実行を提供します。

UPDATE ステートメントと **DELETE** ステートメントの疑似構文は **(UPDATE | DELETE) FROM? ClassName (WHERE WHERE_CONDITIONS)?** です。以下のことに注意してください。

- from 句で、FROM キーワードはオプションです。
- from 句では 1 つのクラスだけ指定できます。エイリアスを指定することはできません (これは現在の Hibernate の制限であり、すぐに取り除かれる予定です)。
- 結合 (暗黙的または明示的) は一括 EJB-QL クエリで指定できません。サブクエリは where 句で使用できます。
- where 句もオプションです。

たとえば、EJB-QL の **UPDATE** を実行するには、**Query.executeUpdate()** メソッドを使用します。

```
EntityManager entityManager = entityManagerFactory.createEntityManager();
entityManager.getTransaction().begin();

String ejbqlUpdate = "update Customer set name = :newName where name =
:oldName";
int updatedEntities = entityManager.createQuery( ejbqlUpdate )
                            .setParameter( "newName", newName )
                            .setParameter( "oldName", oldName )
                            .executeUpdate();
entityManager.getTransaction().commit();
entityManager.close();
```

EJB-QL の **DELETE** を実行するには、同じ **Query.executeUpdate()** メソッド (このメソッドの名前は JDBC の **PreparedStatement.executeUpdate()** に精通しているユーザーに向けて付けられています) を使用します。

```
EntityManager entityManager = entityManagerFactory.createEntityManager();
entityManager.getTransaction().begin();

String hqlDelete = "delete Customer where name = :oldName";
int deletedEntities = entityManager.createQuery( hqlDelete )
                            .setParameter( "oldName", oldName )
                            .executeUpdate();
entityManager.getTransaction().commit();
```

```
entityManager.close();
```

Query.executeUpdate() メソッドにより返された **int** 値は、操作により影響を受けたエンティティの数を示します。これは、データベースで影響を受けた行の数と相関関係にある場合があります (または相関関係にない場合もあります)。EJB-QL の一括操作により、たとえば、結合されたサブクラスに対して複数の実際の SQL ステートメントが実行されることがあります。返された数は、ステートメントにより影響を受けた実際のエンティティの数を示します。結合されたサブクラスの例について、サブクラスのいずれかに対して削除を行うと、サブクラスがマップされたテーブルだけでなく、「ルート」テーブルと継承階層の下層にある結合されたサブクラステーブルに対しても削除が行われます。

第7章 EJB-QL: オブジェクトクエリ言語

EJB3-QL は HQL (ネイティブ Hibernate Query Language) の影響を強く受けています。したがって、両方とも SQL に非常に似ていますが、移植可能でありデータベーススキーマとは独立しています。HQL に精通している方は EJB-QL を問題なく使用できるはずです。実際には、EJB-QL クエリと HQL クエリには同じクエリ API を使用します。EJB アプリケーションを引き続き移植可能にするために、ベンダー固有の拡張機能なしで EJB-QL を使用する必要があります。

7.1. 大文字と小文字を区別

クエリは Java クラスおよびプロパティの名前を除き大文字と小文字を区別します。したがって、**SeLeCT** は **sELEct** と **SELECT** と同じですが、**org.hibernate.eg.FOO** は **org.hibernate.eg.Foo** ではなく、**foo.barSet** は **foo.BARSET** ではありません。

このマニュアルでは、小文字の EJBQL キーワードを使用します。大文字のキーワードがあるクエリの方が読みやすいユーザーもいると思いますが、これは Java コードに組み込む場合に読み難くなります。

7.2. FROM 句

EJB-QL クエリの最も簡単な形式は以下のとおりです。

```
select c from eg.Cat c
```

これは単にクラス **eg.Cat** のすべてのインスタンスを返します。HQL とは異なり、EJB-QL では select 句はオプションではありません。通常、エンティティ名はデフォルトで未修飾クラス名 (**@Entity**) に設定されるため、クラス名を修飾する必要はありません。したがって、通常は以下のように記述します。

```
select c from Cat c
```

気づいたかもしれませんが、クラスにエイリアスを割り当てることができます。**as** キーワードはオプションです。エイリアスを使用すると、クエリの別の箇所で **Cat** を参照できます。

```
select cat from Cat as cat
```

複数のクラスが現れ、直積集合または "クロス" 結合が行われることがあります。

```
select form, param from Formula as form, Parameter as param
```

ローカル変数に対する Java の命名規則と同様にクエリエイリアスの最初の文字を小文字にすることが推奨されます (**domesticCat** など)。

7.3. 関係と結合

また、**join** を使用して関連するエンティティ、または値のコレクションのエレメントにもエイリアスを割り当てることができます。

```
select cat, mate, kitten from Cat as cat
    inner join cat.mate as mate
    left outer join cat.kittens as kitten
```

```
select cat from Cat as cat left join cat.mate.kittens as kittens
```

サポートされた結合タイプは ANSI SQL のものを使用しています。

- **inner join**
- **left outer join**

inner join、**left outer join** コンストラクトは短縮できます。

```
select cat, mate, kitten from Cat as cat
    join cat.mate as mate
    left join cat.kittens as kitten
```

また、"フェッチ (fetch)" 結合では、単一選択を使用して値の関係またはコレクションを親オブジェクトとともに初期化できます。これは、コレクションの場合に特に役に立ち、関係およびコレクションマッピングメタデータのフェッチオプションよりも優先されます。詳細については、Hibernate リファレンスガイドのパフォーマンスに関する章を参照してください。

```
select cat from Cat as cat
    inner join fetch cat.mate
    left join fetch cat.kittens
```

関連するオブジェクトを **where** 句 (または他の任意の句) で使用すべきでないため、フェッチ結合は通常エイリアスを割り当てる必要はありません。また、関連するオブジェクトはクエリ結果で直接返されません。代わりに、これらのオブジェクトは親オブジェクトを介してアクセスできます。エイリアスが必要となる可能性がある唯一のケースは、別のコレクションを再帰的にフェッチ結合する場合です。

```
select cat from Cat as cat
    inner join fetch cat.mate
    left join fetch cat.kittens child
    left join fetch child.kittens
```

fetch コンストラクトは **scroll()** または **iterate()** を使用して呼び出されたクエリで使用できないことに注意してください。また、**fetch** は **setMaxResults()** または **setFirstResult()** とともに使用しないでください。クエリ内の複数のコレクションをフェッチ結合することにより直積集合を作成できます。この集合の結果は予期したものよりも大きくないことに注意してください。複数のコレクションロールをフェッチ結合すると、バッグマッピングに対して予期しない結果がもたらされることがあります。したがって、この場合はクエリをどのように構築するかについて注意してください。

プロパティレベルのレイジーフェッチ (バイトコード計測) を使用する場合は、**fetch all properties** を使用して Hibernate で強制的にレイジープロパティをすぐに (最初のクエリで) フェッチできます。

```
select doc from Document doc fetch all properties order by doc.name
```

```
select doc from Document doc fetch all properties where lower(doc.name)
like '%cats%'
```

7.4. SELECT 句

select 句は、クエリ結果セットで返すオブジェクトとプロパティを取得します。以下のことに留意してください。

```
select mate
from Cat as cat
    inner join cat.mate as mate
```

クエリは 他の**Cat** の **mate** を選択します。実際には、このクエリを以下のようにもっとコンパクトに記述できます。

```
select cat.mate from Cat cat
```

クエリは、以下のようにコンポーネントタイプのプロパティを含む任意の値タイプのプロパティを返すことができます。

```
select cat.name from DomesticCat cat
where cat.name like 'fri%'
```

```
select cust.name.firstName from Customer as cust
```

クエリはタイプ **Object[]** のアレイとして複数のオブジェクトまたはプロパティを返すことができます。

```
select mother, offspr, mate.name
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

または **List** (HQL 固有の機能) として返すことができます。

```
select new list(mother, offspr, mate.name)
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

または実際のタイプセーフ Java オブジェクトとして返すことができます。

```
select new Family(mother, mate, offspr)
from DomesticCat as mother
    join mother.mate as mate
    left join mother.kittens as offspr
```

クラス **Family** が適切なコンストラクトを持っていると見なします。

as を使用して、選択された式にエイリアスを割り当てることができます。

```
select max(bodyWeight) as max, min(bodyWeight) as min, count(*) as n
from Cat cat
```

これは、**select new map** (HQL 固有の機能) とともに使用する場合に最も役に立ちます。

```
select new map( max(bodyWeight) as max, min(bodyWeight) as min, count(*)
as n )
from Cat cat
```

このクエリは、エイリアスからの **Map** を選択された値に返します。

7.5. 集約関数

HQL クエリはプロパティで集約関数の結果を返すこともできます。

```
select avg(cat.weight), sum(cat.weight), max(cat.weight), count(cat)
from Cat cat
```

サポート対象の集約関数は以下のとおりです。

- **avg(...), avg(distinct ...), sum(...), sum(distinct ...), min(...), max(...)**
- **count(*)**
- **count(...), count(distinct ...), count(all...)**

select 句で、算術演算子、連結、および認識された SQL 関数を使用できます (設定されたダイレクト、HQL 固有機能に依存します)。

```
select cat.weight + sum(kitten.weight)
from Cat cat
    join cat.kittens kitten
group by cat.id, cat.weight
```

```
select firstName||' '||initial||' '||upper(lastName) from Person
```

distinct キーワードと **all** キーワードを使用できます。これらのキーワードは SQL と同じセマンティクスを持ちます。

```
select distinct cat.name from Cat cat

select count(distinct cat.name), count(cat) from Cat cat
```

7.6. ポリモーフィックなクエリ

以下のようなクエリは、

```
select cat from Cat as cat
```

Cat だけではなく **DomesticCat** などのサブクラスのインスタンスを返します。Hibernate クエリは **from** 句で**任意の** Java クラスまたはインターフェースを指定できます (移植可能な EJB-QL クエリはマップされたエンティティのみを指定すべきです)。クエリはこのクラスを拡張する、またはインターフェースを実装するすべての永続クラスのインスタンスを返します。以下のクエリはすべての永続オブジェクトを返します。

```
from java.lang.Object o // HQL only
```

インターフェース **Named** はさまざまな永続クラスによって実装できます。

```
from Named n, Named m where n.name = m.name // HQL only
```

これらの最後 2 つのクエリでは、複数の SQL **SELECT** が必要であることに注意してください。つまり、**order by** 句はクエリセット全体を適切に順序付けしません (したがって、**Query.scroll()** を使用してこれらのクエリを呼び出すことはできません)。

7.7. WHERE 句

where 句を使用すると、返されたインスタンスのリストの範囲を狭めることができます。エイリアスが存在しない場合は、名前でプロパティを参照できます。

```
select cat from Cat cat where cat.name='Fritz'
```

上記のクエリは 'Fritz' という名前の **Cat** のインスタンスを返します。

```
select foo
from Foo foo, Bar bar
where foo.startDate = bar.date
```

上記のクエリは **Foo** の **startDate** プロパティに等しい **date** プロパティを持つ **bar** のインスタンスが存在する **Foo** のすべてのインスタンスを返します。複合パス式により、**where** 句は非常に強力になります。以下のことに留意してください。

```
select cat from Cat cat where cat.mate.name is not null
```

このクエリは、テーブル (内部) 結合を持つ SQL クエリに変換されます。以下のように記述すると、

```
select foo from Foo foo
where foo.bar.baz.customer.address.city is not null
```

SQL で 4 つのテーブル結合が必要になります。

= 演算子はプロパティだけでなくインスタンスも比較するために使用できます。

```
select cat, rival from Cat cat, Cat rival where cat.mate = rival.mate
```

```
select cat, mate
from Cat cat, Cat mate
where cat.mate = mate
```

特殊なプロパティ (小文字) **id** はオブジェクトの一意の ID を参照するために使用できます (また、マップされた ID プロパティ名を使用することもできます)。このキーワードは HQL に固有であることに注意してください。

```
select cat from Cat as cat where cat.id = 123

select cat from Cat as cat where cat.mate.id = 69
```

2 つ目のクエリは効率的です。テーブル結合は必要ありません。

複合 ID のプロパティを使用することもできます。**Person** が **country** と **medicareNumber** から構成される複合 ID を持つとします。

```
select person from bank.Person person
where person.id.country = 'AU'
    and person.id.medicareNumber = 123456
```

```
select account from bank.Account account
where account.owner.id.country = 'AU'
    and account.owner.id.medicareNumber = 123456
```

この場合も 2 つ目のクエリはテーブル結合を必要としません。

同様に、特別なプロパティ **class** はポリモーフィック永続化の場合にインスタンスの判別子値にアクセスします。where 句に組み込まれた Java クラス名はその判別子値に変換されます。これは HQL に固有です。

```
select cat from Cat cat where cat.class = DomesticCat
```

また、コンポーネントまたは複合ユーザータイプ (およびコンポーネントのコンポーネントなど) のプロパティを指定することもできます。コンポーネントタイプのプロパティで終了するパス式は使用しないでください (コンポーネントのプロパティとは逆になります)。たとえば、**store.owner** がコンポーネント **address** を持つエンティティである場合は、以下のようになります。

```
store.owner.address.city    // okay
store.owner.address         // error!
```

"any" タイプは特別なプロパティ **id** と **class** を持ち、以下のように結合を記述できます (**AuditLog.item** は **<any>** でマップされたプロパティです)。**Any** は Hibernate に固有です。

```
from AuditLog log, Payment payment
where log.item.class = 'Payment' and log.item.id = payment.id
```

上記のクエリで、**log.item.class** と **payment.class** は完全に異なるデータベースカラムの値を参照することに注意してください。

7.8. 式

where 句で許可される式には、SQL で記述できるほとんどのものが含まれます。

- 算術演算子 **+, -, *, /**
- バイナリ比較演算子 **=, >=, <=, <>, !=, like**
- 論理演算子 **and, or, not**
- かっこ **( )** (グルーピングを示す)
- **in**、**not in**、**between**、**is null**、**is not null**、**is empty**、**is not empty**、**member of**、および **not member of**

- "単純" 条件 **case ... when ... then ... else ... end** と "検索" 条件 **case when ... then ... else ... end (HQL に固有)**
- 文字列連結 **...||...** または **concat(...,...) (use concat() for portable EJB-QL queries)**
- **current_date()**、**current_time()**、**current_timestamp()**
- **second(...)**、**minute(...)**、**hour(...)**、**day(...)**、**month(...)**、**year(...)** (HQL に固有)
- EJB-QL 3.0 で定義された任意の関数または演算子: **substring(), trim(), lower(), upper(), length(), locate(), abs(), sqrt(), bit_length()**
- **coalesce()** および **nullif()**
- **cast(... as ...)** (2 つ目の引数は Hibernate タイプの名前) と **extract(... from ...)** (ANSI **cast()** と **extract()** が基礎となるデータベースでサポートされている場合)
- **sign()**、**trunc()**、**rtrim()**、**sin()** などのデータベースでサポートされた任意の SQL スカラー関数
- JDBC IN パラメータ **?**
- 名前付きパラメータ **:name**、**:start_date**、**:x1**
- SQL リテラル **'foo'**、**69**、**'1970-01-01 10:00:01.0'**
- Java **public static final** 定数 **eg.Color.TABBY**

in と**between** は以下のように使用できます。

```
select cat from DomesticCat cat where cat.name between 'A' and 'B'
```

```
select cat from DomesticCat cat where cat.name in ( 'Foo', 'Bar', 'Baz' )
```

否定形式は以下のように記述できます。

```
select cat from DomesticCat cat where cat.name not between 'A' and 'B'
```

```
select cat from DomesticCat cat where cat.name not in ( 'Foo', 'Bar',
'Baz' )
```

同様に、null 値のテストのために **is null** と **is not null** を使用できます。

Hibernate 設定で HQL クエリの置換を宣言することにより、ブール値を式で簡単に使用できます。

```
hibernate.query.substitutions true 1, false 0
```

これにより、この HQL から変換された SQL でキーワード **true** と **false** がリテラル **1** と **0** に置換されます。

```
select cat from Cat cat where cat.alive = true
```

特別なプロパティ **size** または特別な **size()** 関数 (HQL 固有の機能) でコレクションのサイズをテストできます。

```
select cat from Cat cat where cat.kittens.size > 0
```

```
select cat from Cat cat where size(cat.kittens) > 0
```

インデックス化されたコレクションの場合は、**minindex** 関数と **maxindex** 関数を使用して最小インデックスと最大インデックスを参照できます。同様に、**minelement** 関数と **maxelement** 関数を使用して基本タイプのコレクションの最小エレメントと最大エレメントを参照できます。これらは HQL 固有の機能です。

```
select cal from Calendar cal where maxelement(cal.holidays) > current date
```

```
select order from Order order where maxindex(order.items) > 100
```

```
select order from Order order where minelement(order.items) > 10000
```

SQL 関数 **any, some, all, exists, in** は、コレクションのエレメントまたはインデックスセット (**elements** 関数および **indices** 関数) あるいはサブクエリの結果 (下記参照) が渡された場合にサポートされます。サブクエリは EJB-QL によりサポートされますが、**elements** と **indices** は固有の HQL の機能です。

```
select mother from Cat as mother, Cat as kit
where kit in elements(foo.kittens)
```

```
select p from NameList list, Person p
where p.name = some elements(list.names)
```

```
select cat from Cat cat where exists elements(cat.kittens)
```

```
select cat from Player p where 3 > all elements(p.scores)
```

```
select cat from Show show where 'fizard' in indices(show.acts)
```

コンストラクト **size**、**elements**、**indices**、**minindex**、**maxindex**、**minelement**、**maxelement** は Hibernate の where 句でのみ使用できます。

HQL では、インデックス化されたコレクション (アレイ、リスト、マップ) のエレメントはインデックスにより参照できます (where 句のみ)。

```
select order from Order order where order.items[0].id = 1234
```

```
select person from Person person, Calendar calendar
where calendar.holidays['national day'] = person.birthDay
    and person.nationality.calendar = calendar
```

```
select item from Item item, Order order
where order.items[ order.deliveredItemIndices[0] ] = item and order.id =
11
```

```
select item from Item item, Order order
where order.items[ maxindex(order.items) ] = item and order.id = 11
```

[] の内側の式は算術式にすることもできます。

```
select item from Item item, Order order
where order.items[ size(order.items) - 1 ] = item
```

また、1 対多の関係または値のコレクションのエレメントに対して HQL は組込みの **index()** 関数も提供します。

```
select item, index(item) from Order order
    join order.items item
where index(item) < 5
```

基礎となるデータベースでサポートされたスカラー SQL 関数を使用できます。

```
select cat from DomesticCat cat where upper(cat.name) like 'FRI%'
```

HQL の利点をまだ理解できない場合は、以下の SQL クエリがどれだけ長く、読み難いか確認してください。

```
select cust
from Product prod,
    Store store
    inner join store.customers cust
where prod.name = 'widget'
    and store.location.name in ( 'Melbourne', 'Sydney' )
    and prod = all elements(cust.currentOrder.lineItems)
```

ヒント: 以下のように長く読みにくくなります。

```
SELECT cust.name, cust.address, cust.phone, cust.id, cust.current_order
FROM customers cust,
    stores store,
    locations loc,
    store_customers sc,
    product prod
WHERE prod.name = 'widget'
    AND store.loc_id = loc.id
    AND loc.name IN ( 'Melbourne', 'Sydney' )
    AND sc.store_id = store.id
    AND sc.cust_id = cust.id
    AND prod.id = ALL(
        SELECT item.prod_id
        FROM line_items item, orders o
        WHERE item.order_id = o.id
            AND cust.current_order = o.id
    )
```

7.9. ORDER BY 句

クエリにより返されたリストは返されたクラスまたはコンポーネントのプロパティによって順序付けできます。

```
select cat from DomesticCat cat
order by cat.name asc, cat.weight desc, cat.birthdate
```

オプションの **asc** または **desc** はそれぞれ昇順と降順を示します。

7.10. GROUP BY 句

集約値を返すクエリは、返されたクラスまたはコンポーネントのプロパティによってグループ分けできます。

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
```

```
select foo.id, avg(name), max(name)
from Foo foo join foo.names name
group by foo.id
```

having 句を使用することもできます。

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
having cat.color in (eg.Color.TABBY, eg.Color.BLACK)
```

SQL 関数と集約関数は、基礎となるデータベースでサポートされている場合に **having** 句と **order by** 句で許可されます ((MySQL では許可されません)。

```
select cat
from Cat cat
    join cat.kittens kitten
group by cat
having avg(kitten.weight) > 100
order by count(kitten) asc, sum(kitten.weight) desc
```

group by 句と **order by** 句には算術式を含めることができないことに注意してください。

7.11. サブクエリ

副選択をサポートするデータベースの場合、EJB-QL はクエリ内のサブクエリをサポートします。サブクエリはかっこ (多くの場合、SQL 集約関数コール) で囲む必要があります。相関サブクエリ (外部クエリのエイリアスを参照するサブクエリ) も許可されます。

```
select fatcat from Cat as fatcat
where fatcat.weight > (
    select avg(cat.weight) from DomesticCat cat
)
```

```
select cat from DomesticCat as cat
```

```
where cat.name = some (
    select name.nickName from Name as name
)
```

```
select cat from Cat as cat
where not exists (
    from Cat as mate where mate.mate = cat
)
```

```
select cat from DomesticCat as cat
where cat.name not in (
    select name.nickName from Name as name
)
```

選択リストに複数の式があるサブクエリの場合は、タプルコンストラクタを使用できます。

```
select cat from Cat as cat
where not ( cat.name, cat.color ) in (
    select cat.name, cat.color from DomesticCat cat
)
```

一部のデータベース (Oracle または HSQLDB ではない) では、他のコンテキスト (たとえば、コンポーネントまたは複合ユーザータイプを問い合わせる場合) でタプルコンストラクタを使用できます。

```
select cat from Person where name = ('Gavin', 'A', 'King')
```

これはより詳細な以下のクエリと同じです。

```
select cat from Person where name.first = 'Gavin' and name.initial = 'A'
and name.last = 'King')
```

このようなことを行いたくない 2 つの理由が考えられます。1 つはデータベースプラットフォーム間で完全に移植可能でないこと、もう 1 つはクエリがマッピングドキュメントのプロパティの順序に依存することです。

7.12. EJB-QL の例

Hibernate クエリは、非常に強力かつ複雑にすることができます。実際には、強力なクエリ言語は Hibernate の主な利点の 1 つです (現在は EJB-QL)。この項では、Hibernate のクエリの例を示します。

以下のクエリは注文 ID、アイテム数、特定の顧客に対するすべての未払い注文の注文合計値と該当する最小合計値を返します。価格を決定するには、現在のカタログが使用されます。結果となる SQL クエリ (**ORDER**、**ORDER_LINE**、**PRODUCT**、**CATALOG**、および **PRICE** テーブルに対する) は 4 つの内部結合と 1 つの (相関) 副選択を持ちます。

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
```

```
    and price.product = product
    and catalog.effectiveDate < current_date()
    and catalog.effectiveDate >= all (
        select cat.effectiveDate
        from Catalog as cat
        where cat.effectiveDate < current_date()
    )
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount) desc
```

サブクエリの使用を避けたい場合は、以下のように記述します。

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog = :currentCatalog
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount) desc
```

次のクエリは、現在のユーザーによって最新のステータス変更が行われた **AWAITING_APPROVAL** ステータスのすべての支払いを除く各ステータスの支払い数をカウントします。このクエリは **PAYMENT** テーブル、**PAYMENT_STATUS** テーブル、および **PAYMENT_STATUS_CHANGE** テーブルに対して 2 つの内部結合と 1 つの相関副選択を持つ SQL クエリに変換されます。

```
select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
    join payment.statusChanges as statusChange
where payment.status.name <> PaymentStatus.AWAITING_APPROVAL
    or (
        statusChange.timeStamp = (
            select max(change.timeStamp)
            from PaymentStatusChange change
            where change.payment = payment
        )
        and statusChange.user <> :currentUser
    )
group by status.name, status.sortOrder
order by status.sortOrder
```

statusChanges コレクションがセットの代わりにリストとしてマップされた場合、クエリは非常に単純になります。

```
select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
where payment.status.name <> PaymentStatus.AWAITING_APPROVAL
```

```
    or payment.statusChanges[ maxIndex(payment.statusChanges) ].user <>
:currentUser
group by status.name, status.sortOrder
order by status.sortOrder
```

ただし、クエリは HQL 固有です。

次のクエリは、MS SQL Server **isNull()** 関数を使用して現在のユーザーが属する組織のすべてのアカウントと未払いの支払いを返します。このクエリは、**ACCOUNT**、**PAYMENT**、**PAYMENT_STATUS**、**ACCOUNT_TYPE**、**ORGANIZATION**、および **ORG_USER** テーブルに対して 3 つの外部結合、1 つの外部結合、および 1 つの副選択を持つ SQL クエリに変換されます。

```
select account, payment
from Account as account
    join account.holder.users as user
    left outer join account.payments as payment
where :currentUser = user
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name,
PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber, payment.dueDate
```

7.13. 一括更新および削除に関するステートメント

Hibernate は HQL/EJB-QL の UPDATE ステートメントと DELETE ステートメントをサポートするようになりました。詳細については、「一括更新/削除」 を参照してください。

7.14. ヒントと裏技

コレクションのサイズにより結果の順序を決めるには、以下のクエリを使用します。

```
select usr.id, usr.name
from User as usr
    left join usr.messages as msg
group by usr.id, usr.name
order by count(msg)
```

データベースが副選択をサポートする場合は、クエリの where 句で選択サイズの条件を設定できます。

```
from User usr where size(usr.messages) >= 1
```

データベースが副選択をサポートしない場合は、以下のクエリを使用します。

```
select usr.id, usr.name
from User usr.name
    join usr.messages msg
group by usr.id, usr.name
having count(msg) >= 1
```

このソリューションでは、内部結合のためゼロメッセージで **User** を返すことができないため、以下の形式が役に立ちます。

```
select usr.id, usr.name
```

```
from User as usr
    left join usr.messages as msg
group by usr.id, usr.name
having count(msg) = 0
```

第8章 ネイティブクエリ

データベースのネイティブ SQL ダイアレクトでクエリを記述することもできます。これは、クエリヒントや Oracle の CONNECT BY オプションなどのデータベース固有の機能を使用する場合に役に立ちます。また、直接的な SQL/JDBC ベースのアプリケーションから Hibernate へのクリーンな移行パスも提供します。Hibernate では、すべての作成操作、更新操作、およびロード操作に対して記述した SQL (ストアドプロシージャを含む) を指定できます (詳細については、リファレンスガイドを参照)。

8.1. 結果セットの記述

SQL クエリを使用するには、SQL 結果セットを定義する必要があります。この定義により、**`EntityManager`** を使用してエンティティプロパティに対してカラムをマップできるようになります。これは、**`@SqlResultSetMapping`** アノテーションを使用して実行されます。各 **`@SqlResultSetMapping`** は、**`EntityManager`** に対する SQL クエリの作成時に使用される名前を持ちます。

```
@SqlResultSetMapping(name="GetNightAndArea",
    entities={

@EntityResult(entityClass=org.hibernate.test.annotations.query.Night.class
, fields
        = {
          @FieldResult(name="id", column="nid"),
          @FieldResult(name="duration", column="night_duration"),
          @FieldResult(name="date", column="night_date"),
          @FieldResult(name="area", column="area_id")
        }),

@EntityResult(entityClass=org.hibernate.test.annotations.query.Area.class,
fields
        = {
          @FieldResult(name="id", column="aid"),
          @FieldResult(name="name", column="name")
        })
    }
)

@SqlResultSetMapping(name="defaultSpaceShip",
entities=@EntityResult(entityClass=org.hibernate.test.annotations.query.Sp
aceShip.class))
```

また、スカラー結果を定義して、エンティティ結果とスカラー結果を混在することもできます。

```
@SqlResultSetMapping(name="ScalarAndEntities",
  entities={

@EntityResult(entityClass=org.hibernate.test.annotations.query.Night.class
,
      fields = {
          @FieldResult(name="id", column="nid"),
          @FieldResult(name="duration", column="night_duration"),
          @FieldResult(name="date", column="night_date"),
          @FieldResult(name="area", column="area_id")
      }),
```

```
@EntityResult(entityClass=org.hibernate.test.annotations.query.Area.class,
      fields = {
          @FieldResult(name="id", column="aid"),
          @FieldResult(name="name", column="name")
      })
  },
  columns={
      @ColumnResult(name="durationInSec")
  }
)
```

SQL クエリは、カラムエイリアス **durationInSec** を返す必要があります。

@SqlResultSetMapping. に関する詳細については、Hibernate Annotations リファレンスガイドを参照してください。

8.2. ネイティブ SQL クエリの使用

結果セットが定義されたら、ネイティブ SQL クエリを実行できます。**EntityManager** は必要なすべての API を提供します。最初のメソッドは SQL 結果セットの名前を使用してバインドを実行し、2 つ目のメソッドはエンティティデフォルトマッピングを使用します (返されたカラムはマッピングで使用されたのと同じ名前を持つ必要があります)。3 つ目のメソッド (Hibernate エンティティマネージャでは未サポート) は純粋なスカラー結果を返します。

```
String sqlQuery = "select night.id nid, night.night_duration,
night.night_date, area.id aid, "
    + "night.area_id, area.name from Night night, Area area where
night.area_id = area.id "
    + "and night.night_duration >= ?";
Query q = entityManager.createNativeQuery(sqlQuery, "GetNightAndArea");
q.setParameter( 1, expectedDuration );
q.getResultList();
```

このネイティブクエリは、**GetNightAndArea** 結果セットに基づいて night と area を返します。

```
String sqlQuery = "select * from tbl_spaceship where owner = ?";
Query q = entityManager.createNativeQuery(sqlQuery, SpaceShip.class);
q.setParameter( 1, "Han" );
q.getResultList();
```

2 つ目のバージョンは、SQL クエリがメタデータでマップされたのと同じカラムを再利用する 1 つのエンティティを返すときに役に立ちます。

8.3. 名前付きクエリ

ネイティブの名前付きクエリは、EJB-QL 名前付きクエリと同じ呼出側 API を共有します。コードは 2 つの違いを区別する必要はありません。これは、SQL から EJB-QL への移行に非常に役に立ちます。

```
Query q = entityManager.createNamedQuery("getSeasonByNativeQuery");
q.setParameter( 1, name );
Season season = (Season) q.getSingleResult();
```

付録A 改訂履歴

改訂 5.1.2-2.400 **2013-10-30** **Rüdiger Landmann**
Rebuild with publican 4.0.0

改訂 5.1.2-2 **2012-07-18** **Anthony Towns**
Rebuild for Publican 3.0

改訂 5.1.2-100 **Thu Dec 8 2011** **Jared Morgan**
JBoss Enterprise Application Platform 5.1.2 GAに対する変更を追加。本ガイド文書の変更に関する情報は、『 リリースノート 5.1.2』を参照してください。

改訂 5.1.1-100 **Mon Jul 18 2011** **Jared Morgan**
JBoss Enterprise Application Platform 5.1.1 GAに対する変更を追加。本ガイド文書の変更に関する情報は、『 リリースノート 5.1.1』を参照してください。

改訂 5.1.0-106 **Wed Sep 15 2010** **Laura Bailey**
新しいバージョン要件に合わせてバージョン番号を変更
JBoss Enterprise Application Platform 5.1.0 GA 向けに改訂