Hibernate###############

Version: 2.1.6

目次

前書き

WARNING! This is a translated version of the English Hibernate reference documentation. The translated version might not be up to date! However, the differences should only be very minor. Consult the English reference documentation if you are missing information or encounter a translation error. If you like to contribute to a particular translation, contact us on the Hibernate developer mailing list.

Translator(s): Chikara Honma <chikara-honma@exa-corp.co.jp>, Yusuke Hiroto <yusuke-hiroto@exa-corp.co.jp>

今日のエンタープライズ環境では、オブジェクト指向/リレーショナル・データベースの仕事は、厄介で時間のかかるものであることがあります。Hibernateは、 Java環境のオブジェクト/リレーショナル・マッピングツールです。 オブジェクト/リレーショナル・マッピング (ORM)という言葉は、 オブジェクト・モデルで表現されたデータをリレーショナル、 つまりSQLベースの構造のデータに対応付ける技術のことを言います。

HibernateはJavaクラスからデータベースのテーブルへのマッピングを行うだけではなく、 データのクエリと復元機能を提供します。そのおかげで、 SQLとJDBCによる手作業でデータを操作する方法に比べて、開発時間を格段に減らすことができます。

Hibernateの目標は、プログラミング作業に関わる、共通のデータ永続化作業の95%から、 開発者を解放することです。 Hibernateは、データベース上でビジネス・ロジックを実装するストアド・プロシージャを使うだけの、 データ中心アプリケーションに対するベスト・ソリューションであるだけではありません。 オブジェクト指向ドメインモデルと、Javaベースの中間層におけるビジネス・ロジックに対して最も有用です。 しかし、Hibernateはベンダ固有のSQLコードを取り除いたり、カプセル化できます。 また表形式の表現からオブジェクトのグラフへと、リザルト・セットを翻訳する共通作業の手助けもします。

もしHibernateとオブジェクト/リレーショナル・マッピング、さらにはJavaが初めてなら、 以下のステップで進んでください:

1. Tomcatを使った30分間のチュートリアル 章 1. Tomcatでクイック・スタート を読んでください。

2. Hibernateを使う環境を理解するために 章 2. アーキテクチャ を読んでください。

3. Hibernateディストリビューションの `eg/` ディレクトリを見てください。 そこに簡単なスタンド・アローンのアプリケーションがあります。 JDBCドライバを `lib/` ディレクトリにコピーし、 お使いのデータベースに対する適切な値を指定するように `etc/hibernate.properties` を編集してください。 ディストリビューション・ディレクトリでコマンドプロンプトから `ant eg` とタイプしてください (Antを使用) 。 Windows環境ならば `build eg` とタイプしてください。

4. このリファレンス・ドキュメントを第1の情報源として使ってください。 もしアプリケーション設計のさらなる情報が必要だったり、 ステップ・バイ・ステップのチュートリアルの方がお好みなら、 Hibernate in Action (http://www.manning.com/bauer) を読むことを考えてください。 またhttp://caveatemptor.hibernate.orgに来てHibernate in Actionの 例題アプリケーションをダウンロードしてみてください。

5. HibernateのウェブサイトにFAQがあります。

6. サードパーティのデモ、例、チュートリアルがHibernateのウェブサイトからリンクされています。

7. Hibernateウェブサイトのコミュニティ・エリアは、 デザインパターンやいろいろな統合ソリューション （Tomcat, JBoss, Spring, Struts, EJBなど）のよい情報源です。

質問があれば、Hibernateウェブサイトからリンクされている、ユーザ・フォーラムを使ってください。 またバグレポートと機能要求のために、JIRA議題トラッキング・システムを用意しています。もしHibernateの開発に興味があれば、開発者メーリングリストに参加してください。 もしこのドキュメントのあなたの言語への翻訳に興味があれば、 開発者メーリングリストで私たちにコンタクトしてください。

Hibernateの商業的な開発のサポート、製品サポート、トレーニングはJBoss Inc.を通して利用できます。 (http://www.hibernate.org/SupportTraining/ を見てください。） Hibernate はJBoss Professional Open Sourceプロダクト・スイートのプロジェクトです。

1. 日本語訳について

この日本語版Hibernateリファレンス・ドキュメント（以下 日本語版）は、Hibernateプロジェクトの翻訳プロセスに基づいています。日本語版ならびに原文はLGPLライセンスに準じます。日本語版の翻訳者は、日本語版に対して署名以外のその他の一切の権利を放棄いたします。

日本語版の利用によって第三者がこうむるあらゆる不利益に対して、原著者、翻訳者ならびにその組織は一切の保証をいたしかねます。日本語版は誤りを含む可能性があることを認識した上で、ご利用ください。内容の正確な意味を把握するためには、原文をお読みになることをお勧めします。またもし日本語版に誤りを見つけられた場合は、翻訳者にご連絡いただければ幸いです。ただし内容に関してのお問い合わせには、応じかねますのでご了承ください。

1.1. 日本語版翻訳者について

日本語版バージョン2.1.7の6-8,11-17章は本間力(chikara_honma_exa_corp@yahoo.co.jp)が、前書き,1-5,9,10,18,19章は広戸裕介(yusuke_hiroto_exa_corp@yahoo.co.jp)が翻訳しました。なお、翻訳に際し助言と協力をいただいた、(株)エクサの井関知文さんと平間健一さんに感謝いたします。

第1章 Tomcatでクイック・スタート

1.1. Hibernateを始めよう

このチュートリアルでは、Webベースのアプリケーションのための、 Apache Tomcatサーブレット・コンテナを使った、 Hibernate2.1のセットアップについて述べます。 主要なすべてのJ2EEアプリケーション・サーバや、 スタンドアローン・アプリケーションで管理された環境で、Hibernateはよく動作します。 この例で使うデータベース・システムはPostgreSQL7.3ですが、 HibernateのSQL方言を設定しなおすだけで、他のデータベースに変更できます。

最初のステップは、Tomcatインストール・ディレクトリに、 必要なライブラリのすべてをコピーすることです。 このチュートリアルでは、独立したウェブ・コンテキスト （`webapps/quickstart`）を使います。 そのためグローバル・ライブラリ・サーチパス（`TOMCAT/common/lib`）と `webapps/quickstart/WEB-INF/lib` における コンテキスト・レベルのクラスローダ（JARファイルのため）と `webapps/quickstart/WEB-INF/classes` を考慮しなければいけません。 追々グローバル・クラスパスとコンテキスト・クラスパスの、両方のクラスローダに言及します。

まず、ライブラリを2つのクラスパスにコピーしましょう：

1. 初めに、グローバル・クラスパスに、データベースのJDBCドライバをコピーします。 これはTomcatにバンドルされる、DBCPコネクションプール・ソフトウェアに必要です。 Hibernateはデータベースの SQLを実行するために、JDBCコネクションを使います。 そのため、プールされたJDBCコネクションを用意するか、 Hibernateが直接サポートしているプール（C3P0, Proxool）の1つを使うように、 設定しなければいけません。 このチュートリアルでは、グローバル・クラスローダパスに、 `pg73jdbc3.jar` ライブラリ （PostgreSQL7.3とJDK1.4用）をコピーしてください。 もし他のデータベースを使いたければ、適切なJDBCドライバをコピーするだけです。

2. Tomcatのグローバル・クラスローダパスには、他には決して何もコピーしないでください。 そうでなければ、例えばLog4j, commons-loggingなどのいろいろなツールに、 問題が出てきてしまいます。 必ず各アプリケーションごとに、コンテキストパスを使うようにしてください。 つまり `WEB-INF/lib` にライブラリをコピーして、 `WEB-INF/classes` に、 クラスと設定/プロパティ・ファイルを コピーしてください。 両ディレクトリともに、デフォルトでコンテキスト・レベルのクラスパスです。

3. HibernateはJARライブラリのパッケージになっています。 `hibernate2.jar` ファイルは、アプリケーションの他のクラスと一緒に、 コンテキスト・クラスパスに配置すべきです。 Hibernateはいくつかのサードパーティのライブラリを、実行時に必要とします。 これらはHibernateディストリビューションの `lib/` ディレクトリにまとめられています。 表 1.1. 「 Hibernateサードパーティ・ライブラリ 」 を見てください。 コンテキスト・クラスパスに、必要なサードパーティのライブラリをコピーしてください。

表 1.1. Hibernateサードパーティ・ライブラリ

ライブラリ	説明
dom4j（必須）	Hibernateは、XML設定ファイルとXMLマッピング・メタデータ・ファイルの構文解析に、 dom4jを使います。

ライブラリ	説明
CGLIB（必須）	Hibernateは実行時にクラスをエンハンスするために、コード生成ライブラリを使います （Javaのリフレクションと組み合わせて行います）。
Commons Collections, Commons Logging（必須）	HibernateはApache Jakarta Commonsプロジェクトの、 いろいろなユーティリティ・ライブラリを使います。
ODMG4（必須）	Hibernateは、 オプションのODMG準拠の永続マネージャ・インターフェイスを用意しています。 ODMG APIを使うつもりがなくても、コレクションをマッピングするなら これは必要となります。 このチュートリアルではコレクションのマッピングは行いませんが、 いずれにせよこのJARをコピーしておくのは良い考えです。
EHCache（必須）	Hibernateは第2レベルキャッシュのために、 いろいろなキャッシュ・プロバイダを使うことができます。 設定を変更しなければ、EHCacheがデフォルトのキャッシュ・プロバイダです。
Log4j（オプション）	HibernateはCommons Logging APIを使いますが、代わりのロギング機構として、 Log4jを使うこともできます。 コンテキスト・ライブラリ・ディレクトリにLog4jライブラリが配置されると、Commons LoggingはLog4jとコンテキスト・クラスパス中の `log4j.properties` を使うようになります。 Log4jのプロパティ・ファイルのサンプルが、 Hibernateディストリビューションにあります。 そのため、もし背後で何が行われているのか知りたければ、 コンテキスト・クラスパスにlog4j.jarと設定ファイル（`src/` から）をコピーしてください。
その他	Hibernateディストリビューションの `lib/README.txt` ファイルを見てください。 これはHibernateと一緒に配布されている、サードパーティのライブラリの最新のリストです。 必須のライブラリとオプションのライブラリのすべてを、 ここで見つけることができるでしょう。

以上でデータベース・コネクション・プーリングのセットアップが終了し、 TomcatとHibernateの両方から共有できるようになりました。 これは（組み込みのDBCPプーリング機能を使い） Tomcatがプールされ たJDBCコネクションを提供するということです。 Hibernateは、JNDIを通してこのコネクションを要求します。 TomcatはコネクションプールをJNDIにバインドし、 私たちはTomcatのメインの設定ファイル `TOMCAT/conf/server.xml` に、 リソース定義を追加します：

```
<Context path="/quickstart" docBase="quickstart">
    <Resource name="jdbc/quickstart" scope="Shareable" type="javax.sql.DataSource"/>
    <ResourceParams name="jdbc/quickstart">
        <parameter>
            <name>factory</name>
            <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
        </parameter>

        <!-- DBCP################ -->
        <parameter>
            <name>url</name>
            <value>jdbc:postgresql://localhost/quickstart</value>
        </parameter>
```

```
        <parameter>
            <name>driverClassName</name><value>org.postgresql.Driver</value>
        </parameter>
        <parameter>
            <name>username</name>
            <value>quickstart</value>
        </parameter>
        <parameter>
            <name>password</name>
            <value>secret</value>
        </parameter>

        <!-- DBCP################## -->
        <parameter>
            <name>maxWait</name>
            <value>3000</value>
        </parameter>
        <parameter>
            <name>maxIdle</name>
            <value>100</value>
        </parameter>
        <parameter>
            <name>maxActive</name>
            <value>10</value>
        </parameter>
    </ResourceParams>
</Context>
```

この例では、コンテキスト名を `quickstart` にしました。 そのベースは `TOMCAT/webapp/quickstart` ディレクトリです。 サーブレットにアクセスするには、 ブラウザで `http://localhost:8080/quickstart` にアクセスしてください。 （もちろん `web.xml` に、サーブレットの名前をマッピングしておいてください。） また以上で、空の `process()` を持つ単純なサーブレットを作成できます。

Tomcatはこの設定に基づいて、DBCPコネクション・プールを使います。 そして `java:comp/env/jdbc/quickstart` においてJNDIを通して、 プールされたJDBC `Connection` を提供します。 もしコネクション・プールを使っているときにトラブルが発生するようなら、 Tomcatのドキュメントを見てください。 もしJDBCドライバの例外メッセージを受け取るようなら、 まずHibernateなしでJDBCコネクション・プールを起動してみてください。 TomcatとJDBCのチュートリアルはウェブから利用できます。

次のステップは、JNDIバインドのプールからコネクションを使い、Hibernateを設定することです。 この例題では、XMLベースでHibernateを設定します。 プロパティを使う基本的な方法では、機能は同じものの、利点がいくつか損なわれてしまいます。 普通は、より便利なXMLベースで設定を行います。 XML設定ファイルは、 コンテキスト・クラスパス（`WEB-INF/classes`）に配置してください。 `hibernate.cfg.xml` は以下のようになります：

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration
    PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">

<hibernate-configuration>

    <session-factory>

        <property name="connection.datasource">java:comp/env/jdbc/quickstart</property>
        <property name="show_sql">false</property>
        <property name="dialect">net.sf.hibernate.dialect.PostgreSQLDialect</property>

        <!-- ########## -->
        <mapping resource="Cat.hbm.xml"/>
```

```
    </session-factory>

</hibernate-configuration>
```

ここではSQLコマンドのロギングをオフにしています。 また、どのデータベースのSQL方言が使われていて、 どこからJDBCコネクションを手に入れれば良いかを （データソース・プールがバインドされているJNDIのアドレスを定義することで） Hibernateに伝えます。 データベースは、SQL「標準」の解釈がそれぞれ異なっているので、 方言を設定する必要があります。 Hibernateは、主要なすべての商用とオープンソースのデータベースに対して違いをケアし、 それぞれの方言に対応します。

`SessionFactory` は、データストア1つに対応するHibernateの概念です。 XML設定ファイルを複数用意し、アプリケーションで `Configuration` と `SessionFactory` オブジェクトを複数作成することで、 データベースを複数使うことができます。

`hibernate.cfg.xml` の最後の要素で、 永続クラス `Cat` のためのHibernate XMLマッピング・ファイルの名前として、 `Cat.hbm.xml` を定義しています。 このファイルは、POJOクラスのデータベース・テーブル（や複数のテーブル）へのマッピングに対する メタデータを含んでいます。 またすぐにこのファイルに戻ってくることになります。 それではPOJOクラスを書いて、メタデータのマッピングを定義しましょう。

1.2. 最初の永続クラス

永続クラスの、プレーンの古いJavaオブジェクト (POJO)プログラミング・モデルで、 Hibernateは最も良く動作します （POJOは、プレーンの普通のJavaオブジェクトと呼ばれることもあります）。 クラスのプロパティがgetter, setterメソッドを通してアクセス可能である点で、 POJOはJavaBeanに大変良く似ており、 パブリックで可視のインターフェイスから内部表現を保護します：

```
package net.sf.hibernate.examples.quickstart;

public class Cat {

    private String id;
    private String name;
    private char sex;
    private float weight;

    public Cat() {
    }

    public String getId() {
        return id;
    }

    private void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public char getSex() {
        return sex;
```

```
    }

    public void setSex(char sex) {
        this.sex = sex;
    }

    public float getWeight() {
        return weight;
    }

    public void setWeight(float weight) {
        this.weight = weight;
    }

}
```

Hibernateは、プロパティの型に何を使うかについて制限しません。 Javaコレクション・フレームワークのクラスを含む、すべてのJava JDK型とプリミティブ （`String`, `char`, `Date` など）を マッピングできます。 それらをバリューや、バリューのコレクションや、他のエンティティへの関連として、 マッピングできます。 `id` は、クラスのデータベース識別子（主キー）を表す特別なプロパティで、 `Cat` のようなエンティティには非常におすすめです。 識別子は、Hibernateが内部的に使うだけでも構いませんが、 それではアプリケーション・アーキテクチャの持つ柔軟性をいくらか失うことになります。

永続クラスは、特別なインターフェイスを実装する必要も、 特別なルートの永続クラスからサブクラス化する必要もありません。 また、Hibernateはバイトコード操作のような、ビルド時のプロセスを使いません。 唯一Javaのリフレクションと、 （CGLIBを通した）実行時のクラス・エンハンスメントだけに依存しています。 そのためHibernateに全く依存させず、POJOクラスをデータベースのテーブルにマッピングできます。

1.3. ネコのマッピング

`Cat.hbm.xml` マッピング・ファイルは、 オブジェクト/リレーショナル・マッピングに必要なメタデータを含んでいます。 メタデータは永続クラスの定義と、（カラムや他のエンティティへの外部キー関係のための） プロパティのデータベース・テーブルへのマッピングを含んでいます。

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>

    <class name="net.sf.hibernate.examples.quickstart.Cat" table="CAT">

        <!-- 32 hex##########
                ###UUID########Hibernate######### -->
        <id name="id" type="string" unsaved-value="null" >
            <column name="CAT_ID" sql-type="char(32)" not-null="true"/>
            <generator class="uuid.hex"/>
        </id>

        <!-- cat############################## -->
        <property name="name">
            <column name="NAME" length="16" not-null="true"/>
        </property>

        <property name="sex"/>

        <property name="weight"/>
```

```
    </class>

</hibernate-mapping>
```

すべての永続クラスが、識別子属性を持つべきです （実際には、エンティティのコンポーネントとしてマッピングされるのは、 依存するバリュー・オブジェクトではなく、エンティティを表すクラスだけです。）。 このプロパティは、永続オブジェクトを識別するために使われます: `catA.getId().equals(catB.getId())` がtrueなら、 2つのcatを同じものとみなすこの概念は、 データベース・アイデンティティ と呼ばれています。 Hibernateには、多様なシナリオに対応する、いろいろな識別子ジェネレータがバンドルされています （データベース・シーケンスを使うnativeジェネレータや、hi/lo識別子テーブルや、 アプリケーション代入識別子などがあります）。 この例では、UUIDジェネレータ（データベースが生成する整数の代理キーの方が好ましいので、 これはテスト用としてだけおすすめします）を使い、 また（テーブルの主キーとして）Hibernateが生成する識別子の値のための、 `CAT` テーブルの `CAT_ID` カラムを指定します。

`Cat` の他のプロパティはすべて、同じテーブルにマッピングされます。 `name` プロパティについては、 明示的にデータベース・カラムを定義して、マッピングしています。 HibernateのSchemaExport ツールを使い、 マッピング定義から（SQLのDDL文として）データベース・スキーマを自動生成する場合、 これは特に有用です。 他のプロパティはすべて、Hibernateのデフォルトの設定を使い、マッピングされます。 これは普通なら最も時間が必要になるところです。 データベースの `CAT` テーブルは、以下のようになっています:

```
 ### |         #          | ###
--------+----------------------+-----------
 cat_id | character(32)         | not null
 name   | character varying(16) | not null
 sex    | character(1)          |
 weight | real                  |
Indexes: cat_pkey primary key btree (cat_id)
```

それでは、このテーブルを手作業で作成しましょう。 このステップをSchemaExportツールで自動化したければ、 後で 章 15. ツールセット･ガイド を読んでください。 このツールは、テーブル定義、カスタムカラム型制約、ユニーク制約、インデックスを含む、 完全なSQL DDLを作成することができます。

1.4. ネコと遊ぶ

ついに、Hibernateの `Session` を始める準備ができました。 私たちは、`Cat` のデータベースへの保存と復元のために、 永続マネージャ インターフェイスを使います。 しかしまずは、`SessionFactory` から `Session` (Hibernateの作業単位) を取得しなければいけません:

```
SessionFactory sessionFactory =
            new Configuration().configure().buildSessionFactory();
```

`SessionFactory` は、1つのデータベースだけに責任を持ち、 1つのXML設定ファイル（`hibernate.cfg.xml`）だけを使います。 `SessionFactory`（これは更新不能です）をビルドする 前に `Configuration` にアクセスすることで、 他のプロパティを設定することができます （マッピング・メタデータを変更することさえ可能です）。 それでは、アプリケーション中のどこで `SessionFactory` を作成し、 どのようにしてそれにアクセスすればよいのでしょうか？

普通、`SessionFactory` は1度だけビルドされます。 例えば load-on-startup サーブレットのスター

ト・アップで行います。 つまりこれは、サーブレット中のインスタンス変数ではなく、 どこか他の場所に保持すべきだということでもあります。 さらには、アプリケーション・コードの中で簡単に SessionFactory にアクセスできるよう、 ある種の シングルトン が必要だということです。 設定の問題と、SessionFactory への容易なアクセスの問題の両方を、 次にお見せする方法で解決します。

HibernateUtil ヘルパ・クラスを実装しました：

```
import net.sf.hibernate.*;
import net.sf.hibernate.cfg.*;

public class HibernateUtil {

    private static Log log = LogFactory.getLog(HibernateUtil.class);

    private static final SessionFactory sessionFactory;

    static {
        try {
            // SessionFactory######
            sessionFactory = new Configuration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            log.error("Initial SessionFactory creation failed.", ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static final ThreadLocal session = new ThreadLocal();

    public static Session currentSession() throws HibernateException {
        Session s = (Session) session.get();
        // ####Thread############Session########
        if (s == null) {
            s = sessionFactory.openSession();
            session.set(s);
        }
        return s;
    }

    public static void closeSession() throws HibernateException {
        Session s = (Session) session.get();
        session.set(null);
        if (s != null)
            s.close();
    }
}
```

このクラスは、static属性として SessionFactory を管理するだけではなく、 現在実行中のスレッドに対する Session を保持する ThreadLocal を持っています。 このヘルパを使う前に、スレッド・ローカル変数というJavaの概念を確実に理解してください。

SessionFactory はスレッドセーフであり、多くのスレッドが同時並行的にアクセスし、 Session をリクエストすることができます。 Session は、データベースに対する1つの作業の単位を表す、 スレッドセーフではないオブジェクトです。 Session は SessionFactory によってオープンされ、 すべての作業が完了したときにクローズされます：

```
Session session = HibernateUtil.currentSession();

Transaction tx= session.beginTransaction();

Cat princess = new Cat();
princess.setName("Princess");
princess.setSex('F');
princess.setWeight(7.4f);
```

```
session.save(princess);
tx.commit();

HibernateUtil.closeSession();
```

ある1つの Session において、 すべてのデータベース操作が1つのトランザクション内で起こります （read-only操作でさえです）。 ベースとなるトランザクション戦略（私たちの場合は、JDBCトランザクション）から抽象化するために、 Hibernate Transaction APIを使っています。 このおかげで何も変更せずに、 コードを（JTAを使う）コンテナ管理トランザクションと一緒に配置できます。 上にあげた例で、全く例外処理をしていないことに注目してください。

また、好きなだけ HibernateUtil.currentSession(); をコールすることができます。 すると、常にこのスレッドの現時点の Session を取得します。 作業単位が完了した後、HTTPレスポンスが送られるまでに、サーブレットのコードかサーブレット・フィルタの中で、 確実に Session をクローズしなければなりません。 後者のすばらしい副作用は、簡単なlazy初期化です： ビューがレンダリングされるとき、Session はまだオープンしているので、 グラフのナビゲートの間、Hibernateは初期化されていないオブジェクトをロードすることができます。

Hibernateは、データベースからオブジェクトを復元するための方法をいろいろ用意しています。最も柔軟な方法は、Hibernateクエリ言語（HQL）を使う方法です。 これは簡単に習得することのできる、SQLの強力なオブジェクト指向拡張です：

```
Transaction tx = session.beginTransaction();

Query query = session.createQuery("select c from Cat as c where c.sex = :sex");
query.setCharacter("sex", 'F');
for (Iterator it = query.iterate(); it.hasNext();) {
    Cat cat = (Cat) it.next();
    out.println("Female Cat: " + cat.getName() );
}

tx.commit();
```

Hibernateは、オブジェクト指向の criteriaによるクエリ APIも 用意しています。これはタイプ・セーフなクエリを定式化するために使えます。 当然Hibernateは、データベースとのSQLのやりとりのすべてにおいて、 PreparedStatement とパラメータ・バインディングを使います。 余りないことですが、Hibernateで直接SQLクエリの機能を使うことや、 Session からプレーンのJDBCコネクションを取得することも可能です。

1.5. 終わりに

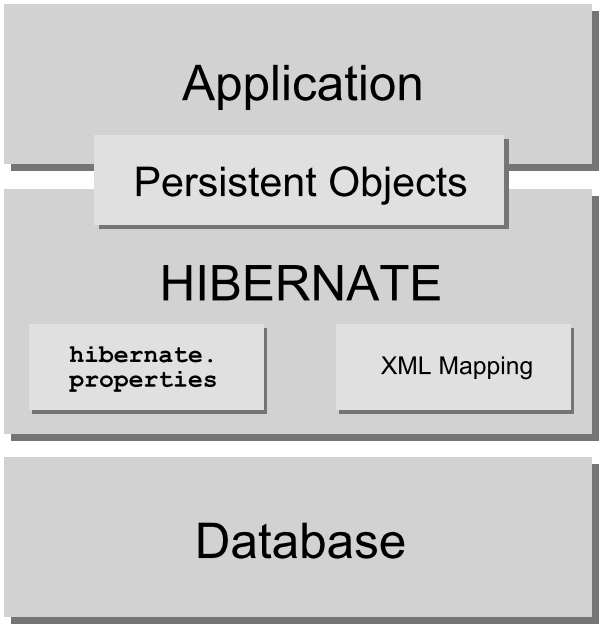
この小さなチュートリアルでは、私たちはHibernateの表面を引っ掻いたに過ぎません。 また今回の例には、サーブレットに特有のコードを含めなかったことに注意してください。 実際には自分でサーブレットを作成し、Hibernateのコードを適切に組みこまなければなりません。

データ・アクセス層として、Hibernateはアプリケーションにしっかりと統合されていることを 心に留めておいてください。 普通は、他の層のすべてが、永続機構に依存します。 この設計の意味するところを、確実に理解するようにしてください。

第2章 アーキテクチャ

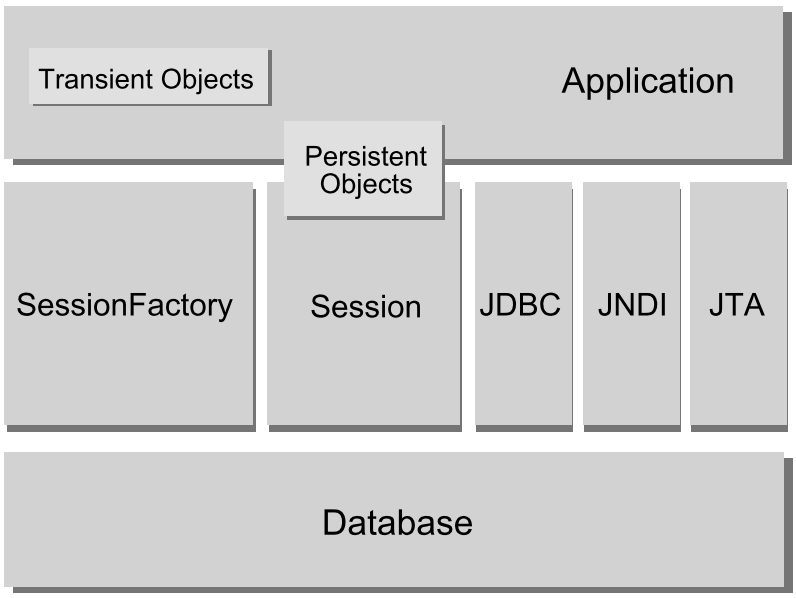
2.1. 概観

Hibernateアーキテクチャの（非常に）高いレベルからのビュー：

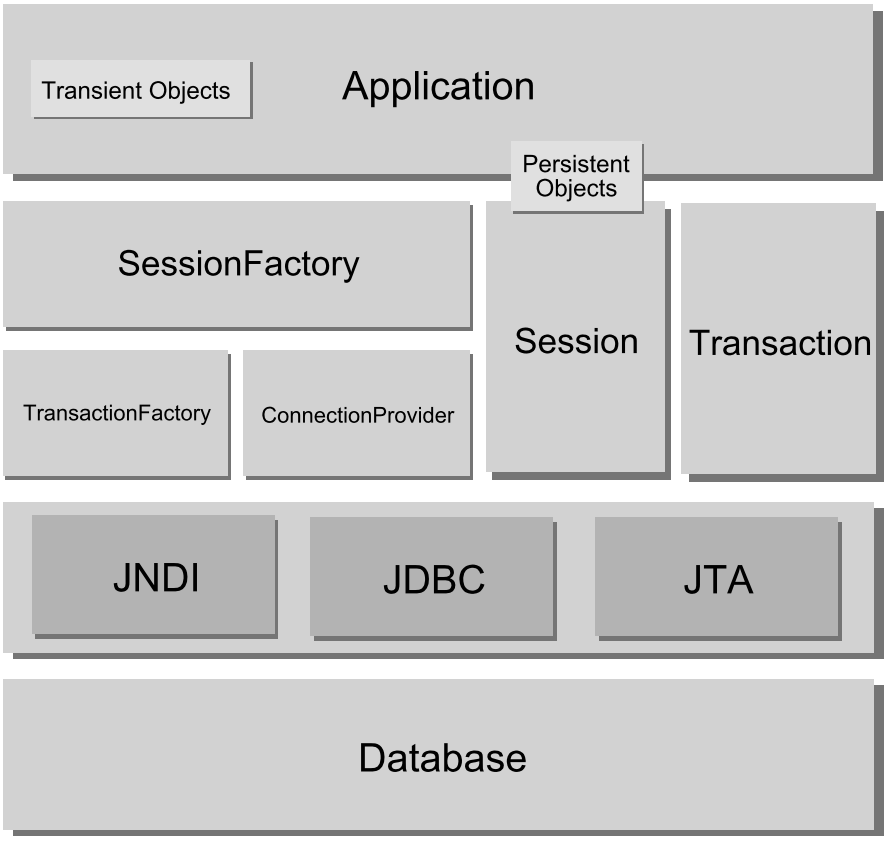


この図は、Hibernateがデータベースと設定データを使い、 アプリケーションのための永続化サービス （と永続オブジェクト）を提供することを示しています。

さらに実行時アーキテクチャの詳細なビューをお見せします。 あいにく、Hibernateは柔軟でいろいろな方法を用意しています。 そのため、対極的な2つの例をお見せします。 「軽い」アーキテクチャは、独自にJDBCコネクションを提供し、 トランザクションを管理するアプリケーションを持っています。 この方法は、Hibernate APIの最小限のサブセットを使います：



「重い」アーキテクチャは、ベースとなるJDBC/JTA APIからアプリケーションを抽象化し、Hibernateが詳細の面倒を見ます。



この図のオブジェクトの定義です：

SessionFactory (`net.sf.hibernate.SessionFactory`)
: 1つのデータベースに対応する、コンパイルされたマッピングの スレッドセーブな（更新不能の）キャッシュ。 `Session` のファクトリで、 `ConnectionProvider` のクライアント。 プロセス・レベルまたはクラスタ・レベルで、 トランザクション間で再利用可能な、データのオプションの （第2レベル）キャッシュを持つことができます。

Session (`net.sf.hibernate.Session`)
: アプリケーションと永続ストアとの対話を表す、 シングル・スレッドで短命のオブジェクト。 JDBCコネクションをラップしています。 `Transaction` のファクトリです。 永続オブジェクトの支配的な（第1レベル）キャッシュを保持します。 このキャッシュはオブジェクトのグラフのナビゲーション時や、 識別子でオブジェクトを検索するときに使われます。

Persistent Objects と Collections
: 永続状態とビジネスの機能を持つ、短命でシングル・スレッドのオブジェクト。 普通のJavaBeansかもしれませんが、特別なことは、 現在の（確実にただ1つの）`Session` と関連していることです。 `Session` がクローズされればすぐに、 それらは切り離されて他のアプリケーション層から自由に使うことができます。 （例えば、データ・トランスファ・オブジェクトとして、 プレゼンテーション層と直接やりとりできます。）

Transient Objects と Collections
: 現時点では `Session` と関連していない、 永続クラスのインスタンス。 アプリケーションでインスタンス化されて永続化される前か、 クローズされた `Session` でインスタンス化されたかの

どちらかです。

Transaction (net.sf.hibernate.Transaction)
: （オプション）作業の最小単位を指定するアプリケーションが使う、 シングル・スレッドで短命のオブジェクト。 ベースとなるJDBC、JTA、CORBAトランザクションから、アプリケーションを抽象化します。 Session は、 いくつかの Transaction を横断するかもしれません。

ConnectionProvider (net.sf.hibernate.connection.ConnectionProvider)
: （オプション）JDBCコネクション（とそのプール）のファクトリ。 アプリケーションを Datasource や DriverManager から抽象化します。 アプリケーションには公開されませんが、開発者が拡張 / 実装することは可能です。

TransactionFactory (net.sf.hibernate.TransactionFactory)
: （オプション）Transaction インスタンスのファクトリ。 アプリケーションには公開されませんが、開発者が拡張 / 実装することは可能です。

「軽い」アーキテクチャでは、アプリケーションは Transaction や TransactionFactory や ConnectionProvider を介さずに、 直接JTAやJDBCと対話することができます。

2.2. JMXとの統合

JMXはJavaコンポーネント管理のJ2EE標準です。 JMX標準のMBeanを通して、Hibernateを管理することは可能ですが、 ほとんどのアプリケーション・サーバはまだJMXをサポートしていないため、 標準でない設定機構もいくつか認められています。

JBossの中でJMXコンポーネントとしてHibernateを実行する方法については、 Hibernateのウェブサイトを見てください。

2.3. JCAのサポート

HibernateはJCAコネクタとしても設定できます。詳細についてはウェブサイトを見てください。

第3章 SessionFactoryの設定

Hibernateは、多くのさまざまな環境で使えるように設計されているので、 設定パラメータがたくさん用意されています。 さいわい、ほとんどのパラメータには適当なデフォルト値が定められており、 またさまざまなオプションを示す hibernate.properties ファイルのサンプルが付属しています。 普通はクラスパスにこのファイルを置いて、カスタマイズするだけで十分のはずです。

3.1. プログラムでの設定

net.sf.hibernate.cfg.Configuration のインスタンスは、 アプリケーションのJava型からリレーショナル・データベースへのマッピングのセットを表します。 Configuration は、 （更新不能の）SessionFactory をビルドするために使います。 マッピングはいろいろなXMLマッピング・ファイルからコンパイルされます。

Configuration インスタンスは、直接インスタンス化して取得できます。 これは（クラスパスにおいて）2つのXML設定ファイルで定義されたマッピングからの データストアの設定例です：

```
Configuration cfg = new Configuration()
    .addFile("Item.hbm.xml")
    .addFile("Bid.hbm.xml");
```

もう1つの（ときどきはより良い）方法は、getResourceAsStream() を使って マッピング・ファイルをロードする方法です。

```
Configuration cfg = new Configuration()
    .addClass(org.hibernate.auction.Item.class)
    .addClass(org.hibernate.auction.Bid.class);
```

すると Hibernate はクラスパスで、 /org/hibernate/autcion/Item.hbm.xml と /org/hibernate/autcion/Bid.hbm.xml という名前のマッピング・ファイルを探します。 この方法を使えば、ファイル名をハード・コーディングしなくてすみます。

Configuration でも、いろいろなオプションのプロパティを指定します：

```
Properties props = new Properties();
...
Configuration cfg = new Configuration()
    .addClass(org.hibernate.auction.Item.class)
    .addClass(org.hibernate.auction.Bid.class)
    .setProperties(props);
```

Configuration は設定時のオブジェクトとして考えられており、 SessionFactory が作成された後は捨てられるものと意図されています。

3.2. SessionFactoryの取得

すべてのマッピングが Configuration によって構文解析されたとき、 アプリケーションは Session インスタンスのファクトリを 取得しなければなりません。 このファクトリはすべてのアプリケーション・スレッドで共有されるものと考えられています：

```
SessionFactory sessions = cfg.buildSessionFactory();
```

しかしHibernateは、アプリケーションが SessionFactory を 複数インスタンス化することを許しています。 これは複数のデータベースをを使うときに役に立ちます。

3.3. ユーザが用意するJDBCコネクション

SessionFactory は、 ユーザが用意したJDBCコネクションで Session をオープンできます。 この設計により、アプリケーションが欲しいときにいつでも、 JDBCコネクションを取得できます：

```
java.sql.Connection conn = datasource.getConnection();
Session session = sessions.openSession(conn);

// ######################
```

アプリケーションは同じJDBCコネクションで、 2つの同時並行 Session をオープンしないように気を付けなければいけません。

3.4. Hibernateが用意するJDBCコネクション

Alternatively, you can have the SessionFactory open connections for you. The SessionFactory must be provided with JDBC connection properties in one of the following ways: もう1つの方法として、SessionFactory にコネクションをオープンさせることもできます。 SessionFactory には以下の方法のどれか1つを使い、 JDBCコネクション・プロパティを与えなければいけません：

1. java.util.Properties のインスタンスを Configuration.setProperties() に渡す。
2. hibernate.properties をクラスパスのルート・ディレクトリに置く。
3. java -Dproperty=value を使い、 System プロパティを設定する。
4. hibernate.cfg.xml に <property> 要素を含める（後で議論します）。

この方法を使えば、以下のように Session のオープンが簡単になります：

```
Session session = sessions.openSession(); // ###Session########
// ######################JDBC###################
```

すべてのHibernateプロパティの名前と意味は、 net.sf.hibernate.cfg.Environment クラスで定義されています。 それでは、JDBCコネクションの最も重要な設定について述べていきます。

以下のプロパティを設定すれば、Hibernateは java.sql.DriverManager を使って、 コネクションを取得（そしてプール）します：

表 3.1. Hibernate JDBC プロパティ

プロパティ名	目的
hibernate.connection.driver_class	JDBCドライバ・クラス
hibernate.connection.url	JDBC URL
hibernate.connection.username	データベースのユーザ
hibernate.connection.password	データベースのユーザのパスワード

プロパティ名	目的
hibernate.connection.pool_size	プールするコネクションの最大数

Hibernate独自のコネクション・プーリング・アルゴリズムは、かなり初歩的なものです。 これはHibernateを始める手助けのためのものであり、 製品のシステムでのしようを意図していません し、 パフォーマンス・テストへの使用でさえ意図していません。 最高のパフォーマンスと安定性を得るためには、サードパーティのプールを使ってください。 そしてコネクション・プールの設定で、 hibernate.connection.pool_size プロパティを置き換えてください。

C3P0はオープン・ソースのJDBCコネクション・プールで、Hibernateと一緒に配布されています。 これは lib ディレクトリにあります。 hibernate.c3p0.* プロパティを設定すれば、 Hibernateはコネクション・プールのために、組み込みの C3P0ConnectionProvider を使います。 また、Apache DBCPとProxoolも組み込みでサポートしています。 DBCPConnectionProvider を有効にするには、hibernate.dbcp.* プロパティ （DBCPコネクション・プール・プロパティ）を設定しなければいけません。 hibernate.dbcp.ps.* （DBCPステートメント・キャッシュ・プロパティ）が設定されれば、Prepared statementのキャッシュが有効になります（非常におすすめです）。 これらのプロパティについては、Apache commons-poolのドキュメントを参照してください。 またProxoolを使いたければ、hibernate.proxool.* プロパティを設定してください。

以下は、C3P0を使った例です：

```
hibernate.connection.driver_class = org.postgresql.Driver
hibernate.connection.url = jdbc:postgresql://localhost/mydatabase
hibernate.connection.username = myuser
hibernate.connection.password = secret
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=1800
hibernate.c3p0.max_statements=50
hibernate.dialect = net.sf.hibernate.dialect.PostgreSQLDialect
```

アプリケーション・サーバの内部で使うために、 HibernateはJNDIで登録された javax.sql.Datasource からコネクションを取得できます。 以下のプロパティを設定してください：

表 3.2. Hibernateデータソース・プロパティ

プロパティ名	目的
hibernate.connection.datasource	データソースのJNDI名
hibernate.jndi.url	JNDIプロバイダのURL（オプション）
hibernate.jndi.class	*InitialContextFactory* のクラス（オプション）
hibernate.connection.username	データベースのユーザ（オプション）
hibernate.connection.password	データベースのユーザのパスワード（オプション）

以下は、JNDIデータソースが提供したアプリケーション・サーバを使った例です：

```
hibernate.connection.datasource = java:/comp/env/jdbc/MyDB
hibernate.transaction.factory_class = \
    net.sf.hibernate.transaction.JTATransactionFactory
```

```
hibernate.transaction.manager_lookup_class = \
    net.sf.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = \
    net.sf.hibernate.dialect.PostgreSQLDialect
```

JNDIデータソースから取得されたJDBCコネクションは、 アプリケーション・サーバのコンテナ管理トランザクションに自動的に登録されます。

任意のコネクション・プロパティは、 ″hibernate.connnection″ の後にプロパティ名を続けることで与えられます。 例えば hibernate.connnection.charSet を使い、 charSet を指定できます。

net.sf.hibernate.connection.ConnectionProvider を実装することで、 JDBCコネクションを取得するための独自のプラグイン戦略を定義できます。 また hibernate.connection.provider_class の設定によって、 カスタムの実装を選べます。

3.5. オプションの設定プロパティ

Hibernateの実行時の振る舞いを制御するために使うプロパティが、他にもたくさんあります。 すべてがオプションで、適当なデフォルト値が設定されています。

システム・レベルのプロパティは java -Dproperty=value で設定するか hibernate.properties で定義し、 Configuration に渡される Properties では設定できません。

表 3.3. Hibernate設定プロパティ

プロパティ名	目的
hibernate.dialect	Hibernate Dialect のクラス名。 プラットフォーム依存の機能を有効にします。 例 full.classname.of.Dialect
hibernate.default_schema	生成されるSQLにおいて、与えられるスキーマ/テーブルスペースで 非修飾のテーブル名を修飾します。 例 SCHEMA_NAME
hibernate.session_factory_name	作成された後、SessionFactory がJNDIにおいて自動的にこの名前にバインドされます。 例 jndi/composite/name
hibernate.use_outer_join	アウター・ジョインによるフェッチを有効にします。推奨されません。 max_fetch_depth を使ってください。 例 true | false
hibernate.max_fetch_depth	末端が1の関連（one-to-one, many-to-one）に対して、 アウター・ジョインのツリーの「深さ」の最大値を設定します。 0 と指定すると、デフォルトのアウター・ジョインによるフェッチ

プロパティ名	目的
	が無効になります。 例 おすすめは 0 から 3 の間の値
hibernate.jdbc.fetch_size	0でない値はJDBCのフェッチ・サイズを決定します （Statement.setFetchSize() をコールします）。
hibernate.jdbc.batch_size	0でない値はHibernateによるJDBC2のバッチ更新を有効にします。 例 おすすめは 5 から 30 の間の値
hibernate.jdbc.batch_versioned_data	もしJDBCドライバが executeBatch() から正しい行数を返すなら、 このプロパティを true に設定してください （普通はこのオプションをオンにしても安全です）。 すると自動的にバージョン付けされたデータに対して、 Hibernateはバッチの DMLを使います。 デフォルトは false です。 例 true | false
hibernate.jdbc.use_scrollable_resultset	HibernateによるJDBC2スクローラブル・リザルトセットを有効にします。 このプロパティはユーザが用意するJDBCコネクションを使うときにだけ必要で、 そうでなければHibernateはコネクション・メタデータを使います。 例 true | false
hibernate.jdbc.use_streams_for_binary	binary や serializable 型をJDBCに書き込んだり 読み込んだりするときに、ストリームを使います （システム・レベルのプロパティです）。 例 true | false
hibernate.jdbc.use_get_generated_keys	挿入の後にネイティブに生成されたキーを復元するための PreparedStatement.getGeneratedKeys() の使用を有効にします。 これはJDBC3+ドライバとJRE1.4+を必要とし、 もしHibernateの識別子ジェネレータに問題が発生するようならfalseに設定してください。 デフォルトではコネクション・メタデータを使いドライバの能力を決定するようにしてください。 例 true | false
hibernate.cglib.use_reflection_optimizer	実行時リフレクションの代わりのCGLIBの使用を有効にします （システム・レベルのプロパティ） リフレクションはトラブルシューティングの

プロパティ名	目的
	ときに役立つことがあります。 オプティマイザをオフにしているときでさえ、 Hibernateには必ずCGLIBが必要なことに注意してください。 このプロパティは hibernate.cfg.xml で設定できません。 例 true | false
hibernate.jndi.*<propertyName>*	propertyName プロパティを、 JNDI InitialContextFactory に渡します。
hibernate.connection.isolation	JDBCトランザクション分離レベルを設定します。 妥当な値を調べるためには java.sql.Connection をチェックしてください。 しかし使用するデータベースが、すべての分離レベルをサポートしているとは限りません。 例 1, 2, 4, 8
hibernate.connection.*<propertyName>*	propertyName JDBC プロパティを DriverManager.getConnection() に渡します。
hibernate.connection.provider_class	カスタム ConnectionProvider のクラス名。 例 classname.of.ConnectionProvider
hibernate.cache.provider_class	カスタム CacheProvider のクラス名。 例 classname.of.CacheProvider
hibernate.cache.use_minimal_puts	書き込みを最小限にするために、第2レベル・キャッシュの操作を最適化します。 その代わりに、読み込みがより頻繁に発生するようになります （クラスタ・キャッシュに役に立ちます）。 例 true|false
hibernate.cache.use_query_cache	クエリのキャッシュを有効にします。 個々のクエリはまだキャッシャブルに設定されなければなりません。 例 true|false
hibernate.cache.query_cache_factory	カスタム QueryCache インターフェイスのクラス名。 デフォルトは組み込みの StandardQueryCache です。 例 classname.of.QueryCache
hibernate.cache.region_prefix	第2レベル・キャッシュの領域の名前に使うプリフィックス。 例 prefix

プロパティ名	目的
hibernate.transaction.factory_class	Hibernate Transaction APIと一緒に使われるTransactionFactory のクラス名。 （デフォルトでは JDBCTransactionFactory です）。 例 classname.of.TransactionFactory
jta.UserTransaction	アプリケーション・サーバからJTA UserTransaction を取得するために JTATransactionFactory に使われるJNDI名。 例 jndi/composite/name
hibernate.transaction.manager_lookup_class	TransactionManagerLookup のクラス名。 JTA環境において、JVMレベルのキャッシュを有効にするために必要です。 例 classname.of.TransactionManagerLookup
hibernate.query.substitutions	Hibernateクエリのトークンから SQLのトークンへのマッピング （例えばトークンは関数やリテラルの名前かもしれません）。 例 hqlLiteral=SQL_LITERAL, hqlFunction=SQLFUNC
hibernate.show_sql	すべてのSQL文をコンソールに書き出します。 例 true | false
hibernate.hbm2ddl.auto	SessionFactory が作成されるとき、 自動的にスキーマDDLをデータベースにエクスポートします。 create-drop と指定されると、SessionFactory が明示的にクローズされるときに データベース・スキーマがドロップされます。 例 update | create | create-drop

3.5.1. SQL方言

hibernate.dialect プロパティには、 使用するデータベースの正しい net.sf.hibernate.dialect.Dialect のサブクラスを、 必ず指定すべきです。 もし native または sequence 主キー生成や （例えば Session.lock() や Query.setLockMode() で） 悲観的ロックを使いたいのでなければ、厳密に言うと必須ではありません。 しかし方言を指定すれば、Hibernateは上述したプロパティのいくつかについて、 より適切なデフォルト値を使います。 そうすれば、それらを手作業で設定する手間が省けます。

表 3.4. Hibernate SQL方言 (**hibernate.dialect**)

RDBMS	方言
DB2	net.sf.hibernate.dialect.DB2Dialect
DB2 AS/400	net.sf.hibernate.dialect.DB2400Dialect
DB2 OS390	net.sf.hibernate.dialect.DB2390Dialect
PostgreSQL	net.sf.hibernate.dialect.PostgreSQLDialect
MySQL	net.sf.hibernate.dialect.MySQLDialect
Oracle (any version)	net.sf.hibernate.dialect.OracleDialect
Oracle 9/10g	net.sf.hibernate.dialect.Oracle9Dialect
Sybase	net.sf.hibernate.dialect.SybaseDialect
Sybase Anywhere	net.sf.hibernate.dialect.SybaseAnywhereDialect
Microsoft SQL Server	net.sf.hibernate.dialect.SQLServerDialect
SAP DB	net.sf.hibernate.dialect.SAPDBDialect
Informix	net.sf.hibernate.dialect.InformixDialect
HypersonicSQL	net.sf.hibernate.dialect.HSQLDialect
Ingres	net.sf.hibernate.dialect.IngresDialect
Progress	net.sf.hibernate.dialect.ProgressDialect
Mckoi SQL	net.sf.hibernate.dialect.MckoiDialect
Interbase	net.sf.hibernate.dialect.InterbaseDialect
Pointbase	net.sf.hibernate.dialect.PointbaseDialect
FrontBase	net.sf.hibernate.dialect.FrontbaseDialect
Firebird	net.sf.hibernate.dialect.FirebirdDialect

3.5.2. アウター・ジョインによるフェッチ

データベースがANSIやOracleスタイルのアウター・ジョインをサポートしていれば、 （可能な限りデータベース自体に仕事を任せ） データベースとのやりとりの回数を制限する アウター・ジョインによるフェッチ によって、 パフォーマンスを改善できるかもしれません。 アウター・ジョインにより、many-to-one, one-to-many, one-to-one関連で接続されたオブジェクトのグラフを、 SQL SELECT 1つだけで復元できます。

デフォルトではオブジェクトの取得は、リーフ・オブジェクト、 コレクション、プロキシ・オブジェクト、循環が発生するところで終わります。

特定の関連 に対しては、 XMLマッピングで outer-join 属性を設定することで、 フェッチを有効または無効に（そしてデフォルトの振る舞いをオーバーライド）できます。

hibernate.max_fetch_depth プロパティを 0 に設定することで、 アウター・ジョインによるフェッチを グローバルに 無効にできます。 1 以上に設定すると、すべてのone-to-oneとmany-to-one関

連に対して、 アウター・ジョインによるフェッチが可能になります。 これらのアウター・ジョインは、デフォルトでは auto に設定されています。 しかし特定の関連それぞれについて明示的に宣言しなければ、 one-to-many関連とコレクションはアウター・ジョインでフェッチされません。 この振る舞いは、Hibernateクエリで実行時にオーバーライドすることもできます。

3.5.3. バイナリ・ストリーム

OracleはJDBCドライバとの間でやりとりされる byte 配列のサイズを制限します。 binary や serializable 型の大きなインスタンスを使いたければ、 hibernate.jdbc.use_streams_for_binary を有効にしてください。 ただし これはJVMレベルの設定だけです 。

3.5.4. カスタム CacheProvider

インターフェイス net.sf.hibernate.cache.CacheProvider を実装すると、 JVMレベル（またはクラスタ）第2レベル・キャッシュ・システムを統合できます。 hibernate.cache.provider_class を設定すると、 カスタムの実装を選べます。

3.5.5. トランザクション戦略の設定

Hibernate Transaction APIを使いたければ、 プロパティ hibernate.transaction.factory_class を設定して、 Transaction インスタンスのためのファクトリ・クラスを 指定しなければなりません。 Transaction APIは、ベースとなるトランザクション機構を隠蔽し、 管理された環境と管理されていない環境の両方で、Hibernateのコードの実行を可能にします。

標準的な2つの（組み込みの）選択肢があります：

net.sf.hibernate.transaction.JDBCTransactionFactory
: データベース（JDBC）トランザクションに委譲します（デフォルト）

net.sf.hibernate.transaction.JTATransactionFactory
: JTAに委譲します（既存のトランザクションが実行中なら、 Session はそのコンテキストで動作します。 そうでなければ、新しいトランザクションを開始します。）

独自のトランザクション戦略も定義できます （例えばCORBAトランザクション・サービスのための戦略）。

JTA環境で更新不能のデータのJVMレベル・キャッシュを使いたければ、 これはJ2EEに対して標準化されていないため、 JTA TransactionManager を取得するための戦略を 指定しなければいけません：

表 3.5. JTAトランザクション・マネージャ

トランザクション・ファクトリ	アプリケーション・サーバ
net.sf.hibernate.transaction.JBossTransactionManagerLookup	JBoss
net.sf.hibernate.transaction.WeblogicTransactionManagerLookup	Weblogic
net.sf.hibernate.transaction.WebSphereTransactionManagerLookup	WebSphere
net.sf.hibernate.transaction.OrionTransactionManagerLookup	Orion

トランザクション・ファクトリ	アプリケーション・サーバ
net.sf.hibernate.transaction.ResinTransactionManagerLookup	Resin
net.sf.hibernate.transaction.JOTMTransactionManagerLookup	JOTM
net.sf.hibernate.transaction.JOnASTransactionManagerLookup	JOnAS
net.sf.hibernate.transaction.JRun4TransactionManagerLookup	JRun4
net.sf.hibernate.transaction.BESTransactionManagerLookup	Borland ES

3.5.6. JNDIバインドの SessionFactory

JNDIバインドのHibernate SessionFactory は、 ファクトリの検索と新しい Session の検索を単純化できます。

SessionFactory を JNDI 名前空間にバインドさせたければ、 プロパティ hibernate.session_factory_name を使い、 名前（例 java:comp/env/hibernate/SessionFactory）を指定します。 このプロパティを設定しなければ、SessionFactory はJNDIにバインドされません。これはread-onlyのJNDIデフォルト実装の環境において特に役に立ちます。例 Tomcat）

SessionFactory をJNDIにバインドさせるとき、 Hibernateはイニシャル・コンテキストをインスタンス化するために hibernate.jndi.url, hibernate.jndi.class の値を使います。 それらの値が指定されなければ、デフォルトの InitialContext が使われます。

JNDIを使うことにしたなら、JNDIルック・アップを使い、 EJBや他のユーティリティ・クラスは SessionFactory を取得できます。

3.5.7. クエリ言語の置き換え

hibernate.query.substitutions を使い、 新しいHibernateクエリ・トークンを定義することができます。 例えば：

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

これはトークン true と false を 生成されるSQLにおいて整数リテラルに翻訳します。

```
hibernate.query.substitutions toLowercase=LOWER
```

これはSQLの LOWER 関数の名前の付け替えを可能にします。

3.6. ロギング

HibernateはApache commons-loggingを使い、いろいろなイベントをログに取ります。

commons-loggingサービスは（クラスパスに log4j.jar を含めれば）Apache Log4jに、 また（JDK1.4 かそれ以上で実行させれば）JDK1.4 logging に直接出力します。 Log4jは http://jakarta.apache.org からダウンロードできます。 Log4jを使うためには、クラスパスに log4j.properties ファイルを配置する必要があります。 例のプロパティ・ファイルはHibernateと

一緒に配布され、それは src/ ディレクトリにあります。

Hibernateのログ・メッセージになれることを強くおすすめします。 Hibernateのログは読みにくくならずにできる限り詳細になるように努力されています。 これは必須のトラブルシューティング・デバイスです。 また上で述べたようにSQLロギングを有効にすることを忘れないでください （hibernate.show_sql）。 パフォーマンスの問題を探すときこれが最初のステップとなります。

3.7. NamingStrategy の実装

インターフェイス net.sf.hibernate.cfg.NamingStrategy を使うと データベース・オブジェクトとスキーマ要素のための「命名標準」を指定できます。

Javaの識別子からデータベースの識別子を自動生成することや、 マッピング・ファイルで与えた「論理的な」カラムとテーブル名から 「物理的な」テーブルとカラム名を生成することのためのルールを用意することができます。 この機能は繰り返しの雑音（例えば TBL_ プリフィックス）を取り除き、 マッピング・ドキュメントの冗長さを減らすことに役立ちます。 Hibernateが使うデフォルトの戦略はかなり最小限に近いものです。

マッピングを追加する前に Configuration.setNamingStrategy() をコールすることで 以下のように異なる戦略を指定することができます：

```
SessionFactory sf = new Configuration()
    .setNamingStrategy(ImprovedNamingStrategy.INSTANCE)
    .addFile("Item.hbm.xml")
    .addFile("Bid.hbm.xml")
    .buildSessionFactory();
```

net.sf.hibernate.cfg.ImprovedNamingStrategy is a built-in strategy that might be a useful starting point for some applications. net.sf.hibernate.cfg.ImprovedNamingStrategy は組み込みの戦略です。 これはいくつかのアプリケーションにとって有用な開始点となるかもしれません。

3.8. XML設定ファイル

もう1つの方法は hibernate.cfg.xml という名前のファイルで 十分な設定を指定する方法です。 このファイルは hibernate.properties ファイルの代わりとなります。 もし両方のファイルがあれば、プロパティが置き換えられます。

XML設定ファイルはデフォルトで CLASSPATH に配置されることを期待されています。 これが例です：

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 2.0//EN"

 "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">

<hibernate-configuration>

    <!-- /jndi/name#############SessionFactory###### -->
    <session-factory
        name="java:comp/env/hibernate/SessionFactory">

        <!-- ##### -->
        <property name="connection.datasource">my/first/datasource</property>
        <property name="dialect">net.sf.hibernate.dialect.MySQLDialect</property>
```

```
        <property name="show_sql">false</property>
        <property name="use_outer_join">true</property>
        <property name="transaction.factory_class">
            net.sf.hibernate.transaction.JTATransactionFactory
        </property>
        <property name="jta.UserTransaction">java:comp/UserTransaction</property>

        <!-- ########## -->
        <mapping resource="org/hibernate/auction/Item.hbm.xml"/>
        <mapping resource="org/hibernate/auction/Bid.hbm.xml"/>

    </session-factory>

</hibernate-configuration>
```

Hibernateの設定は以下のように簡単になります。

```
SessionFactory sf = new Configuration().configure().buildSessionFactory();
```

また以下のようにすると異なるXML設定ファイルを使うことができます。

```
SessionFactory sf = new Configuration()
    .configure("/my/package/catdb.cfg.xml")
    .buildSessionFactory();
```

第4章 永続クラス

永続クラスは、ビジネスの問題のエンティティを実装する、アプリケーションのクラスです （例 Eコマース・アプリケーションの顧客や注文）。 名前からわかる通り、永続クラスは一時的なインスタンスと、 データベースに格納される永続インスタンスを含んでいます。

このクラスがいくつかの簡単なルールを守れば、Hibernateは最も良く動作します。 これはプレーンの古いJavaオブジェクト（POJO）プログラミング・モデルとしても知られています。

4.1. 簡単なPOJOの例

ほとんどのJavaアプリケーションには、ネコ科を表す永続クラスが必要です。

```
package eg;
import java.util.Set;
import java.util.Date;

public class Cat {
    private Long id; // ###
    private String name;
    private Date birthdate;
    private Cat mate;
    private Set kittens
    private Color color;
    private char sex;
    private float weight;

    private void setId(Long id) {
        this.id=id;
    }
    public Long getId() {
        return id;
    }

    void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }

    void setMate(Cat mate) {
        this.mate = mate;
    }
    public Cat getMate() {
        return mate;
    }

    void setBirthdate(Date date) {
        birthdate = date;
    }
    public Date getBirthdate() {
        return birthdate;
    }
    void setWeight(float weight) {
        this.weight = weight;
    }
    public float getWeight() {
        return weight;
    }

    public Color getColor() {
        return color;
    }
```

```
    void setColor(Color color) {
        this.color = color;
    }
    void setKittens(Set kittens) {
        this.kittens = kittens;
    }
    public Set getKittens() {
        return kittens;
    }
    // addKitten##Hibernate############
    public void addKitten(Cat kitten) {
        kittens.add(kitten);
    }
    void setSex(char sex) {
        this.sex=sex;
    }
    public char getSex() {
        return sex;
    }
}
```

以下に守るべき4つのルールがあります：

4.1.1. 永続フィールドに対するアクセサとミューテータを定義する

`Cat` はすべての永続フィールドに対する アクセサ・メソッドを定義しています。 他の多くのORMツールはインスタンス変数を直接永続化します。 私たちは永続化機構から実装の詳細を分離する方がずっと良いことだと信じています。 HibernateはJavaBeansスタイルのプロパティを永続化し、`getFoo`, `isFoo`, `setFoo` のような形式のメソッド名を認識します。

プロパティをパブリックで定義する必要は ありません。 Hibernateはデフォルト, `protected`, `private` のget / setのペアを持つプロパティを永続化できます。

4.1.2. デフォルト・コンストラクタを実装する

`Cat` には暗黙的なデフォルト（引数なし）コンストラクタがあります。 すべての永続クラスはデフォルト・コンストラクタ（パブリックでなくても構いません） を持たなければいけません。 そのためHibernateは `Constructor.newInstance()` を使って、 それらをインスタンス化できます。

4.1.3. 識別子プロパティを用意する(オプション)

`Cat` には `id` というプロパティがあります。 プロパティはどのような名前でも構いませんし、 型もどんなプリミティブ型でも、プリミティブの「ラッパー」型でも、 `java.lang.String` や `java.util.Date` 型でも構いません。 （レガシー・データベースのテーブルに複合キーがあれば、これらのプロパティを持つユーザ定義クラスを使うこともできます。 以下の複合識別子の節を見てください。）

識別子プロパティはオプションです。 これを使わずに、Hibernateが内部的にオブジェクトの識別子を追いかけるようにもできます。 しかし識別子を使うことが、多くのアプリケーションにとって良い（そしてとても一般的な） 設計の決定となります。

さらにいくつかの機能は、識別子プロパティを定義するクラスだけで利用することができます：

- カスケード更新（「ライフサイクル・オブジェクト」を見てください）
- `Session.saveOrUpdate()`

すべての永続クラスで一貫した名前の識別子プロパティを定義することをおすすめします。 さらにnullでない（つまりプリミティブ型でない）型を使うことをおすすめします。

4.1.4. finalクラスにしない（オプション）

Hibernateの核となる機能の プロキシ は finalではない永続クラスに依存するか、 またはすべてのパブリック・メソッドを定義するインターフェイスに依存しなければなりません。

Hibernateはインターフェイスを実装しない `final` クラスを永続化できますが、 プロキシを使うことはできません。 それはパフォーマンス・チューニングのオプションをいくらか制限します。

4.2. 継承の実装

サブクラスは1番目と2番目のルールに従わなければいけません。 それは識別子プロパティを `Cat` から継承します。

```
package eg;

public class DomesticCat extends Cat {
        private String name;

        public String getName() {
                return name;
        }
        protected void setName(String name) {
                this.name=name;
        }
}
```

4.3. `equals()` と `hashCode()` の実装

永続クラスの混合オブジェクト（例えば `Set` で）を使うつもりなら、 `equals()` と `hashCode()` をオーバーライドしなければいけません。

これは2つの異なる *`Session`* で、 これらのオブジェクトがロードされるときだけにあてはまります。 というのはHibernateは1つの *`Session`* でJVMアイデンティティ （ *`a == b`* , *`equals()`* のデフォルトの実装） だけを保証するからです。

`a` と `b` の両方のオブジェクトが、 同じデータベースの行だったとしても（識別子として同じ主キーを持つ）、 ある特定の `Session` コンテキストの外で それらが同じJavaインスタンスであることを保証できません。

最も明快な方法は、両方のオブジェクトの識別子の値の比較に、 `equals()` / `hashCode()` を実装する方法です。 その値が同じなら両者は同じデータベースの行でなければならず、 そのためそれらは等しくなります （もし両者が `Set` に加えられれば、 `Set` に1つの要素だけを持つことになります）。 あいにく私たちはこの方法を使うことができません。 Hibernateは永続であるオブジェクトだけに識別子の値を割り当て、 新しく作成されたインスタンスは識別子の値を持ちません。 私たちは ビジネス・キー等価性 を使い、 `equals()` と `hashCode()` を実装することをおすすめします。

ビジネス・キー等価性とは、 `equals()` メソッドがビジネス・キーを形作るプロパティだけを比較することを意味します。 キーは現実世界の私たちのインスタンスを識別します （自然な 候補キー

です）：

```
public class Cat {

    ...
    public boolean equals(Object other) {
        if (this == other) return true;
        if (!(other instanceof Cat)) return false;

        final Cat cat = (Cat) other;

        if (!getName().equals(cat.getName())) return false;
        if (!getBirthday().equals(cat.getBirthday())) return false;

        return true;
    }

    public int hashCode() {
        int result;
        result = getName().hashCode();
        result = 29 * result + getBirthday().hashCode();
        return result;
    }

}
```

候補キー（この例では名前と誕生日の複合物）は ある特定の比較操作だけに妥当であることを心に留めておいてください （おそらく1つのユースケースだけです）。 普通は現実の主キーに当てはまる、安定criteriaは必要ありません。

4.4. Lifecycleコールバック

オプションとして永続クラスは `Lifecycle` インターフェイスを実装するかもしれません。 それは永続オブジェクトがセーブやロードの後、そして削除や更新の前に、 必要な初期化/クリーンアップを実行することを可能にするコールバックを提供します。

しかしHibernateはでしゃばりすぎないもう一つの `Interceptor` も用意しています。

```
public interface Lifecycle {
        public boolean onSave(Session s) throws CallbackException;   (1)
        public boolean onUpdate(Session s) throws CallbackException; (2)
        public boolean onDelete(Session s) throws CallbackException; (3)
        public void onLoad(Session s, Serializable id);              (4)
}
```

(1) `onSave` – オブジェクトがセーブまたは挿入される直前に コールされます

(2) `onUpdate` – オブジェクトが更新される直前 （オブジェクトが `Session.update()` に渡されるとき） にコールされます

(3) `onDelete` – オブジェクトが削除される直前にコールされます

(4) `onLoad` – オブジェクトがロードされた直後にコールされます

`onSave()`, `onDelete()`, `onUpdate()` は依存オブジェクトのカスケード・セーブと カスケード削除のために使うことができます。 これはマッピング・ファイルでカスケード操作を定義する代わりの方法となります。 `onLoad()` は永続状態から オブジェクトの一時的プロパティを初期化することに使うことができます。 しかし `Session` インターフェイスは このメソッドの内部から起動されないかもしれないので、 それを依存オブジェクトのロードのために使うことはできません。 他に考えられる `onLoad()`, `onSave()`, `onUpdate()` の使い方は、 後で使うために現在の `Session` への参照を保存しておくことです。

onUpdate() はオブジェクトの永続状態が更新されるたびに 必ずコールされるわけではないことに注意してください。 それがコールされるのは、一時的オブジェクトが Session.update() に渡されるときだけです。

もし onSave(), onUpdate(), onDelete() が true を返すなら、 その操作は中止されます。 CallbackException がスローされれば、 その操作は中止され、アプリケーションに例外が返されます。

onSave() はnativeキー生成が使われるときを除いて、 オブジェクトに識別子が割り当てられた後にコールされることに注意してください。

4.5. Validatableコールバック

永続クラスが状態を永続化する前に不変式をチェックする必要があれば、 以下のインターフェイスを実装することができます:

```
public interface Validatable {
        public void validate() throws ValidationFailure;
}
```

不変式が満たされなければ、オブジェクトは ValidationFailure をスローすべきです。 Validatable のインスタンスは、 validate() の内部で状態を更新すべきではありません。

Lifecycle インターフェイスのコールバック・メソッドとは違い、 validate() は何回コールされるかわかりません。 アプリケーションはビジネスの機能を validate() に依存させるべきではありません。

4.6. XDocletマークアップの使用

次の章では簡単で読みやすいXMLフォーマットを使い、 Hibernateのマッピングがどのように表現されるかをお見せします。 Hibernateユーザの多くはXDocletの @hibernate.tags を使い、 マッピング情報を直接ソースコードに埋め込む方法を好みます。 厳密にはこれはXDocletの領分だと考えられるので、 この方法についてはこのドキュメントでは述べません。 しかしXDocletマッピングを使った Cat の例を以下にお見せします。

```
package eg;
import java.util.Set;
import java.util.Date;

/**
 * @hibernate.class
 *  table="CATS"
 */
public class Cat {
    private Long id; // ###
    private Date birthdate;
    private Cat mate;
    private Set kittens
    private Color color;
    private char sex;
    private float weight;

    /**
     * @hibernate.id
     *  generator-class="native"
```

```
     *  column="CAT_ID"
     */
    public Long getId() {
        return id;
    }
    private void setId(Long id) {
        this.id=id;
    }

    /**
     * @hibernate.many-to-one
     *  column="MATE_ID"
     */
    public Cat getMate() {
        return mate;
    }
    void setMate(Cat mate) {
        this.mate = mate;
    }

    /**
     * @hibernate.property
     *  column="BIRTH_DATE"
     */
    public Date getBirthdate() {
        return birthdate;
    }
    void setBirthdate(Date date) {
        birthdate = date;
    }
    /**
     * @hibernate.property
     *  column="WEIGHT"
     */
    public float getWeight() {
        return weight;
    }
    void setWeight(float weight) {
        this.weight = weight;
    }

    /**
     * @hibernate.property
     *  column="COLOR"
     *  not-null="true"
     */
    public Color getColor() {
        return color;
    }
    void setColor(Color color) {
        this.color = color;
    }
    /**
     * @hibernate.set
     *  lazy="true"
     *  order-by="BIRTH_DATE"
     * @hibernate.collection-key
     *  column="PARENT_ID"
     * @hibernate.collection-one-to-many
     */
    public Set getKittens() {
        return kittens;
    }
    void setKittens(Set kittens) {
        this.kittens = kittens;
    }
    // addKitten##Hibernate############
    public void addKitten(Cat kitten) {
        kittens.add(kitten);
    }
```

```
    /**
     * @hibernate.property
     *  column="SEX"
     *  not-null="true"
     *  update="false"
     */
    public char getSex() {
        return sex;
    }
    void setSex(char sex) {
        this.sex=sex;
    }
}
```

第5章 O/Rマッピングの基本

5.1. マッピング定義

オブジェクト/リレーショナル・マッピングは、XMLドキュメントで定義します。 マッピング・ドキュメントは、読みやすく手作業で編集しやすいようにデザインされています。 マッピング言語はJava中心、つまりテーブル定義ではなく、 永続クラス定義に基づいて構築されるマッピングとなっています。

Hibernateユーザの多くはXMLマッピングを手作業で行いますが、 XDoclet, Middlegen, AndroMDAというようなマッピング・ドキュメント生成ツールが いくつか存在します。

例題のマッピングから始めましょう：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping package="eg">

        <class name="Cat" table="CATS" discriminator-value="C">
                <id name="id" column="uid" type="long">
                        <generator class="hilo"/>
                </id>
                <discriminator column="subclass" type="character"/>
                <property name="birthdate" type="date"/>
                <property name="color" not-null="true"/>
                <property name="sex" not-null="true" update="false"/>
                <property name="weight"/>
                <many-to-one name="mate" column="mate_id"/>
                <set name="kittens">
                        <key column="mother_id"/>
                        <one-to-many class="Cat"/>
                </set>
                <subclass name="DomesticCat" discriminator-value="D">
                        <property name="name" type="string"/>
                </subclass>
        </class>

        <class name="Dog">
                <!-- ###Dog########### -->
        </class>

</hibernate-mapping>
```

それでは、マッピング・ドキュメントの内容について述べていきます。 ただし、Hibernateが実行時に使うドキュメント要素と属性についてだけ述べます。 マッピング・ドキュメントは、スキーマ・エクスポート・ツールがエクスポートする データベース・スキーマに影響を与える、 いくつかのオプションの属性と要素も含みます（例えば `not-null` 属性）。

5.1.1. Doctype

XMLマッピングでは、必ずお見せしたようなdoctypeを定義すべきです。 実際のDTDは、上記のURLまたは `hibernate-x.x.x/src/net/sf/hibernate` または `hibernate.jar` ディレクトリにあります。 Hibernateは常に、クラスパスでDTDを探し始めます。

5.1.2. hibernate-mapping

この要素には、オプションの属性が3つあります。 schema 属性では、 このマッピングから参照するテーブルが、その名前のスキーマに属することを指定します。 指定されると、テーブル名は与えられたスキーマ名で修飾されます。 そうでなければ、テーブル名は修飾されません。 default-cascade 属性では、 cascade 属性を指定していないプロパティやコレクションを、 どのカスケード・スタイルと解釈すべきかを指定します。 auto-import 属性は、 クエリ言語で、修飾されていないクラス名をデフォルトで使えるようにします。

```
<hibernate-mapping
        schema="schemaName"                         (1)
        default-cascade="none|save-update"          (2)
        auto-import="true|false"                    (3)
        package="package.name"                      (4)
 />
```

(1) schema（オプション）：データベース・スキーマの名前。

(2) default-cascade（オプション - デフォルトは none）： デフォルトのカスケード・スタイル。

(3) auto-import （オプション - デフォルトは true）： クエリ言語で（このマッピングのクラスの） 修飾されていないクラス名を使えるかどうかを指定します。

(4) package（オプション）： マッピング・ドキュメントの中の修飾されていないクラス名に使う、 パッケージ・プリフィックスを指定します。

（修飾されていない）同じ名前の永続クラスが2つあるなら、 auto-import="false" を設定すべきです。 もし2つのクラスに同じ「インポートされた」名前を割り当てようとすると、 Hibernateはエラーをスローします。

5.1.3. class

class 要素で、永続クラスを定義できます：

```
<class
        name="ClassName"                            (1)
        table="tableName"                           (2)
        discriminator-value="discriminator_value"   (3)
        mutable="true|false"                        (4)
        schema="owner"                              (5)
        proxy="ProxyInterface"                      (6)
        dynamic-update="true|false"                 (7)
        dynamic-insert="true|false"                 (8)
        select-before-update="true|false"           (9)
        polymorphism="implicit|explicit"            (10)
        where="arbitrary sql where condition"       (11)
        persister="PersisterClass"                  (12)
        batch-size="N"                              (13)
        optimistic-lock="none|version|dirty|all"    (14)
        lazy="true|false"                           (15)
/>
```

(1) name：永続クラス（またはインターフェイス）の完全修飾Javaクラス名。

(2) table：そのデータベース・テーブルの名前。

(3) discriminator-value（オプション - デフォルトはクラス名）： ポリモーフィックな振る舞いのために使われる、個々のサブクラスを区別するための値。 値は null か not null のいずれかです。

(4) mutable（オプション - デフォルトは true）： クラスのインスタンスが更新可能（不可能）であることを指定します。

(5) schema（オプション）： ルートの <hibernate-mapping> 要素で指定されたスキーマ名を、 オーバーライドします。

(6) proxy（オプション）： lazy初期化プロキシに使うインターフェイスを指定します。 永続化するクラスの名前そのものを指定できます。

(7) dynamic-update（オプション - デフォルトは false）： 実行時に、値の更新されたカラムだけに対する UPDATE SQLを生成することを指定します。

(8) dynamic-insert（オプション - デフォルトは false）： 実行時に、nullでないカラムだけを含む INSERT SQLを生成することを指定します。

(9) select-before-update（オプション - デフォルトは false）： オブジェクトが実際に更新されたのが確実でなければ、 HibernateがSQL UPDATE を 決して実行しない ことを指定します。 特定の場合（実際には、一時的オブジェクトが update() を使い、 新しいSessionに関連付けられているときだけ）、 実際に UPDATE が必要かどうかを決定するために、 余分なSQL SELECT が実行されることを意味します。

(10) polymorphism （オプション - デフォルトは implicit）： 暗黙的(implicit)か明示的(explicit)、どちらのポリモーフィズムを使うかを決定します。

(11) where（オプション）： このクラスのオブジェクトを復元するときに使われる、 任意のSQL WHERE 条件を指定します。

(12) persister（オプション）： カスタム ClassPersister を指定します。

(13) batch-size（オプション - デフォルトは 1）： 識別子でこのクラスのインスタンスを復元するときの「バッチ・サイズ」を指定します。

(14) optimistic-lock（オプション - デフォルトは version）： 楽観的ロック戦略を指定します。

(15) lazy（オプション）： lazy="true" と設定することは、 #### インターフェイスとして 永続化するクラスの名前そのものを指定することと同じです。

永続クラスの名前にインターフェイスを指定しても、全く問題ありません。 <subclass> 要素で、そのインターフェイスの実装クラスを定義します。 どんな static 内部クラスでも永続化できます。 例えば eg.Foo$Bar のように、標準形式を使いクラス名を指定すべきです。

mutable="false" と指定された更新不能クラスは、 アプリケーションから更新や削除をできません。 これにより、Hibernateのパフォーマンスが少しだけ良くなります。

オプションの proxy 属性を指定すると、 クラスの永続インスタンスのlazy初期化ができるようになります。 Hibernateは最初に、名前付きのインターフェイスを実装したCGLIBプロキシを返します。 実際の永続オブジェクトは、プロキシのメソッドが起動されたときにロードされます。 以下の「lazy初期化のためのプロキシ」を見てください。

暗黙的(Implicit) ポリモーフィズムとは、 スーパークラスや実装するインターフェイスを指すクエリによって、 そのクラスのインスタンスが返されることを意味します。 そしてそのサブクラスのインスタンスは、そのクラス自体の名前が指定されたときだけ返されます。 明示的(Explicit) ポリモーフィズムとは、 クラス名が明示的に指定されたときにだけ、そのインスタンスが返されることを意味します。 そして <class> 要素の中で <subclass> や <joined-subclass> とマッピングされているサブクラスのインスタンスだけが返されます。 ほとんどの用途ではデフォルトの polymorphism="implicit" が適切です。 明示的ポリモーフィズムは、異なった2つのクラスが同じテーブルにマッピングされているときに 役立ちます （これはテーブルのカラムのサブセットを含む「軽量」クラスを可能にします）。

persister 属性を使うと、 クラスの永続戦略をカスタマイズすることができます。 例えば net.sf.hibernate.persister.EntityPersister のサブクラスを指定することができ、 また例えばストアド・プロシージャ・コール、フラット・ファイルのシリアライゼーション、 LDAPなどを通して永続性を実装する net.sf.hibernate.persister.ClassPersister インターフェイスの 完全に新しい

実装を用意することさえできます。 簡単な例として net.sf.hibernate.test.CustomPersister を見てください（これは Hashtable の「永続化」です）。

dynamic-update と dynamic-insert の設定は、 サブクラスに継承されないことに注意してください。 そのため <subclass> や <joined-subclass> 要素を指定することもできます。 この設定はパフォーマンスを向上させることも、低下させることもあります。 賢く使ってください。

select-before-update を使うと、 普通はパフォーマンスが低下します。 不必要にデータベース更新トリガがコールされないようにするためには 非常に有効です。

dynamic-update を有効にすれば、 楽観的ロック戦略を選ぶことになります：

- version バージョン/タイムスタンプ・カラムをチェックします
- all すべてのカラムをチェックします
- dirty 変更されたカラムをチェックします
- none 楽観的ロックを使いません

Hibernateで楽観的ロックを使うなら、 バージョン/タイムスタンプ・カラムを使うことを 非常に強くおすすめします。 これはパフォーマンスの点から見て最適な戦略です。 さらにSessionの外での修正 （つまり Session.update() が使われるとき） を正確に扱うことのできる唯一の戦略でもあります。 どの unsaved-value 戦略を選ぶとしても、 バージョンまたはタイムスタンプ・プロパティはnullではいけないことを 心に留めておいてください。 そうでなければ、インスタンスは一時的なものだと受け取られてしまいます。

5.1.4. id

マッピングするクラスには、データベース・テーブルの主キーカラムを定義しなければ なりません。 ほとんどのクラスにはインスタンスのユニークな識別子を保持する JavaBeansスタイルのプロパティもあります。 <id> 要素は、 そのプロパティから主キーカラムへのマッピングを定義します。

```
<id
        name="propertyName"                          (1)
        type="typename"                              (2)
        column="column_name"                         (3)
        unsaved-value="any|none|null|id_value"   (4)
        access="field|property|ClassName">       (5)

        <generator class="generatorClass"/>
</id>
```

(1) name（オプション）：識別子プロパティの名前。
(2) type（オプション）：Hibernate型を示す名前。
(3) column（オプション - デフォルトはプロパティ名）： 主キーカラムの名前。
(4) unsaved-value（オプション - デフォルトは null）： インスタンスが、新しくインスタンス化された （セーブされていない）ことを示す識別子プロパティの値。 以前のSessionでセーブまたはロードされた一時的インスタンスと区別するために 使います。
(5) access（オプション - デフォルトは property）： プロパティの値へのアクセスに、Hibernateが使う戦略

name 属性がなければ、クラスには識別子プロパティがないものとみなされます。

unsaved-value 属性は重要です。 クラスの識別子プロパティがデフォルトで null でなければ、 実際のデフォルト値を指定すべきです。

複合キーを持つレガシー・データにアクセスするために、 <composite-id> という代わりの方法があります。 他の用途への使用は全くおすすめできません。

5.1.4.1. generator

必須の <generator> 子要素は、 永続クラスのインスタンスのユニークな識別子を生成するために使う、Javaクラスを指定します。 ジェネレータ・インスタンスを設定または初期化するためにパラメータが必要なら、 <param> 要素を使い、渡せます。

```
<id name="id" type="long" column="uid" unsaved-value="0">
        <generator class="net.sf.hibernate.id.TableHiLoGenerator">
                <param name="table">uid_table</param>
                <param name="column">next_hi_value_column</param>
        </generator>
</id>
```

すべてのジェネレータは、インターフェイス net.sf.hibernate.id.IdentifierGenerator を実装します。 これはとても単純なインターフェイスなので、 いくつかのアプリケーションでは独自に特別な実装を用意するかもしれません。 しかしHibernateは組み込みの実装をいくつも用意しています。 組み込みのジェネレータにはショートカット名があります：

increment
: long, short, int 型の識別子を生成します。 それらは他のプロセスが同じテーブルにデータを挿入しないときだけユニークです。 クラスタでは使わないでください。

identity
: DB2, MySQL, MS SQL Server, Sybase, HypersonicSQLのアイデンティティ・カラムを サポートします。 返される識別子の型は long, short, int のいずれかです。

sequence
: DB2, PostgreSQL, Oracle, SAP DB, McKoiのシーケンス、 またはInterbaseのジェネレータを使います。 返される識別子の型は long, short, int のいずれかです。

hilo
: long, short, int 型の識別子を効率的に生成するhi/loアルゴリズムを使います。 hi値のソースとして、テーブルとカラムを与えます （デフォルトでは hibernate_unique_key と next_hi それぞれが）。 hi/loアルゴリズムは特定のデータベースだけでユニークな識別子を生成します。 JTAに登録されているコネクションや ユーザが用意したコネクションには、このジェネレータを使わないでください。

seqhilo
: long, short, int 型の識別子を効率的に生成するhi/loアルゴリズムを使います。 名前付きのデータベース・シーケンスを与えます。

uuid.hex
: ネットワークでユニークなstring型の識別子を生成する、 128ビットUUIDアルゴリズムを使います（IPアドレスを使います）。 UUIDは長さ32の16進数の文字列としてエンコードされます。

uuid.string

同じUUIDアルゴリズムを使います。 UUIDはASCII文字からなる長さ16の文字列にエンコードされます。 PostgreSQLでは使わないでください。

native
: データベースにより identity, sequence, hilo のいずれかが選ばれます。

assigned
: save() がコールされる前に、 アプリケーションがオブジェクトに識別子を代入することを可能にします。

foreign
: 他の関連オブジェクトの識別子を使います。 普通は、 <one-to-one> 主キー関連で使います。

5.1.4.2. Hi/Loアルゴリズム

hilo と seqhilo ジェネレータは、 hi/loアルゴリズムの2つの選択可能な実装を提供します。 これは識別子生成の好ましいアプローチです。 1番目の実装は、次回に利用される"hi"値を保持する「特別な」データベース・テーブルを 必要とします。 2番目の実装は、Oracleスタイルのシーケンスを使います（サポートされている場合）。

```
<id name="id" type="long" column="cat_id">
        <generator class="hilo">
                <param name="table">hi_value</param>
                <param name="column">next_value</param>
                <param name="max_lo">100</param>
        </generator>
</id>
```

```
<id name="id" type="long" column="cat_id">
        <generator class="seqhilo">
                <param name="sequence">hi_value</param>
                <param name="max_lo">100</param>
        </generator>
</id>
```

あいにく、独自に Connection を用意するときや、 JTAに登録されているコネクションを取得するために アプリケーション・サーバ・データストアを使うときは、 hilo を使えません。 Hibernateは、新しいトランザクションで"hi"値を取得できなければなりません。 EJB環境での標準の方法は、ステートレス・セッション・ビーンを使い、 hi/loアルゴリズムを実装する方法です。

5.1.4.3. UUIDアルゴリズム

UUIDには以下のものが含まれます： IPアドレス、JVMのスタートアップ・タイム（4分の1秒の正確さ）、 システム時間、（JVMに対してユニークな）カウンタ値。 JavaコードからMACアドレスやメモリ・アドレスを取得することはできないので、 JNIが使えないときには最良の方法です。

uuid.string は、PostgreSQLでは使わないでください。

5.1.4.4. アイデンティティ・カラムとシーケンス

アイデンティティ・カラムをサポートしているデータベース（DB", MySQL, Sybase, MS SQL）では、 identity キー生成を使えます。 シーケンスをサポートするデータベース（DB2, Oracle, PostgreSQL, Interbase, McKoi, SAP DB）では、 sequence スタイルのキー生成を使えます。 これらの戦略は両方とも、新しいオブジェクトを挿入するために、 SQLクエリを2つ必要とします。

```
<id name="id" type="long" column="uid">
        <generator class="sequence">
                <param name="sequence">uid_sequence</param>
        </generator>
</id>
```

```
<id name="id" type="long" column="uid" unsaved-value="0">
        <generator class="identity"/>
</id>
```

クロスプラットフォームの開発では、native 戦略は identity, sequence, hilo 戦略の中から1つを選択しますが、 これはデータベースの能力に依存します。

5.1.4.5. 代入識別子

（Hibernateが生成するものではなく）アプリケーションに識別子を代入させたければ、 assigned ジェネレータを使うことができます。 この特別なジェネレータは、すでにオブジェクトの識別子プロパティに代入された値を 識別子に使います。 この機能をビジネス上の意味として使う場合は十分に注意してください （ほとんどの場合は悪い設計といえます）。

固有の性質として、このジェネレータを使うエンティティは、 SessionのsaveOrUpdate()メソッドでセーブできません。 その代わりに、Sessionの save() または update() メソッドをコールすることで、 オブジェクトをセーブまたは更新すべきかを、 明示的にHibernateに指定します。

5.1.5. composite-id

```
<composite-id
        name="propertyName"
        class="ClassName"
        unsaved-value="any|none"
        access="field|property|ClassName">

        <key-property name="propertyName" type="typename" column="column_name"/>
        <key-many-to-one name="propertyName class="ClassName" column="column_name"/>
        ......
</composite-id>
```

複合キーのあるテーブルに対して、 クラスの複数のプロパティを識別子プロパティとしてマッピングすることができます。 <composite-id> 要素は、 <key-property> プロパティ・マッピングと <key-many-to-one> マッピングを子要素として受け入れます。

```
<composite-id>
        <key-property name="medicareNumber"/>
        <key-property name="dependent"/>
</composite-id>
```

複合識別子の等価性を実装するためには、永続クラスは equals() と hashCode() をオーバーライドしなければなりません。 また Serializable も実装しなければいけません。

あいにく複合識別子のためのこの方法は、 永続オブジェクト自体が自分の識別子であることを意味しています。 オブジェクトそのものを「扱う」以上の便利さはありません。 永続クラス自体のインスタンスをインスタンス化しなければならず、 また複合キーに関連した永続状態を load() できるようになる前に、 識別子プロパティを設定しなければなりません。 別クラスとして複合識別子が実装されている場合の、 より便利な方法について 項7.4. 「複合識別子としてのコンポーネント」 で述べます。 以下で述べる属性は、この代わりの方法だけに当てはまります：

- name（オプション）： 複合識別子を保持するコンポーネント型のプロパティ（次の節を見てください）。
- class （オプション – デフォルトはリフレクションにより決定されるプロパティの型）： 複合識別子として使われるコンポーネントのクラス（次の節を見てください）。
- unsaved-value （オプション – デフォルトは none）： any と設定されると、 一時的インスタンスが新しくインスタンス化されたことを示します。

5.1.6. discriminator

<discriminator> 要素は、 table-per-class-hierarchyマッピング戦略を使うポリモーフィックな永続化に必要で、 テーブルのディスクリミネータ・カラムを定義します。 ディスクリミネータ・カラムは、ある行に対して永続層がどのサブクラスをインスタンス化するかを 伝えるマーカー値を含んでいます。 以下のような型に制限されます：string, character, integer, byte, short, boolean, yes_no, true_false.

```
<discriminator
        column="discriminator_column"  (1)
        type="discriminator_type"      (2)
        force="true|false"             (3)
        insert="true|false"            (4)
/>
```

(1) column（オプション – デフォルトは class）： ディスクリミネータ・カラムの名前。

(2) type （オプション – デフォルトは string）：Hibernate型を示す名前。

(3) force （オプション – デフォルトは false）： ルート・クラスのすべてのインスタンスが復元されたときでも、 Hibernateに許可されたディスクリミネータ・カラムの指定を「強制」します。

(4) insert （オプション – デフォルトは true）： false に設定すると、 ディスクリミネータ・カラムがマッピングされる複合識別子の一部になります。

ディスクリミネータ・カラムの実際の値は、 <class> と <subclass> 要素の discriminator-value 属性で指定されます。

永続クラスへマッピングされない「おまけの」ディスクリミネータ値を持つ行が テーブルにあれば、（そのときに限り）force 属性は有効です。 普通はそういうことはありません。

5.1.7. version（オプション）

<version> 要素はオプションで、 テーブルがバージョン・データを含むことを示します。 これはロング・トランザクション を使うつもりなら、特に役立ちます（以下を見てください）。

```
<version
        column="version_column"                         (1)
        name="propertyName"                             (2)
        type="typename"                                 (3)
        access="field|property|ClassName"               (4)
        unsaved-value="null|negative|undefined"         (5)
/>
```

(1) column（オプション – デフォルトはプロパティ名）： バージョン番号を保持するカラムの名前。

(2) name：永続クラスのプロパティの名前。

(3) type （オプション – デフォルトは integer）：バージョン番号の型。

(4) access （オプション – デフォルトは property）： プロパティの値へのアクセスにHibernateが使う戦略。

(5) unsaved-value （オプション – デフォルトは undefined）： インスタンスが新しくインスタンス化されたことを示す （セーブされていないことを示す）バージョン・プロパティの値。 以前のSessionでセーブまたはロードされていない一時的なインスタンスと区別するために 使います。 （undefined は識別子プロパティの値が使われることを指定します。）

バージョン番号は long, integer, short, timestamp, calendar 型のいずれかです。

5.1.8. timestamp（オプション）

オプションの <timestamp> 要素は、 テーブルがタイムスタンプ・データを含むことを示します。 これはバージョン付けの代わりの方法として用意されています。 タイムスタンプはもともと楽観的ロックの安全性の低い実装です。 しかしアプリケーションは異なる用途で使うかもしれません。

```
<timestamp
        column="timestamp_column"           (1)
        name="propertyName"                 (2)
        access="field|property|ClassName"   (3)
        unsaved-value="null|undefined"      (4)
/>
```

(1) column（オプション – デフォルトはプロパティ名）： タイムスタンプを保持するカラムの名前。

(2) name： 永続クラスのJava型 Date または Timestamp のJavaBeansスタイル・プロパティの名前。

(3) access （オプション – デフォルトは property）： プロパティの値へのアクセスにHibernateが使う戦略。

(4) unsaved-value （オプション – デフォルトは null）： インスタンスが新しくインスタンス化された （セーブされていない）ことを示すバージョン・プロパティの値。 以前のSessionでセーブまたはロードされた一時的なインスタンスと 区別するために使われます。 （undefined と指定すると、 識別子プロパティの値が使われます。）

<timestamp> が <version type="timestamp"> と等価であることに注意してください。

5.1.9. property

<property> 要素は、クラスのJavaBeanスタイルの永続プロパティを定義します。

```
<property
        name="propertyName"                 (1)
        column="column_name"                (2)
        type="typename"                     (3)
        update="true|false"                 (4)
        insert="true|false"                 (4)
        formula="arbitrary SQL expression"  (5)
        access="field|property|ClassName"   (6)
/>
```

(1) name：小文字で始まるプロパティ名。

(2) column（オプション – デフォルトはプロパティ名）： マッピングされたデータベース・テーブル・カラムの名前。

(3) type（オプション）：Hibernate型を示す名前。

(4) update, insert （オプション – デフォルトは true）：マッピングされたカラムがSQL UPDATE

や INSERT に含まれることを指定します。 両方とも false に設定すると、 同じカラムにマッピングされた他のプロパティやトリガや 他のアプリケーションによって初期化された純粋な「導出」プロパティが可能になります。

(5) formula（オプション）： 計算 プロパティのための値を定義するSQL式。 計算プロパティには自分のカラム・マッピングがありません。

(6) access（オプション – デフォルトは property）： プロパティの値へのアクセスにHibernateが使う戦略。

typename には以下のようなものが可能です：

1. Hibernateの基本型の名前（例 integer, string, character, date, timestamp, float, binary, serializable, object, blob）。
2. デフォルトの基本型のJavaクラス名 （例 int, float, char, java.lang.String, java.util.Date, java.lang.Integer, java.sql.Clob）。
3. PersistentEnum のサブクラスの名前（例 eg.Color）。
4. シリアライズ可能なJavaクラスの名前。
5. カスタム型のクラス名（例 com.illflow.type.MyCustomType）。

型を指定しなければ、Hibernateは正しいHibernate型を推測するために、 名前付きのプロパティに対してリフレクションを使います。 Hibernateはルール2, 3, 4をその順序に使い、 getterプロパティの返り値のクラスの名前を解釈しようとします。 しかしこれは常に十分とは限りません。 type 属性が必要な場合があります。 （例えば Hibernate.DATE と Hibernate.TIMESTAMP を区別するとき、 またはカスタム型を指定するとき。）

access 属性で、 実行時にHibernateがどのようにプロパティにアクセスするかを制御できます。 デフォルトではHibernateはプロパティのget/setのペアをコールします。 access="field" と指定すれば、 Hibernateはリフレクションを使いget/setのペアを介さずに、直接フィールドにアクセスします。 インターフェイス net.sf.hibernate.property.PropertyAccessor を 実装するクラスを指定することで、プロパティへのアクセスに独自の戦略を指定することができます。

5.1.10. many-to-one

他の永続クラスへの普通の関連は many-to-one 要素を使い定義します。 リレーショナル・モデルはmany-to-one関連です。 （本当は単なるオブジェクト参照です。）

```
<many-to-one
        name="propertyName"                                 (1)
        column="column_name"                                (2)
        class="ClassName"                                   (3)
        cascade="all|none|save-update|delete"               (4)
        outer-join="true|false|auto"                        (5)
        update="true|false"                                 (6)
        insert="true|false"                                 (6)
        property-ref="propertyNameFromAssociatedClass"      (7)
        access="field|property|ClassName"                   (8)
                unique="true|false"                                     (9)
/>
```

(1) name：プロパティ名。

(2) column（オプション）：カラム名。

(3) class（オプション – デフォルトは、 リフレクションにより決定されるプロパティの型）：関連クラスの名前。

(4) cascade（オプション）： 親オブジェクトから関連オブジェクトへ、どの操作をカスケードさせるかを指定します。

(5) outer-join（オプション - デフォルトは auto）： hibernate.use_outer_join が設定されたとき、 この関連へのアウター･ジョインによるフェッチを可能にします。

(6) update, insert（オプション - デフォルトは true）： マッピングされたカラムがSQL UPDATE や INSERT 文に含まれることを指定します。 両方とも false に設定すると、 同じカラムをマッピングされた他のプロパティやトリガや 他のアプリケーションによって初期化された純粋な「導出」プロパティが可能になります。

(7) property-ref（オプション）： この外部キーに結合された関連クラスのプロパティ名。 指定されなければ、関連クラスの主キーが使われます。

(8) access（オプション - デフォルトは property）： プロパティの値へのアクセスにHibernateが使う戦略。

(9) unique（オプション）： DDLの生成において、外部キーカラムに対するユニーク制約を可能にします。

cascade 属性は以下の値を受け付けます： all, save-update, delete, none。 none 以外の値を設定すると、ある操作が関連（子）オブジェクトへ伝播されます。 以下の「ライフサイクル・オブジェクト」を見てください。

outer-join 属性は、異なる3つの値を受け付けます：

- auto（デフォルト）：関連クラスのプロキシがなければ、 アウター･ジョインで関連をフェッチします。
- true 常にアウター･ジョインで関連をフェッチします。
- false アウター･ジョインでは関連をフェッチしません。

典型的な many-to-one の定義は、以下のように単純です。

```
<many-to-one name="product" class="Product" column="PRODUCT_ID"/>
```

property-ref 属性はレガシー・データのマッピングだけに使われます。 そのレガシー・データでは、外部キーは主キーではない関連テーブルのユニーク・キーを参照します。 これは醜いリレーショナル・モデルです。 例えば Product クラスが、 主キーでないユニークなシリアル・ナンバーを持っていると仮定してみてください。 （unique 属性はSchemaExportツールを使ったHibernateのDDL生成を制御します。）

```
<property name="serialNumber" unique="true" type="string" column="SERIAL_NUMBER"/>
```

以下のように OrderItem のマッピングを使えます：

```
<many-to-one name="product" property-ref="serialNumber" column="PRODUCT_SERIAL_NUMBER"/>
```

しかしこれは決しておすすめできません。

5.1.11. one-to-one

他の永続クラスへのone-to-one関連は、one-to-one 要素で定義します。

```
<one-to-one
        name="propertyName"                                 (1)
        class="ClassName"                                   (2)
        cascade="all|none|save-update|delete"               (3)
        constrained="true|false"                            (4)
        outer-join="true|false|auto"                        (5)
        property-ref="propertyNameFromAssociatedClass"      (6)
        access="field|property|ClassName"                   (7)
```

```
/>
```

(1) name：プロパティ名。
(2) class（オプション - デフォルトはリフレクションにより決定されるプロパティの型）： 関連クラスの名前。
(3) cascade（オプション）： 親オブジェクトから関連オブジェクトへ、どの操作をカスケードするかを指定します。
(4) constrained（オプション）： マッピングされたテーブルの主キーに対する外部キー制約が、関連クラスのテーブルを参照することを指定します。 このオプションは save() と delete() がカスケードされる順序に影響します （そしてスキーマ・エクスポート・ツールにも使われます）。
(5) outer-join（オプション - デフォルトは auto）： hibernate.use_outer_join が設定されたとき、 この関連に対するアウター･ジョインによるフェッチを有効にします。
(6) property-ref （オプション）：このクラスの主キーに結合された、関連クラスのプロパティ名。 指定されなければ、関連クラスの主キーが使われます。
(7) access（オプション - デフォルトは property）： プロパティの値へのアクセスにHibernateが使う戦略。

one-to-one関連には2種類あります：

・ 主キー関連

・ ユニーク外部キー関連

主キー関連には、余分なテーブル・カラムは必要ありません。 もし2つの行が関連により関係していれば、2つのテーブルは同じ主キーの値を共有します。 そのため2つのオブジェクトを1つの主キーに関係付けたければ、 同じ識別子の値を代入することを確実にしなければいけません。

主キー関連には、以下のマッピングを Employee と Person にそれぞれ追加してください。

```
<one-to-one name="person" class="Person"/>
```

```
<one-to-one name="employee" class="Employee" constrained="true"/>
```

今PERSONとEMPLOYEEの関係する行の主キーが同じであることを確実にしなければいけません。 私たちは foreign という、特別なHibernate識別子生成戦略を使います：

```
<class name="person" table="PERSON">
    <id name="id" column="PERSON_ID">
        <generator class="foreign">
            <param name="property">employee</param>
        </generator>
    </id>
    ...
    <one-to-one name="employee"
        class="Employee"
        constrained="true"/>
</class>
```

Employee インスタンスが、Person の employee プロパティで参照されるように、 新しくセーブされた Person のインスタンスには同じ主キーの値が代入されます。

もう1つの方法として、Employee から Person への ユニーク制約を使った外部キーは以下のように

表現されます：

```
<many-to-one name="person" class="Person" column="PERSON_ID" unique="true"/>
```

そしてこの関連は、以下の記述を Person のマッピングに追加すると双方向にできます：

```
<one-to-one name"employee" class="Employee" property-ref="person"/>
```

5.1.12. component, dynamic-component

<component> 要素は、 子オブジェクトのプロパティを親クラスのテーブルのカラムへマッピングします。 コンポーネントは順番に、自分のプロパティ、コンポーネント、コレクションを定義できます。 以下の「コンポーネント」を見てください。

```
<component
        name="propertyName"                 (1)
        class="className"                   (2)
        insert="true|false"                 (3)
        upate="true|false"                  (4)
        access="field|property|ClassName">  (5)

        <property ...../>
        <many-to-one .... />
        ........
</component>
```

(1) name：プロパティ名。

(2) class（オプション - デフォルトはリフレクションにより決定されるプロパティの型）： コンポーネント（子）クラスの名前。

(3) insert：マッピングされたカラムをSQL INSERT に現れるようにするか？

(4) update: マッピングされたカラムがSQL UPDATE に現れるようにするか？

(5) access （オプション - デフォルトは property）： プロパティの値へのアクセスにHibernateが使う戦略。

子の <property> タグで、子のクラスのプロパティをテーブル・カラムにマッピングします。

<component> 要素は、親エンティティへ戻る参照として、 コンポーネントのクラスのプロパティをマッピングする <parent> サブ要素を受け入れます。

<dynamic-component> 要素は、 1つの Map がコンポーネントとしてマッピングされることを可能にします。 そこではプロパティ名はmapのキーを参照します。

5.1.13. subclass

最後にポリモーフィックな永続化には、ルートの永続クラスの各サブクラスの定義が必要です。 （おすすめの）table-per-class-hierarchyマッピング戦略では、 <subclass> 定義が使われます。

```
<subclass
        name="ClassName"                                (1)
        discriminator-value="discriminator_value"       (2)
        proxy="ProxyInterface"                          (3)
        lazy="true|false"                               (4)
        dynamic-update="true|false"
        dynamic-insert="true|false">

        <property .... />
        .....
```

```
</subclass>
```

(1) name：サブクラスの完全修飾されたクラス名。
(2) discriminator-value（オプション - デフォルトはクラス名）： 個々のサブクラスを区別するための値。
(3) proxy（オプション）：lazy初期化プロキシに使うクラスやインターフェイスを指定します。
(4) lazy（オプション）： lazy="true" と設定することは、 #### インターフェイスとしてクラス名を指定することと同じです。

各サブクラスでは、永続プロパティとサブクラスを定義します。 <version> と <id> プロパティは、 ルートクラスから継承されると仮定されます。 階層構造におけるサブクラスは、ユニークな discriminator-value を定義しなければいけません。 noneが指定されると完全修飾されたJavaクラス名が使われます。

5.1.14. joined-subclass

もう1つの方法として、自分のテーブルに永続化されるサブクラスは、 <joined-subclass> 要素で定義します。 （table-per-subclassマッピング戦略）。

```
<joined-subclass
        name="ClassName"                    (1)
        proxy="ProxyInterface"              (2)
        lazy="true|false"                   (3)
        dynamic-update="true|false"
        dynamic-insert="true|false">

        <key .... >

        <property .... />
        .....
</joined-subclass>
```

(1) name：サブクラスの完全修飾されたクラス名。
(2) proxy（オプション）：lazy初期化プロキシに使うクラスやインターフェイスを指定します。
(3) lazy（オプション）：lazy="true" と設定することは、 #### インターフェイスとしてクラス名を指定することと同じです。

このマッピング戦略には、ディスクリミネータ・カラムは必要ありません。 しかし各サブクラスは <key> 要素を使い、 オブジェクト識別子を保持するテーブル・カラムを定義しなければいけません。 この章の初めのマッピングは以下のように書き直せます：

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping package="eg">

        <class name="Cat" table="CATS">
                <id name="id" column="uid" type="long">
                        <generator class="hilo"/>
                </id>
                <property name="birthdate" type="date"/>
                <property name="color" not-null="true"/>
                <property name="sex" not-null="true"/>
                <property name="weight"/>
                <many-to-one name="mate"/>
                <set name="kittens">
```

```
                        <key column="MOTHER"/>
                        <one-to-many class="Cat"/>
                </set>
                <joined-subclass name="DomesticCat" table="DOMESTIC_CATS">
                        <key column="CAT"/>
                        <property name="name" type="string"/>
                </joined-subclass>
        </class>

        <class name="eg.Dog">
                <!-- ###Dog############ -->
        </class>

</hibernate-mapping>
```

5.1.15. map, set, list, bag

コレクションについては後で述べます。

5.1.16. import

アプリケーションに同じ名前の2つの永続クラスがあるとします。 そしてHibernateクエリで完全修飾された（パッケージの）名前を指定したくないとします。 そのような場合は `auto-import="true"` に頼らず、 クラスが「インポート」されたものであると明示することができます。 明示的にマッピングされていないクラスやインターフェイスでさえもインポートできます。

```
<import class="java.lang.Object" rename="Universe"/>
```

```
<import
        class="ClassName"               (1)
        rename="ShortName"              (2)
/>
```

(1) `class`：Javaクラスの完全修飾されたクラス名。

(2) `rename`（オプション - デフォルトは修飾されていないクラス名）： クエリ言語で使われる名前。

5.2. Hibernate型

5.2.1. エンティティとバリュー

永続サービスに関わるいろいろなJava言語レベルのオブジェクトの振る舞いを理解するためには、それらを2つのグループに分ける必要があります：

エンティティ はエンティティへの参照を保持する、 他のすべてのオブジェクトから独立して存在します。 参照されないオブジェクトはガベージ・コレクトされてしまう通常のJavaモデルと、 これを比べてみてください。 （親エンティティから子へ、セーブと削除が カスケード されうることを除いて） エンティティは明示的にセーブまたは削除されなければいけません。 これは到達可能性によるオブジェクト永続化のODMGモデルとは異なっています。 大規模なシステムでアプリケーション・オブジェクトが普通どのように使われるかにより緊密に対応します。 エンティティは循環と参照の共有をサポートします。 またそれらはバージョン付けすることもできます。

エンティティの永続状態は他のエンティティや バリュー 型のインスタンスへの参照から構成されます。 バリューはプリミティブ、コレクション、コンポーネント、更新不能オブジェクトです。エンティティとは違い、（特定のコレクションとコンポーネントにおいて）バリューは、 到達可能性による永続化や削除が 行われます 。 バリュー・オブジェクト（とプリミティブ）は、含んでいるエンティティと一緒に永続化や削除が行われるので、 それらを独立にバージョン付けすることはできません。 バリューには独立したアイデンティティがないので、複数のエンティティやコレクションがこれを共有することはできません。

コレクションを除くすべてのHibernate型が、nullをサポートします。

これまで「永続クラス」という言葉をエンティティの意味で使ってきました。 これからもそうしていきます。 厳密に言うと、永続状態を持つユーザ定義のクラスのすべてが エンティティというわけではありません。 コンポーネント はバリューの意味を持つユーザ定義クラスです。

5.2.2. 基本のバリュー型

基本型 は大まかに以下のように分けられます。

`integer, long, short, float, double, character, byte, boolean, yes_no, true_false`
: Javaプリミティブやラッパークラスから適切な（ベンダー固有の）SQLカラム型への型マッピング。 `boolean, yes_no` と `true_false` は、 すべてJavaの `boolean` または `java.lang.Boolean` の代替エンコードです。

`string`
: `java.lang.String` から `VARCHAR` （またはOracleの `VARCHAR2`）への型マッピング。

`date, time, timestamp`
: `java.util.Date` とそのサブクラスからSQL型 `DATE, TIME, TIMESTAMP` （またはそれらと等価なもの）への型マッピング。

`calendar, calendar_date`
: `java.util.Calendar` からSQL型 `TIMESTAMP, DATE` （またはそれらと等価なもの）への型マッピング。

`big_decimal`
: `java.math.BigDecimal` から `NUMERIC` （またはOracleの `NUMBER`）への型マッピング。

`locale, timezone, currency`
: `java.util.Locale, java.util.TimeZone, java.util.Currency` から `VARCHAR` （またはOracleの `VARCHAR2`）への型マッピング。 `Locale` と `Currency` のインスタンスは、 それらのISOコードにマッピングされます。 `TimeZone` のインスタンスは、それらの `ID` にマッピングされます。

`class`
: `java.lang.Class` から `VARCHAR` （またはOracleの `VARCHAR2`）への型マッピング。 `Class` はその完全修飾された名前にマッピングされます。

`binary`
: バイト配列は、適切なSQLバイナリ型にマッピングされます。

`text`

長いJava文字列は、SQL CLOB または TEXT 型にマッピングされます。

serializable
: シリアライズ可能なJava型は、適切なSQLバイナリ型にマッピングされます。 デフォルトで基本型や PersistentEnum を実装しないシリアライズ可能なJavaクラスや インターフェイスの名前を指定することで、 Hibernate型を serializable とすることもできます。

clob, blob
: JDBCクラス java.sql.Clob と java.sql.Blob に対する型マッピング。 blobやclobオブジェクトはトランザクションの外では再利用できないため、 アプリケーションによっては不便かもしれません。 （さらにはドライバ・サポートがパッチで首尾一貫していません。）

エンティティとコレクションのユニークな識別子は、binary, blob, clob を除く、どんな基本型でも構いません。 （複合識別子でも構いません。以下を見てください。）

基本のバリュー型にはそれぞれ、net.sf.hibernate.Hibernate で定義された Type 定数があります。 例えば、Hibernate.STRING は string 型を表現しています。

5.2.3. 永続列挙型

列挙 型は、クラスが更新不能インスタンスの（小さな）定数を持つという、 Javaに共通のイディオムです。 net.sf.hibernate.PersistentEnum を実装し、 操作 toInt() と fromInt() を定義することで、 永続列挙型を生成できます：

```
package eg;
import net.sf.hibernate.PersistentEnum;

public class Color implements PersistentEnum {
    private final int code;
    private Color(int code) {
        this.code = code;
    }
    public static final Color TABBY = new Color(0);
    public static final Color GINGER = new Color(1);
    public static final Color BLACK = new Color(2);

    public int toInt() { return code; }

    public static Color fromInt(int code) {
        switch (code) {
            case 0: return TABBY;
            case 1: return GINGER;
            case 2: return BLACK;
            default: throw new RuntimeException("Unknown color code");
        }
    }
}
```

Hibernate型の名前は、単に列挙クラスの名前です。 この例では eg.Color です。

5.2.4. カスタム・バリュー型

開発者が独自のバリュー型を作成することは、比較的簡単です。 例えば、java.lang.BigInteger 型のプロパティから VARCHAR カラムへ永続化したいかもしれません。 Hibernateはこのための組み込み型を用意していません。 しかしカスタム型を使えば、プロパティ（またはコレクションの要素）を1つのテーブル・カラムにマッピングできます。 例えば、java.lang.String 型の getName() / setName() プロパティを FIRST_NAME, INITIAL, SURNAME カラムにマッピングできます。

カスタム型を実装するには、net.sf.hibernate.UserType または net.sf.hibernate.CompositeUserType を実装し、 型の完全修飾された名前を使ってプロパティを定義します。 net.sf.hibernate.test.DoubleStringType を確認してください。

```
<property name="twoStrings" type="net.sf.hibernate.test.DoubleStringType">
    <column name="first_string"/>
    <column name="second_string"/>
</property>
```

<column> タグで、プロパティを複数のカラムへマッピングできることに注目してください。

Hibernateには多様な組み込み型とコンポーネントのサポートがあるので、 カスタム型を使う必要はめったに ありません。 しかしそれでも、アプリケーション内で頻繁に発生する（エンティティでない） クラスのためにカスタム型を使うのは良いことだと考えられます。 例えば MonetoryAmount（お金）クラスは、 コンポーネントとしてマッピングすることも簡単ですが、 CompositeUserType を使うと良いと考えられます。 その動機の1つは抽象化です。 カスタム型を使えば、お金の値をどのように表現しようとも、 マッピング・ドキュメントは起こりうる変化に対して影響を抑えることができます。

5.2.5. Any型のマッピング

プロパティ・マッピングはさらにもう1つあります。 <any> マッピング要素は、 複数のテーブルからクラスへのポリモーフィックな関連を定義します。 この型のマッピングには必ず複数のカラムが必要です。1番目のカラムは関連エンティティの型を保持します。 残りのカラムは識別子を保持します。この種類の関連には外部キー制約を指定することはできません。 そのためこれは最も使われることのない（ポリモーフィックな）関連のマッピング方法です。 非常に特別な場合に（例えば、検査ログやユーザ・セッション・データなど）これを使うべきです。

```
<any name="anyEntity" id-type="long" meta-type="eg.custom.Class2TablenameType">
    <column name="table_name"/>
    <column name="id"/>
</any>
```

meta-type 属性は、アプリケーションがあるカスタム型を指定するために使います。 そのカスタム型はデータベース・カラムの値を id-type で指定された型の識別子プロパティを持つ永続クラスへマッピングするものです。 meta-typeが java.lang.Class のインスタンスを返すなら、他には何も必要ありません。 そうではなく string や character のような基本型なら、 値からクラスへのマッピングを指定しなければいけません。

```
<any name="anyEntity" id-type="long" meta-type="string">
    <meta-value value="TBL_ANIMAL" class="Animal"/>
    <meta-value value="TBL_HUMAN" class="Human"/>
    <meta-value value="TBL_ALIEN" class="Alien"/>
    <column name="table_name"/>
    <column name="id"/>
</any>
```

```
<any
        name="propertyName"                     (1)
        id-type="idtypename"                    (2)
        meta-type="metatypename"                (3)
        cascade="none|all|save-update"          (4)
        access="field|property|ClassName"       (5)
>
        <meta-value ... />
        <meta-value ... />
```

```
        .....
        <column .... />
        <column .... />
        .....
</any>
```

(1) `name`：プロパティ名。

(2) `id-type`：識別子の型。

(3) `meta-type`（オプション - デフォルトは `class`）： `java.lang.Class` を1つのデータベース・カラム、 またはディスクリミネータ・マッピングで許された型にマッピングする型。

(4) `cascade`（オプション - デフォルトは `none`）： カスケードのスタイル。

(5) `access`（オプション - デフォルトは `property`）： プロパティの値へのアクセスにHibernateが使う戦略。

Hibernate1.2で同じような役割を果たしていた、古い `object` 型はまだサポートされてはいますが、 現在は準非推奨になっています。

5.3. SQL引用識別子

マッピング・ドキュメントでテーブルやカラムの名前をバック・クォートで囲むことで、Hibernateに生成されるSQL中の識別子を引用させることができます。 HibernateはSQL `Dialect` に対応する、正しい引用スタイルを使います （普通はダブル・クォートですが、SQL Serverではかぎ括弧、MySQLではバック・クォートです）。

```
<class name="LineItem" table="`Line Item`">
    <id name="id" column="`Item Id`"/><generator class="assigned"/></id>
    <property name="itemNumber" column="`Item #`"/>
    ...
</class>
```

5.4. モジュール式マッピング・ファイル

分割されたマッピング・ドキュメントで、`hibernate-mapping` の直下に `subclass` や `joined-subclass` を指定できます。 これにより単に新しいマッピング・ファイルを追加して、クラスの階層構造を拡張することができます。 そのためにはサブクラスのマッピングの `extends` 属性で、 以前にマッピングされたスーパークラスの名前を指定しなければいけません。 この機能を使うとマッピング・ドキュメントの並び順が重要になります。

```
<hibernate-mapping>
        <subclass name="eg.subclass.DomesticCat" extends="eg.Cat" discriminator-value="D">
             <property name="name" type="string"/>
        </subclass>
</hibernate-mapping>
```

第6章 コレクションのマッピング

6.1. 永続性コレクション

この節はJavaのコード例をあまり含んでいません。Javaのコレクション・フレームワークの使用法は既に知っているものとします。もし既に使用法を知っているなら、それ以上知っておく必要はありません。いつもの方法でコレクションを使用してください。

Hibernate はjava.util.Map, java.util.Set, java.util.SortedMap, java.util.SortedSet, java.util.List,そしてどんな永続性エンティティや値の配列のインスタンスも永続化することができます。 java.util.Collection や java.util.List型のプロパティを"bag"として永続化することもできます。

警告:コレクションの永続化は、コレクション・インターフェースを実装するクラスによって加えられた余分な動作(例えばLinkedHashSetの反復順序)は保持しません。コレクションの永続化は実際にそれぞれHashMap, HashSet, TreeMap, TreeSet および ArrayListの様に振舞います。更に言えば、コレクションを保持する属性のJavaの型はインタフェース型でなくてはなりません(つまりHashMap, TreeSet , ArrayListではなく、Map, Set や Listでなくてはならない)。Hibernateは知らない間にMap, Set , Listのインスタンスを Map, Set , Listの永続化機能を実装したインスタンスと取り替えるのでこの制限が存在します(従ってコレクション上で==を使用する場合注意してください)。

```
Cat cat <literal>==</literal> new DomesticCat();
Cat kitten = new DomesticCat();
....
Set kittens = new HashSet();
kittens.add(kitten);
cat.setKittens(kittens);
session.save(cat);
kittens = cat.getKittens(); //Okay, kittens collection is a Set
(HashSet) cat.getKittens(); //Error!
```

コレクションはバリュータイプの通常の規則に従います。つまり、含まれているエンティティと一緒に生成、削除された参照は共有されません。リレーショナル・モデルをベースにしているためnull値のセマンティクスをサポートしません。つまりHibernateは参照先の無いコレクションと空のコレクションを見分けません。

コレクションは永続性オブジェクトによって参照されたときに自動的に永続化され、永続性オブジェクトによって参照されなくなったときに自動的に削除されます。もしコレクションがある永続性オブジェクトから他の永続性オブジェクトに渡されたなら、その要素はあるテーブルから別のテーブルへ移動されます。このことについてあまり心配する必要はありません。普通のJavaのコレクションを使うのと同じ方法でHibernateのコレクションを使ってください。しかしHibernateのコレクションを使用する前に、後述する双方向関連のセマンティクスをきちんと理解してください。

コレクションのインスタンスは、データベース内で所有するエンティティの外部キーによって識別されます。 この外部キーはコレクション・キーと呼ばれます。コレクション・キーは<key>要素によってマッピングされます。

コレクションは基本型、カスタム型、エンティティ型、コンポーネントだけでなく、ほとんどのHibernate型を格納することができます。コレクション内のオブジェクトは"値渡し"セマンティクスで扱われるか(そのためコレクションの所有者に完全に依存します)、または自身のライフサイクルを使って他のエンティティへの参照でありえます。このことは重要な定義です。また、コレクショ

ンは他のコレクションを格納できません。格納される型はコレクション要素型とよばれます。コレクションの要素は、<element>, <composite-element>, <one-to-many>, <many-to-many> あるいは <many-to-any>によってマッピングされます。最初の2つは値のセマンティクスで要素をマッピングし、あとの3つはエンティティの関連をマッピングするために使用されます。

Setとbagを除くすべてのコレクション型は、配列またはListのインデックスやMapのキーへマッピングするインデックス ・カラムを持ちます。Mapのインデックスは任意の基本型、エンティティ型あるいは合成型などかもしれません(コレクションではありえません)。配列やlistのインデックスは常に##型です。インデックスは<index>, <index-many-to-many>, <composite-index> あるいは <index-many-to-any> を使用してマッピングされます。

Hibernateはコレクションに対して、多くのリレーショナル・モデルをカバーする幅広いマッピングを提供しています。どのようにして様々なマッピング宣言をデータベースへ変換するか感じをつかむため、スキーマ生成ツールを使ってみることを提案します。

6.2. コレクションのマッピング

コレクションは<set>, <list>, <map>, <bag>, <array> そして <primitive-array> 要素によって宣言されます。以下に<map>で例を示します。:

```
<map
    name="propertyName"                                          (1)
    table="table_name"                                           (2)
    schema="schema_name"                                         (3)
    lazy="true|false"                                            (4)
    inverse="true|false"                                         (5)
    cascade="all|none|save-update|delete|all-delete-orphan"      (6)
    sort="unsorted|natural|comparatorClass"                      (7)
    order-by="column_name asc|desc"                              (8)
    where="arbitrary sql where condition"                        (9)
    outer-join="true|false|auto"                                 (10)
    batch-size="N"                                               (11)
    access="field|property|ClassName"                            (12)
>

    <key .... />
    <index .... />
    <element .... />
</map>
```

(1) name コレクションのプロパティ名

(2) table（オプション - デフォルトはプロパティ名）コレクションテーブルの名前(one-to-many関連では使用しません)

(3) schema（オプション）ルート要素のスキーマ宣言を書き換える、テーブルスキーマの名前

(4) lazy（オプション - デフォルトはfalse）lazy初期化を可能にします。(配列には使用しません)

(5) inverse（オプション - デフォルトはfalse）双方向関連の「逆」側としてこのコレクションにマークします。

(6) cascade（オプション - デフォルトはnone）子エンティティへのカスケード操作を可能にします

(7) sort（オプション）##ソート順序または与えられたコンパレータ・クラスを使ってソート・コレクションを指定します

(8) order-by（オプション, JDK1.4のみ）オプションのasc , descを一緒に使って、Map, Set,bagの反復順序を決めるテーブル・カラム(カラム)を指定します

(9) (optional)（オプション）コレクションを検索や削除するときに使う任意のSQLのWHERE条件を指定します。(コレクションが利用可能なデータの部分集合だけを含んでいるとき有用です。)
(10)（オプション）可能な場合は常にouter-joinでコレクションをフェッチすることを指定します。SQLのSELECT一回に一つだけのコレクションを返すかもしれません
(11) batch-size（オプション，デフォルトは1）lazyにこのコレクションのインスタンスを取って来るための「バッチ・サイズ」を指定します。
(12) access（オプション - デフォルトはproperty)：プロパティの値へのアクセスのためにHibernateが使用するべき戦略

Listや配列のマッピングは、インデックス(foo[i]のi)を保持するテーブル･カラムを分離することを要求します。リレーショナル・モデルがインデックス・カラムを持っていない場合、例えばレガシー・データを用いて仕事をしている場合、順序の無いSetを代わりに使用します。このことは、Listは順序の無いコレクションにアクセスするより便利であると考える人達からは、かけ離れた意見と思います。HibernateのコレクションはSet，List および Mapインターフェースに付けられた実際のセマンティクスに厳密に従います。Listは要素を自然と再整列などしません!

また一方でbagの動作をエミュレートするためにListを使用することを考えた人々は、まさに不満を持っています。bagは同じ要素を複数回含んでいるかもしれない、順序が無く、インデックスがないコレクションです。Javaのコレクションフ・レームワークにはBagインターフェースがないのでListを使ってエミュレートしなくてはなりません。HibernateはListやCollection型のプロパティと<bag>の要素をマッピングさせます。bagは実際にはCollectionとではなく、List規約のセマンティクスと衝 突するものであることに注意してください(しかしこの章の後で議論するようにbagを適宜ソートできます)。

注意:inverse="false"でマッピングされた大きなHibernateのbagは非効率的であり、避けるべきです。それはbagは個々の列を識別するために使用できるキーがないので、Hibernateは列を個々に作成し、削除し、更新することができないからです。

6.3. 値のコレクションとmany-to-many関連

コレクション･テーブルは、任意の値コレクションと任意のmany-to-many関連としてマッピングされた他のエンティティを参照するコレクションに対して必要となります。テーブルはキー･カラム(外部キー)、要素カラム、そしておそらくインデックス・カラムを必要とします。

コレクションテーブルから所有するクラスのテーブルへの外部キー<key>要素を使用して宣言されます。

```
<key column="column_name"/>
```

(1) column（必須）：外部キーのカラム名

mapやlistのような索引付けされたコレクションについては<index>要素が必要です。listは、このインデックス･カラムが、番号が0から付けられた連続する整数を含んでいます。もしレガシー･データを扱わなければならないなら、インデックスが間違いなく0から始まるようにしてください。mapについてはカラムがHibernate型のどんな値も含むことができます。

```
<index
        column="column_name"                (1)
        type="typename"                     (2)
/>
```

(1) column（必須）: コレクション・インデックスの値を保持するカラムの名前
(2) type（オプション，デフォルトはInteger）: コレクション・インデックスの型

もうひとつの方法として、mapはエンティティ型のオブジェクトによって索引付けされるかもしれません。我々は<index-many-to-many> 要素を使用します

```
<index-many-to-many
        column="column_name"                (1)
        class="ClassName"                   (2)
/>
```

(1) column（必須）: コレクション・インデックスの値を保持する外部キーカラムの名前
(2) class（必須）: コレクション・インデックスとして使用されるエンティティ・クラス

値のコレクションに関しては<element>タグを使用します。

```
<element
        column="column_name"                (1)
        type="typename"                     (2)
/>
```

(1) column（必須）: コレクション・インデックスの値を保持するカラムの名前
(2) type（必須）: コレクション要素の型

コレクション自身のテーブルを備えたエンティティのコレクションは、many-to-many関連についてのリレーショナルの概念に相当します。many-to-many関連はJavaのコレクションの最も自然なマッピングではありますが、通常最良のリレーショナル・モデルではありません。

```
<many-to-many
        column="column_name"                                (1)
        class="ClassName"                                   (2)
        outer-join="true|false|auto"                        (3)
    />
```

(1) column（必須）: 外部キーカラムの要素の名前
(2) class（required）:（必須）: 関連クラスの名前
(3) outer-join（オプション – デフォルトはauto）: hibernate.use_outer_joinが設定されたとき関連をアウター・ジョインでフェッチ可能にする

まずStringの集合の例:

```
<set name="names" table="NAMES">
    <key column="GROUPID"/>
    <element column="NAME" type="string"/>
</set>
```

整数を含んでいるbag:(order-by属性によって反復順序を定義されたbag):

```
<bag name="sizes" table="SIZES" order-by="SIZE ASC">
    <key column="OWNER"/>
    <element column="SIZE" type="integer"/>
</bag>
```

エンティティの配列–この場合many-to-many関連(ここでエンティティはライフサイクル・オブジェクトであり、cascade="all"であることに注意):

```
<array name="foos" table="BAR_FOOS" cascade="all">
    <key column="BAR_ID"/>
```

```
    <index column="I"/>
    <many-to-many column="FOO_ID" class="org.hibernate.Foo"/>
</array>
```

Stringインデックスから日付へのマップッピング:

```
<map name="holidays" table="holidays" schema="dbo" order-by="hol_name asc">
    <key column="id"/>
    <index column="hol_name" type="string"/>
    <element column="hol_date" type="date"/>
</map>
```

コンポーネントのlist(次の章で議論します):

```
<list name="carComponents" table="car_components">
    <key column="car_id"/>
    <index column="posn"/>
    <composite-element class="org.hibernate.car.CarComponent">
            <property name="price" type="float"/>
            <property name="type" type="org.hibernate.car.ComponentType"/>
            <property name="serialNumber" column="serial_no" type="string"/>
    </composite-element>
</list>
```

6.4. one-to-many関連

one-to-many関連はコレクション・テーブルを介在せずに、2つのクラスのテーブルを直接リンクします。(これはone-to-manyリレーショナル・モデルを実装します。)このリレーショナル・モデルは以下のように、Javaのコレクションのいくつかのセマンティクスを失います:

- map、set、listにnull値は含めません。
- コレクションに含まれたエンティティ・クラスのインスタンスは、2つ以上のコレクションのインデックスに属せません。
- コレクションに含まれたエンティティ・クラスのインスタンスは、2つ以上のコレクション・インデックスの値には現れません。

FooからBarへの関連はキーカラムと、もしあるならばインデックス・カラムをコレクションに含まれたエンティティ・クラスのテーブル、つまりBarへ追加することを要求します。これらのカラムは上記のように<key>と<index>要素を使ってマッピングします。

<one-to-many>タグは、one-to-many関連を示します

```
<one-to-many class="ClassName"/>
```

(1) class (必須): 関連クラスの名前

例:

```
<set name="bars">
    <key column="foo_id"/>
    <one-to-many class="org.hibernate.Bar"/>
</set>
```

<one-to-many>要素はカラムを宣言する必要がないことに注意してください。同様に####名を指定する必要もありません。

非常に重要な注意:もし<one-to-many>関連の<key>カラムがNOT NULLと宣言される場合、Hibernateはそれが関連を作成するか更新する時、制約違反を引き起こすかもしれません。 この問題を防ぐためには、manyの値の側(setやbag)にinverse="true"とマークした双方向関連を使用しなくてはなりません。この章で 後述する双方向関連を見てください。

6.5. lazy初期化(Lazy Initialization)

コレクション(配列以外)は、アプリケーションがアクセスを必要としたときだけデータベースから状態をロードするlazy初期化が出来ます。lazy初期化はユーザにとって透過的に起こります。そのためアプリケーションが通常このことについて関心を持つ必要がありません(実際、透過的なlazy初期化はHibernateが何故自身のコレクションの実装クラスを必要とするかという主な理由となります)しかしアプリケーションで以下のようなことをする場合問題を招きます。:

```
s = sessions.openSession();
User u = (User) s.find("from User u where u.name=?", userName, Hibernate.STRING).get(0);
Map permissions = u.getPermissions();
s.connection().commit();
s.close();

Integer accessLevel = (Integer) permissions.get("accounts");  // Error!
```

これは嫌な驚きに直面しそうになります。 Sessionがコミットされたときにpermissionコレクションが初期化されていないので、コレクションは決してその状態をロードできません。コレクションからコミットの直前へ読み込み行を移動して、修正してください(しかしこの問題を解決するさらに良い方法が他にもあります)。

もうひとつの方法としてlazyでないコレクションを使用してください。lazy初期化が上記のようにバグに結びつくこともあるのでデフォルトはlazy初期化を行いません。しかしながらlazy初期化がほとんどすべてのコレクションのために、特にエンティティのコレクションのために(効率の理由のための)使用されるように意図されています。

コレクションをlazy初期化する時に起こる例外はLazyInitializationExceptionでラップされています。

オプションのlazyプロパティを使用してlazyなコレクションを宣言するには以下のようにします。:

```
<set name="names" table="NAMES" lazy="true">
    <key column="group_id"/>
    <element column="NAME" type="string"/>
</set>
```

複数のアプリケーション・アーキテクチャーで、特にHibernateを使ってデータにアクセスするコードとそれを使用するコードが異なるアプリケーション層にある場合には、コレクションが初期化されるときにSessionが開いていることを保証する問題があります。以下はこの問題に対処する2つの基礎的な方法です:

- 一度ビューのレンダリングが完全になれば、ウェブ・ベースのアプリケーションではユーザ・リクエストの終端においてのみ、Sessionをクローズするためにサーブレット・フィルタを使用することができます。もちろんこのことはアプリケーション・インフラに例外処理の正確性が要求されます。ビューを提供している最中に例外が起こったときでさえ、ユーザーに処理が戻る前にSessionがクローズされ、そしてトランザクションが終了するということは極めて重要です。サ

ーブレット・フィルタはこのアプローチのために、Sessionにアクセス可能でなければなりません。私たちは現在のSessionを保持するために、#########な変数の使用を勧めます(実装例は1章の1.4節を見てください 項1.4. 「ネコと遊ぶ」)。

- 個別のビジネス層を備えたアプリケーションでは、ビジネスロジックは処理が返る前にウェブ層によって必要とされるすべてのコレクションを準備しなくてはなりません。このことはビジネス層は特定のユースケースに対し必要となるプレゼンテーション層／Web層にすでに、初期化されたデータをすべてロードし返すべきであるということを意味します。通常アプリケーションはウェブ層によって必要とされるコレクションそれぞれに対しHibernate.initialize()を呼ぶか(この呼び出しはSessionがクローズする前に呼ばれねばならない)、FETCH句のあるクエリーを使ってeagerにコレクションを復元します。

- 初期化していないコレクション(またはプロキシ)にアクセスする前に、以前にロードしたオブジェクトをupdate()やlock()を使って新しいSessionに関連させることが出来ます。アドホックなトランザクションのセマンティクスを導入したので、Hibernateはこれを自動的に行なうことが出来ません。

初期化せずにコレクションのサイズを得るためにHibernate Session APIのfilter()メソッドを使用することができます:

```
( (Integer) s.filter( collection, "select count(*)" ).get(0) ).intValue()
```

filter()やcreateFilter()もコレクション全体を初期化する必要なしに、効率的にコレクションの部分集合を復元するために使用されます。

6.6. ソートされたコレクション

Hibernateはjava.utilの実装であるjava.util.SortedMapとjava.util.SortedSetをサポートします。開発者はマッピング定義ファイルにコンパレーターを指定しなければなりません:

```
<set name="aliases" table="person_aliases" sort="natural">
    <key column="person"/>
    <element column="name" type="string"/>
</set>

<map name="holidays" sort="my.custom.HolidayComparator" lazy="true">
    <key column="year_id"/>
    <index column="hol_name" type="string"/>
    <element column="hol_date" type="date"/>
</map>
```

###属性として指定できる値はunsorted, natural, そしてjava.util.Comparatorの実装クラス名です。

ソートされたコレクションは実際にjava.util.TreeSetやjava.util.TreeMapのように動作します。

もしデータベース自身にコレクション要素を並べさせたいならばset, bag, mapのorder-by属性を使うと良いでしょう。この解決法はJDK 1.4かあるいはそれ以上のバージョンで利用可能です(LinkedHashSetまたはLinkedHashMapを使用して実装されます)。これはSQLクエリ内で整列を実行するのであってメモリ上ではありません。

```
<set name="aliases" table="person_aliases" order-by="name asc">
    <key column="person"/>
    <element column="name" type="string"/>
</set>
```

```
<map name="holidays" order-by="hol_date, hol_name" lazy="true">
    <key column="year_id"/>
    <index column="hol_name" type="string"/>
    <element column="hol_date type="date"/>
</map>
```

order-by属性の値がSQL命令であってHQL命令ではないことに注意してください！

関連は実行時にfilter()を使用することで任意のcriteriaによってソートされるかもしれません。

```
sortedUsers = s.filter( group.getUsers(), "order by this.name" );
```

6.7. <idbag>###

もし複合キーは悪いことであり、エンティティは人工の識別子(代理キー)を持つべきであるという考え方に賛成するならば、我々が示したmany-to-many関連と値のコレクションを複合キーを用いたテーブルへマッピングするということは少し奇妙に感じるかもしれません。この考えには確かに議論の余地があります。それは純粋な関連テーブルは代理キーによって十分な利益を得られるようには思えないからです(合成値のコレクションは利益があるかもしれませんが)。にもかかわらずHibernateは代理キーを使ったテーブルへのmany-to-many関連と値のコレクションを許す特徴を提供します。

<idbag>要素でbagの動作を持ったlist(またはCollection)をマッピングできます。

```
<idbag name="lovers" table="LOVERS" lazy="true">
    <collection-id column="ID" type="long">
        <generator class="hilo"/>
    </collection-id>
    <key column="PERSON1"/>
    <many-to-many column="PERSON2" class="eg.Person" outer-join="true"/>
</idbag>
```

ご覧のように<idbag>はエンティティ・クラスのように人工のidジェネレータを持っています。異なる代理キーはそれぞれのコレクションの列に割り当てられます。 しかしHibernateは特定の列の代理キーの値を発見するメカニズムを提供しません。

<idbag>の更新のパフォーマンスは通常の<bag>よりもいいことに注目してください。 Hibernateは個々の列を効果的に見つけることができ、ちょうどlistやmapやsetのように個別にそれらを更新、削除できます。

現在の実装では<idbag>コレクション識別子に対して、native識別子生成戦略はサポートされていません。

6.8. 双方向関連

双方向関連は関連のどちらの側からでもナビゲーションできます。２種類の双方向関連がサポートされています:

one-to-many
: 片側がsetかbag、もう片方の多重度が1です。

many-to-many
　両側がsetかbagです。

Hibernateはmany側がインデックス付けされたコレクション(listやmapや配列)である双方向one-to-many関連をサポートしていないことに注意してください。setまたはbagマッピングを使用してください。

２つのmany-to-many関連から同じデータベース・テーブルへのマッピングを使用するか、片側をinverseと宣言して双方向many-to-many関連を指定することが出来ます(どちらを選ぶかは選択次第です)。以下に、あるクラスからそれ自身のクラスへの双方向many-to-many関連の例を示します(各categoryは多くのitemsを持つことができます。また、各itemは多くのcategory内にありえます)。:

```
<class name="org.hibernate.auction.Category">
    <id name="id" column="ID"/>
    ...
    <bag name="items" table="CATEGORY_ITEM" lazy="true">
        <key column="CATEGORY_ID"/>
        <many-to-many class="org.hibernate.auction.Item" column="ITEM_ID"/>
    </bag>
</class>

<class name="org.hibernate.auction.Item">
    <id name="id" column="ID"/>
    ...

    <!-- inverse end -->
    <bag name="categories" table="CATEGORY_ITEM" inverse="true" lazy="true">
        <key column="ITEM_ID"/>
        <many-to-many class="org.hibernate.auction.Category" column="CATEGORY_ID"/>
    </bag>
</class>
```

関連の逆側へのみ行なわれた変更は永続化されません。 このことはHibernateがメモリ内のすべての双方向関連に対して2つの表現を持つことを意味します。1つはAからBへのリンクと、もうひとつはBからAでのリンクです。もしJavaのオブジェクト･モデルのことを考え、どのようにJavaでmany-to-many関連を作成するか考えればこれを理解するのは簡単です。:

```
category.getItems().add(item);          // The category now "knows" about the relationship
item.getCategories().add(category);     // The item now "knows" about the relationship

session.update(item);                     // No effect, nothing will be saved!
session.update(category);                 // The relationship will be saved
```

inverseではない側はデータベースへメモリ内の表現を保存するために使用されます。もし両方の側が変更を引き起こす場合、不必要なInsert/Updateそして恐らく外部キー制約違反を起こすでしょう。もちろん双方向one-to-many関連に対しても同じことが言えます。

同じテーブル･カラムへのone-to-many関連をmany-to-one関連としてマッピングし、many側の関連端を`inverse="true"`と宣言することで双方向one-to-many関連をマッピングできます。

```
<class name="eg.Parent">
    <id name="id" column="id"/>
    ....
    <set name="children" inverse="true" lazy="true">
        <key column="parent_id"/>
        <one-to-many class="eg.Child"/>
    </set>
</class>
```

```
<class name="eg.Child">
    <id name="id" column="id"/>
    ....
    <many-to-one name="parent" class="eg.Parent" column="parent_id"/>
</class>
```

inverse="true"である関連のone側のマッピングはカスケード操作に影響ありません。両者は異なる概念です。

6.9. 3項関連

3項関連のマッピングには２つの可能なアプローチがあります。 １つ目のアプローチはcomposite要素を使用することです(議論は後ほど)。もうひとつのアプローチは関連をインデックスとして使ったMapを使用することです。:

```
<map name="contracts" lazy="true">
    <key column="employer_id"/>
    <index-many-to-many column="employee_id" class="Employee"/>
    <one-to-many class="Contract"/>
</map>
```

```
<map name="connections" lazy="true">
    <key column="node1_id"/>
    <index-many-to-many column="node2_id" class="Node"/>
    <many-to-many column="connection_id" class="Connection"/>
</map>
```

6.10. Heterogeneousな関連

<many-to-any>および<index-many-to-any>要素はHeterogeneousな関連(異なる型が混在する関連)に対して提供されます。これらのマッピング要素は<any>要素と同様の作用をします。あるとしてもめったに使用されません。

6.11. コレクションの例

今までの節はかなり混乱させられるので以下の例を見てください。:

```
package eg;
import java.util.Set;

public class Parent {
    private long id;
    private Set children;

    public long getId() { return id; }
    private void setId(long id) { this.id=id; }

    private Set getChildren() { return children; }
    private void setChildren(Set children) { this.children=children; }

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

このクラスは eg.Childのインスタンスのコレクションを持っています。もし各々のchildが最大で

も１つのparentを持つならば最も自然なマッピングはone-to-many関連です。:

```
<hibernate-mapping>

    <class name="eg.Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" lazy="true">
            <key column="parent_id"/>
            <one-to-many class="eg.Child"/>
        </set>
    </class>

    <class name="eg.Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

このマッピング定義は下のテーブル定義へマッピングします。

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, name varchar(255), parent_id bigint )
alter table child add constraint childfk0 (parent_id) references parent
```

もしそのparentが要求されたならば双方向one-to-many関連を使用してください(Parent/Child関係の章を見てください)。:

```
<hibernate-mapping>

    <class name="eg.Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" inverse="true" lazy="true">
            <key column="parent_id"/>
            <one-to-many class="eg.Child"/>
        </set>
    </class>

    <class name="eg.Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
        <many-to-one name="parent" class="eg.Parent" column="parent_id" not-null="true"/>
    </class>

</hibernate-mapping>
```

NOT NULL制約に注意してください。

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null
                     primary key,
                     name varchar(255),
                     parent_id bigint not null )
alter table child add constraint childfk0 (parent_id) references parent
```

一方で、childが複数のparentを持てるならばmany-to-many関連が妥当です。:

```
<hibernate-mapping>
```

```
    <class name="eg.Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" lazy="true" table="childset">
            <key column="parent_id"/>
            <many-to-many class="eg.Child" column="child_id"/>
        </set>
    </class>

    <class name="eg.Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

テーブル定義は以下のようになります。:

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, name varchar(255) )
create table childset ( parent_id bigint not null,
                        child_id bigint not null,
                        primary key ( parent_id, child_id ) )
alter table childset add constraint childsetfk0 (parent_id) references parent
alter table childset add constraint childsetfk1 (child_id) references child
```

第7章 コンポーネントのマッピング

コンポーネントは、Hibernateを通して様々な状況の中で異なる目的のために再利用されます。

7.1. 依存オブジェクト

コンポーネントはエンティティではなくバリュー･タイプとして永続化された、包含されたオブジェクトです。つまりコンポーネントはエンティティではなく、バリュー･タイプです。コンポーネントという言葉はコンポジションというオブジェクト指向の概念を参照してください(アーキテクチャレベルのコンポーネントではありません)。例えば以下のようなPersonをモデルとします。:

```
public class Person {
    private java.util.Date birthday;
    private Name name;
    private String key;
    public String getKey() {
        return key;
    }
    private void setKey(String key) {
        this.key=key;
    }
    public java.util.Date getBirthday() {
        return birthday;
    }
    public void setBirthday(java.util.Date birthday) {
        this.birthday = birthday;
    }
    public Name getName() {
        return name;
    }
    public void setName(Name name) {
        this.name = name;
    }
    ......
    ......
}
```

```
public class Name {
    char initial;
    String first;
    String last;
    public String getFirst() {
        return first;
    }
    void setFirst(String first) {
        this.first = first;
    }
    public String getLast() {
        return last;
    }
    void setLast(String last) {
        this.last = last;
    }
    public char getInitial() {
        return initial;
    }
    void setInitial(char initial) {
        this.initial = initial;
    }
}
```

いまNameはPersonのコンポーネントとして永続化することができます。ここでNameの永続化属性に対

してゲッター、セッター・メソッドを定義しますが、インターフェースや識別子フィールドを宣言する必要がないことに注意してください

マッピング定義は以下のようになります。

```
<class name="eg.Person" table="person">
    <id name="Key" column="pid" type="string">
        <generator class="uuid.hex"/>
    </id>
    <property name="birthday" type="date"/>
    <component name="Name" class="eg.Name"> <!-- class attribute optional -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </component>
</class>
```

personテーブルはpid, birthday, initial, first , lastを持ちます。

すべてのバリュー・タイプのように、コンポーネントの参照は共有されません。コンポーネントのnull値におけるセマンティクスはアドホックです。含んでいるオブジェクトを再読み込みする場合、Hibernateはすべてのコンポーネントのカラムがnullである、すなわちコンポーネント自体がnullであると想定します。この事は大抵の場合問題ありません。

コンポーネントのプロパティはどんなHibernateの型でも構いません。(コレクション、many-to-one関連、他のコンポーネントなど)。ネストされたコンポーネントをめったに使わない使用法と考えるべきではありません。Hibernateは非常にきめの細かいオブジェクト・モデルをサポートするように意図されています。

<component>要素は親エンティティへの逆参照として、コンポーネント・クラスの属性をマッピングする<parent>サブ要素を使用できます。

```
<class name="eg.Person" table="person">
    <id name="Key" column="pid" type="string">
        <generator class="uuid.hex"/>
    </id>
    <property name="birthday" type="date"/>
    <component name="Name" class="eg.Name">
        <parent name="namedPerson"/> <!-- reference back to the Person -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </component>
</class>
```

7.2. 依存オブジェクトのコレクション

Hibernateはコンポーネントのコレクションをサポートしています(例えばName型の配列)。<element>タグの代わりに<composite-element>タグを使ってコンポーネントのコレクションを宣言してください。:

```
<set name="someNames" table="some_names" lazy="true">
    <key column="id"/>
    <composite-element class="eg.Name"> <!-- class attribute required -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </composite-element>
</set>
```

注意:コンポジット・エレメントのSetを定義したならばequals()とhashCode()を正しく実装することは重要です。

コンポジット・エレメントはコレクションではなく、コンポーネントを含んでいることもあります。コンポジット・エレメント自身がコンポーネントを含んでいる場合は<nested-composite-element>タグを使用してください。自身がコンポーネントを持っているコンポーネントのコレクション、といったケースはめったにありません。この段階までにone-to-many関連の方がより適切なのではないかと考えてください。エンティティとしてコンポジット・エレメントを再モデリングしてみてください。しかしJavaモデルは同じですが、リレーショナル・モデルと永続動作はまだわずかに異なることに注意してください。

もし<set>を使用するなら、コンポジット・エレメントのマッピングがnullを持つ属性をサポートしていないことに注意してください。Hibernateはオブジェクトを削除するときに、レコードを識別するために各々のカラムの値を使用する必要があるので、null値を持てません(コンポジット・エレメントのテーブルには別の主キーカラムはありません)。not-nullの属性をコンポジット・エレメントで使用するか、または<list>, <map>, <bag>, <idbag>を選択するかしなくてはなりません。

コンポジット・エレメントの特別なケースは入れ子の<many-to-one> 要素を持ったコンポジット・エレメントです。このようなマッピングはコンポジット・エレメント・クラスにmany-to-many関連テーブルの余分なカラムをマッピングすることを可能にします。次はOrderから、purchaseDate、price、quantityが関連からの属性であるItemへのmany-to-many関連の例です。

```
<class name="eg.Order" .... >
    ....
    <set name="purchasedItems" table="purchase_items" lazy="true">
        <key column="order_id">
        <composite-element class="eg.Purchase">
            <property name="purchaseDate"/>
            <property name="price"/>
            <property name="quantity"/>
            <many-to-one name="item" class="eg.Item"/> <!-- class########## -->
        </composite-element>
    </set>
</class>
```

3項関連(あるいは4項、それ以上)さえ可能です:

```
<class name="eg.Order" .... >
    ....
    <set name="purchasedItems" table="purchase_items" lazy="true">
        <key column="order_id">
        <composite-element class="eg.OrderLine">
            <many-to-one name="purchaseDetails class="eg.Purchase"/>
            <many-to-one name="item" class="eg.Item"/>
        </composite-element>
    </set>
</class>
```

コンポジット・エレメントは他のエンティティへの関連として同じシンタックスを使っているクエリ内で使用できます。

7.3. mapのインデックスとしてのコンポーネント

<composite-index>要素はMapのキーとしてコンポーネントのクラスをマッピングします。 必ずマッピングクラス上でhashCode()およびequals()を正確にオーバライドするようにしてください。

7.4. 複合識別子としてのコンポーネント

コンポーネントをエンティティ・クラスの識別子としても使用できます。コンポーネント・クラスは以下のある条件を満たさなければなりません。:

- java.io.Serializableを実装しなければなりません。
- データベースの複合キーと同等の概念としてequals() と hashCode()を一貫して再実装しなければなりません。

複合キーを生成するためにIdentifierGeneratorを使用することができません。代わりにアプリケーションはそれ自身の識別子を割り当てなくてはなりません。

複合識別子は保存する前にオブジェクトに割り当てられなくてはならないので、新しく作成されたインスタンスと以前のSessionで保存されたインスタンスを区別するために、識別子のunsaved-valueを使用することができません。

saveOrUpdate()あるいは保存/更新のカスケードを使用したければ、その代りにInterceptor.isUnsaved()を実装できます。代案として、新しい一時的インスタンスを示す値を指定するために<version>要素(または<timestamp>要素)にunsaved-value属性を設定できます。 この場合、エンティティのバージョンは(割り当てられた)識別子の代わりに使用されます。また、Interceptor.isUnsaved()をあなた自身が実装する必要がありません。

複合識別子クラスの宣言に<id>の代わりに<composite-id>タグ(<component>と同じ属性および要素)を使用してください。:

```
<class name="eg.Foo" table"FOOS">
    <composite-id name="compId" class="eg.FooCompositeID">
        <key-property name="string"/>
        <key-property name="short"/>
        <key-property name="date" column="date_" type="date"/>
    </composite-id>
    <property name="name"/>
    ....
</class>
```

テーブルFOOSへの外部キーも複合キーです。他のクラス用のマッピング定義の中でこれを宣言しなければなりません。Fooへの関連は以下のように宣言されます。:

```
<many-to-one name="foo" class="eg.Foo">
<!-- the "class" attribute is optional, as usual -->
    <column name="foo_string"/>
    <column name="foo_short"/>
    <column name="foo_date"/>
</many-to-one>
```

この新しい<column>タグも多重カラム・カスタム型によって使用されます。実際にどこでもcolumn属性の代わりとなります。Foo型の要素を備えたコレクションは以下のように使用します。:

```
<set name="foos">
    <key column="owner_id"/>
    <many-to-many class="eg.Foo">
        <column name="foo_string"/>
        <column name="foo_short"/>
        <column name="foo_date"/>
    </many-to-many>
</set>
```

他方で、通常<one-to-many>はカラムを宣言しません。

Foo自身がコレクションを含んでいる場合、コレクションは複合外部キーを必要とします。

```
<class name="eg.Foo">
    ....
    ....
    <set name="dates" lazy="true">
        <key>   <!-- a collection inherits the composite key type -->
            <column name="foo_string"/>
            <column name="foo_short"/>
            <column name="foo_date"/>
        </key>
        <element column="foo_date" type="date"/>
    </set>
</class>
```

7.5. 動的コンポーネント

以下のようにMap型のプロパティのマッピングも可能です：

```
<dynamic-component name="userAttributes">
    <property name="foo" column="FOO"/>
    <property name="bar" column="BAR"/>
    <many-to-one name="baz" class="eg.Baz" column="BAZ"/>
</dynamic-component>
```

<dynamic-component>マッピングのセマンティクスは<component>と同じです。この種のマッピングの利点はマッピング文書編集により、配置時にビーンの属性を決定できる点です。(さらに、DOMパーサを使用してマッピングドキュメントを実行時に操作することも可能です)。

第8章 継承のマッピング

8.1. 3つの戦略

Hibernateは3つの基本的な継承のマッピング戦略をサポートします。

- table per class hierarchy -1つのクラス階層ごとに1つのテーブル
- table per subclass -1つのサブクラスごとに1つのテーブル
- table per concrete class -1つの具象クラスごとに1つのテーブル(いくつか制限があります)

同じ継承階層の異なる枝について異なるマッピング戦略を使用することは可能ですが、table-per-concrete-classマッピングを適用する場合と同じ制限が適用されます。Hibernateは同じ<class>要素内で<subclass>マッピングと<joined-subclass>マッピングを一緒に使うことはサポートしていません。

Paymentインターフェースとそれを実装するCreditCardPayment, CashPayment, ChequePaymentがあると仮定してください。table-per-class-hierarchyマッピングは次のようになります。:

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
        ...
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        ...
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        ...
    </subclass>
</class>
```

このマッピングにはちょうど1つのテーブルが要求されます。このマッピング戦略には1つ大きな制限があります。それはサブクラスによって宣言されたカラムはNOT NULL制約をもてないことです。

table-per-subclassマッピングは次のようになります。:

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="AMOUNT"/>
    ...
    <joined-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
```

```
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
</class>
```

このマッピングには4つのテーブルが要求されます。3つのサブクラスのテーブルは、スーパークラスのテーブルへの主キー関連を持っています(したがって、リレーショナル・モデルは現実にone-to-one関連です)。

Hibernateのtable-per-subclass戦略の実装はdiscriminatorカラムを要求されないことに注意してください。他のO/Rマッパーは Hibernateとは違って、サブクラスのテーブルにdiscriminatorカラムを要求するtable-per-subclass戦略の実装を用いています。Hibernateが取ったアプローチは実装することは難しくなりますが関連の視点からみて恐らくより正確です。

これら2つのどちらのマッピング戦略にとっても、Paymentへのポリモーフィックな関連は<many-to-one>を使ってマッピングします。

```
<many-to-one name="payment"
    column="PAYMENT"
    class="Payment"/>
```

table-per-concrete-classマッピングは他の2つとは非常に異なります。

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
    <id name="id" type="long" column="CREDIT_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CREDIT_AMOUNT"/>
    ...
</class>

<class name="CashPayment" table="CASH_PAYMENT">
    <id name="id" type="long" column="CASH_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CASH_AMOUNT"/>
    ...
</class>

<class name="ChequePayment" table="CHEQUE_PAYMENT">
    <id name="id" type="long" column="CHEQUE_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CHEQUE_AMOUNT"/>
    ...
</class>
```

このマッピングには3つのテーブルが要求されます。Paymentインターフェースを明示的に言及する場所がないことに注意してください。代わりにHibernateの暗黙のポリモーフィズムを利用します。またPaymentのプロパティがそれぞれのサブクラスにマッピングされることにも注 意してください。

この場合、ポリモーフィズムを使ったPaymentへの関連は<any>を使ってマッピングされます。

```
<any name="payment"
        meta-type="class"
        id-type="long">
    <column name="PAYMENT_CLASS"/>
    <column name="PAYMENT_ID"/>
</any>
```

discriminator型の文字列からPaymentのサブクラスへのマッピングを扱うために、meta-typeとしてUserTypeを定義することはよりいいことでしょう。

```
<any name="payment"
        meta-type="PaymentMetaType"
        id-type="long">
    <column name="PAYMENT_TYPE"/> <!-- CREDIT, CASH ## CHEQUE -->
    <column name="PAYMENT_ID"/>
</any>
```

このマッピングに関して気づくことがもう1つあります。サブクラスはそれぞれを<class>要素としてマッピングしているので(かつPaymentは単なるインターフェースなので)、それぞれのサブクラスは簡単にtable-per-classマッピングやtable-per-subclassマッピングの継承階層の一部となれます!(また、今までどおりPaymentインターフェースに対するポリモーフィズムのクエリを使用することができます)。

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
    <id name="id" type="long" column="CREDIT_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="CREDIT_CARD" type="string"/>
    <property name="amount" column="CREDIT_AMOUNT"/>
    ...
    <subclass name="MasterCardPayment" discriminator-value="MDC"/>
    <subclass name="VisaPayment" discriminator-value="VISA"/>
</class>

<class name="NonelectronicTransaction" table="NONELECTRONIC_TXN">
    <id name="id" type="long" column="TXN_ID">
        <generator class="native"/>
    </id>
    ...
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="amount" column="CASH_AMOUNT"/>
        ...
    </joined-subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="amount" column="CHEQUE_AMOUNT"/>
        ...
    </joined-subclass>
</class>
```

もう一度言いますが、私たちはPaymentを明示的に指定しません。もしPaymentインターフェースに対し、例えば「from Payment」クエリを実行したなら、Hibernateは自動的にNonelectronicTransactionのインスタンスではなくCreditCardPayment(そしてそのサブクラスを返します。なぜならそれらはPaymentの実装だからです)、CashPayment および ChequePayment のインスタンスを返します。

8.2. 制限

Hibernateでは関連はちょうど1つの外部キーカラムとマッピングすると想定します。1つの外部キーによる多重関連は許容されますが(inverse="true"や、insert="false" update="false"と指定する必要がある)、しかし多重外部キーへの任意の関連をマッピングする方法はありません。このことは次のことを意味します。:

・ 関連が修正されるとき、更新されるのは常に同じ外部キーです。

・ 関連がlazyに取ってこられる場合、単一のデータベース・クエリが使用されます。

・ 関連がeagerに取ってこれらる場合、1つのアウター・ジョインを使ってフェッチされます。

それは特にtable-per-concrete-class戦略を使ってマッピングされたクラスへのポリモーフィズムを使ったone-to-many関連はサポートされていないということを暗示しています(この関連を取って来ることは多数のクエリ、あるいは多数のジョインを要求するでしょう)。

次のテーブルはHibernateにおけるtable-per-concrete-classマッピングの制限と、暗黙ポリモーフィズムの制限を示しています。

表 8.1. 継承のマッピングの特徴

継承戦略	many-to-oneのポリモーフィズム	one-to-oneのポリモーフィズム	one-to-manyのポリモーフィズム	many-to-manyのポリモーフィズム	ポリモーフィズムを使った**load()/get()**	ポリモーフィズムを使ったクエリ	ポリモーフィズムを使ったジョイン
table-per-class-hierarchy	<many-to-one>	<one-to-one>	<one-to-many>	<many-to-many>	s.get(Payment.class, id)	from Payment p	from Order o join o.payment p
table-per-subclass	<many-to-one>	<one-to-one>	<one-to-many>	<many-to-many>	s.get(Payment.class, id)	from Payment p	from Order o join o.payment p
table-per-concrete-class (implicit polymorphism)	<any>	サポートしていません	サポートしていません	<many-to-any>	クエリの使用	from Payment p	サポートしていません

第9章 永続データの操作

9.1. 永続オブジェクトの作成

オブジェクト（エンティティ・インスタンス）は特定の Session に対して 一時的 または 永続的です。 新しくインスタンス化されたオブジェクトは、当然一時的です。 Sessionは一時的なインスタンスをセーブするサービス（つまり永続化）を行います：

```
DomesticCat fritz = new DomesticCat();
fritz.setColor(Color.GINGER);
fritz.setSex('M');
fritz.setName("Fritz");
Long generatedId = (Long) sess.save(fritz);
```

```
DomesticCat pk = new DomesticCat();
pk.setColor(Color.TABBY);
pk.setSex('F');
pk.setName("PK");
pk.setKittens( new HashSet() );
pk.addKitten(fritz);
sess.save( pk, new Long(1234) );
```

引数が1個の save() は、fritz にユニークな識別子を生成し、代入します。 引数が2個の方は、与えられた識別子を使い pk を永続化しようとします。 ビジネス上の意味を持つ主キーを作成するために使えてしまうので、 一般に引数が2個の方はおすすめしません。 しかしBMPエンティティ・ビーンの永続化にHibernateを使うような特殊な状況なら、引数が2個の方は有用です。

外部キーのカラムに対して NOT NULL 制約がなければ、 好きな順序で関連オブジェクトを永続化できます。 外部キー制約に違反するリスクは全くありません。 しかし、間違った順序でオブジェクトを save() してしまうと、 NOT NULL 制約に違反するかもしれません。

9.2. オブジェクトのロード

もし識別子がわかっていれば、Session の load() メソッドを使い、 永続インスタンスを復元できます。 1番目のバージョンはクラス・オブジェクトを引数に取り、 新しくインスタンス化したオブジェクトに状態をロードします。 インスタンスを引数に取るもう1つのバージョンは、 HibernateをBMPエンティティ・ビーンと一緒に使う目的のためなら、特に有用です。 また他にも使いみちがあるかもしれません。（DIYインスタンス・プールなど）

```
Cat fritz = (Cat) sess.load(Cat.class, generatedId);
```

```
// ##########################
long pkId = 1234;
DomesticCat pk = (DomesticCat) sess.load( Cat.class, new Long(pkId) );
```

```
Cat cat = new DomesticCat();
// pk####cat#######
sess.load( cat, new Long(pkId) );
Set kittens = cat.getKittens();
```

該当する行が存在しなければ、load() は復帰不可能な例外をスローすることに注意してください。 クラスがプロキシでマッピングされていれば、load() は初期化されていないプロキシを返し、 オ

ブジェクトのメソッドを起動するまでは、実際にデータベースに問い合わせません。 実際にデータベースからロードせずに、オブジェクトへの関連を作成したければ、この振る舞いはとても役に立ちます。

該当する行が存在することが確実でなければ、`get()` メソッドを使うべきです。 これはすぐにデータベースに問い合わせ、該当する行が存在しなければnullを返します。

```
Cat cat = (Cat) sess.get(Cat.class, id);
if (cat==null) {
    cat = new Cat();
    sess.save(cat, id);
}
return cat;
```

SQLの `SELECT ... FOR UPDATE` を使い、オブジェクトをロードすることもできます。 Hibernateの `LockMode` についての議論は、次節を見てください。

```
Cat cat = (Cat) sess.get(Cat.class, id, LockMode.UPGRADE);
```

`FOR UPDATE` では、関連インスタンスや含んでいるコレクションは 選択 されない ことに注意してください。

`refresh()` メソッドを使うと、オブジェクトやすべてのコレクションをいつでもリロードできます。データベース・トリガを使って、オブジェクトのプロパティのいくつかが初期化されるときに、これは役に立ちます。

```
sess.save(cat);
sess.flush(); // SQL#INSERT######
sess.refresh(cat); // ######################
```

9.3. クエリの実行

探すオブジェクトの識別子がわからなければ、 `Session` の `find()` メソッドを使ってください。Hibernateはシンプルかつ強力なオブジェクト指向クエリ言語を用意しています。

```
List cats = sess.find(
    "from Cat as cat where cat.birthdate = ?",
    date,
    Hibernate.DATE
);

List mates = sess.find(
    "select mate from Cat as cat join cat.mate as mate " +
    "where cat.name = ?",
    name,
    Hibernate.STRING
);

List cats = sess.find( "from Cat as cat where cat.mate.bithdate is null" );

List moreCats = sess.find(
    "from Cat as cat where " +
    "cat.name = 'Fritz' or cat.id = ? or cat.id = ?",
    new Object[] { id1, id2 },
    new Type[] { Hibernate.LONG, Hibernate.LONG }
);

List mates = sess.find(
    "from Cat as cat where cat.mate = ?",
    izi,
```

```
    Hibernate.entity(Cat.class)
);

List problems = sess.find(
    "from GoldFish as fish " +
    "where fish.birthday > fish.deceased or fish.birthday is null"
);
```

`find()` の2番目の引数は、オブジェクトまたはオブジェクトの配列を受け取ります。 3番目の引数はHibernate型またはHibernate型の配列を受け取ります。 これらは、与えられたオブジェクトを `?` クエリ・プレースホルダにバインドするために使われます。 （JDBCの `PreparedStatement` のINパラメータへマッピングします。） JDBCと同じように、文字列操作ではなくこのバインディング機構を使うべきです。

ほとんどの組み込み型へのアクセスを提供するために、 `net.sf.hibernate.type.Type` のインスタンスとして、 `Hibernate` のクラスはstaticメソッドと定数をたくさん用意しています。

クエリが非常に多くのオブジェクトを返すにもかかわらず、それらすべてを使うわけではないならば、 `iterate()` メソッドを使うと良いパフォーマンスが得られます。 これは `java.util.Iterator` を返します。 イテレータは初期のSQLクエリから返される識別子を使い、 要求に応じてオブジェクトをロードします。（全部でn+1のセレクトになります。）

```
// id######
Iterator iter = sess.iterate("from eg.Qux q order by q.likeliness");
while ( iter.hasNext() ) {
    Qux qux = (Qux) iter.next();  // ############
    // ##############
    if ( qux.calculateComplicatedAlgorithm() ) {
        // ################
        iter.remove();
        // ################
        break;
    }
}
```

あいにく `java.util.Iterator` は何も例外を定義していないので、 発生するSQLやHibernateの例外は `LazyInitializationException` （ `RuntimeException` のサブクラス）でラップされます。

多くのオブジェクトがすでにSessionでロードされてキャッシュされているか、 クエリのリザルトが同じオブジェクトを何度も含むなら、 `iterate()` メソッドもよく動作します。 （データがキャッシュされたりイテレートされたりしなければ、 ほとんどの場合 `find()` の方がより高速です。）以下は `iterate()` を使ってコールすべきクエリの例です：

```

Iterator iter = sess.iterate(
    "select customer, product " +
    "from Customer customer, " +
    "Product product " +
    "join customer.purchases purchase " +
    "where product = purchase.product"
);
```

`find()` を使って上記のクエリをコールすると、 同じデータを何度も含む非常に大きなJDBC `ResultSet` が返されます。

Hibernateクエリはオブジェクトのタプルを返すことがあります。 その場合、各タプルは配列として返されます：

```
Iterator foosAndBars = sess.iterate(
    "select foo, bar from Foo foo, Bar bar " +
    "where bar.date = foo.date"
);
while ( foosAndBars.hasNext() ) {
    Object[] tuple = (Object[]) foosAndBars.next();
    Foo foo = tuple[0]; Bar bar = tuple[1];
    ....
}
```

9.3.1. スカラ・クエリ

`select` 句の中で、クラスのプロパティを指定できます。 さらにSQLの集計関数もコールできます。プロパティや集計は「スカラ」です。

```
Iterator results = sess.iterate(
        "select cat.color, min(cat.birthdate), count(cat) from Cat cat " +
        "group by cat.color"
);
while ( results.hasNext() ) {
    Object[] row = results.next();
    Color type = (Color) row[0];
    Date oldest = (Date) row[1];
    Integer count = (Integer) row[2];
    .....
}
```

```
Iterator iter = sess.iterate(
    "select cat.type, cat.birthdate, cat.name from DomesticCat cat"
);
```

```
List list = sess.find(
    "select cat, cat.mate.name from DomesticCat cat"
);
```

9.3.2. クエリ・インターフェイス

リザルトセットに境界（復元したい行の最大数や復元したい最初の行）を指定する必要があれば、`net.sf.hibernate.Query` のインスタンスを取得すべきです：

```
Query q = sess.createQuery("from DomesticCat cat");
q.setFirstResult(20);
q.setMaxResults(10);
List cats = q.list();
```

マッピング・ドキュメントの中で、名前付きのクエリを定義することもできます （マークアップとして翻訳される文字を含んでいれば `CDATA` セクションを使うことを忘れないでください）。

```
<query name="eg.DomesticCat.by.name.and.minimum.weight"><![CDATA[
    from eg.DomesticCat as cat
        where cat.name = ?
        and cat.weight > ?
] ]></query>
```

```
Query q = sess.getNamedQuery("eg.DomesticCat.by.name.and.minimum.weight");
q.setString(0, name);
q.setInt(1, minWeight);
List cats = q.list();
```

クエリ・インターフェイスでは名前付きのパラメータを使えます。 名前付きのパラメータとは、クエリ文字列中の `:name` の形式の識別子のことを言います。 `Query` には、名前付きのパラメータやJDBCスタイルの `?` パラメータにバインドするためのメソッドが用意されています。 JDBCとは違い、Hibernateの数パラメータは0から始まります。 名前付きパラメータの利点は、以下のようなものです：

- クエリ文字列中に現れる順序に影響されない。
- 同じクエリ中に複数回現れてもよい。
- 名前で意味を表現できる。

```
// ###############
Query q = sess.createQuery("from DomesticCat cat where cat.name = :name");
q.setString("name", "Fritz");
Iterator cats = q.iterate();
```

```
// #######
Query q = sess.createQuery("from DomesticCat cat where cat.name = ?");
q.setString(0, "Izi");
Iterator cats = q.iterate();
```

```
// #############
List names = new ArrayList();
names.add("Izi");
names.add("Fritz");
Query q = sess.createQuery("from DomesticCat cat where cat.name in (:namesList)");
q.setParameterList("namesList", names);
List cats = q.list();
```

9.3.3. スクローラブル・イテレーション

JDBCドライバがスクローラブル `ResultSet` に対応していれば、 `Query` インターフェイスを使って `ScrollableResults` を取得できます。 これを使うと、クエリのリザルトを柔軟にナビゲーションできます。

```
Query q = sess.createQuery("select cat.name, cat from DomesticCat cat " +
                           "order by cat.name");
ScrollableResults cats = q.scroll();
if ( cats.first() ) {

    // cat#####################################
    firstNamesOfPages = new ArrayList();
    do {
        String name = cats.getString(0);
        firstNamesOfPages.add(name);
    }
    while ( cats.scroll(PAGE_SIZE) );

    // cat#############
    pageOfCats = new ArrayList();
    cats.beforeFirst();
    int i=0;
    while( ( PAGE_SIZE > i++ ) && cats.next() ) pageOfCats.add( cats.get(1) );

}
```

9.3.4. コレクションのフィルタリング

コレクションの フィルタ は、 永続コレクションや配列に適用されるクエリの特別なタイプです。

クエリ文字列から、現時点のコレクション要素を意味する `this`

```
Collection blackKittens = session.filter(
    pk.getKittens(), "where this.color = ?", Color.BLACK, Hibernate.enum(Color.class)
);
```

返されるコレクションはbagです。

フィルタには `from` 句が必要ありません（あってもかまいません）。 またフィルタは、コレクション要素そのものを返せます。

```
Collection blackKittenMates = session.filter(
    pk.getKittens(), "select this.mate where this.color = eg.Color.BLACK"
);
```

9.3.5. Criteriaクエリ

HQLは極めて強力ですが、Javaコードの中に文字列として埋め込むのではなく、 オブジェクト指向のAPIを使って、動的にクエリを組み立てる方法を好む人もいます。 そういう人のために、Hibernateでは直感的な `Criteria` クエリAPIが用意されています。

```
Criteria crit = session.createCriteria(Cat.class);
crit.add( Expression.eq("color", eg.Color.BLACK) );
crit.setMaxResults(10);
List cats = crit.list();
```

SQLライクな構文が嫌いなら、多分これがHibernateを始める最も簡単な方法です。 このAPIはHQLより簡単に拡張できます。 アプリケーションで `Criterion` インターフェイスの独自の実装を用意することもできます。

9.3.6. ネイティブSQLのクエリ

`createSQLQuery()` を使い、SQLの中でクエリを表現することもできます。 その場合、中括弧でSQLのエイリアスを囲まなければなりません。

```
List cats = session.createSQLQuery(
    "SELECT {cat.*} FROM CAT {cat} WHERE ROWNUM<10",
    "cat",
    Cat.class
).list();
```

```
List cats = session.createSQLQuery(
    "SELECT {cat}.ID AS {cat.id}, {cat}.SEX AS {cat.sex}, " +
           "{cat}.MATE AS {cat.mate}, {cat}.SUBCLASS AS {cat.class}, ... " +
    "FROM CAT {cat} WHERE ROWNUM<10",
    "cat",
    Cat.class
).list()
```

Hibernateクエリと同じように、SQLクエリでも名前付きパラメータと位置パラメータが使えます。

9.4. オブジェクトの更新

9.4.1. 同じSessionでの更新

トランザクショナル永続インスタンス （ Session でロード、セーブ、作成、 クエリ実行されたオブジェクト）はアプリケーションで操作でき、 Session が フラッシュ されるときに 永続状態への変更が永続化されます。 （この章の終わりに議論します。） そのため、オブジェクトの状態を更新する最も簡単な方法は、 Session がオープンしている間に load() して直接操作する方法です：

```
DomesticCat cat = (DomesticCat) sess.load( Cat.class, new Long(69) );
cat.setName("PK");
sess.flush();  // cat####################
```

同じSessionで（オブジェクトをロードするための）SQL SELECT と （更新された状態を永続化するための）SQL UPDATE の両方が必要となるため、 このプログラミング・モデルはときどき非効率です。 そのため、Hibernateは代わりの方法を用意しています。

9.4.2. 関連付けをやめたオブジェクトの更新

多くのアプリケーションでは、1つのトランザクションでオブジェクトを復元して、 操作のためにUI層へ送って、新しいトランザクションで変更をセーブする必要があります。 （同時並行性の高い環境でこの種の方法を使うアプリケーションは、 普通トランザクションの分離を確実にするためにバージョン・データを使います。） この方法は、前の節で述べたプログラミング・モデルとはかなり異なったプログラミング・モデルが必要です。 Hibernateは Session.update() メソッドを用意することで、 このモデルに対応しています。

```
// ###Session#
Cat cat = (Cat) firstSession.load(Cat.class, catId);
Cat potentialMate = new Cat();
firstSession.save(potentialMate);

// ################
cat.setMate(potentialMate);

// #######Session#
secondSession.update(cat);  // cat###
secondSession.update(mate); // mate###
```

アプリケーションが更新を行おうとするとき、 識別子 catId を持つ Cat が secondSession ですでにロードされていれば、例外がスローされます。

与えられた一時的なインスタンスから到達可能な一時的なインスタンスを、 アプリケーションは個別に update() すべきです。 そしてそれは状態も更新したいときに 限ります 。 （ライフサイクル・オブジェクトを除く。後で議論します。）

Hibernateのユーザは新しい識別子を生成して一時的なインスタンスをセーブしたり、 現在の識別子に関連する永続状態を更新する汎用的なメソッドをリクエストしていました。 今ではこの機能を実装する saveOrUpdate() メソッドが用意されています。

Hibernateは識別子（またはバージョンかタイムスタンプ）プロパティの値で、 「新しい」（セーブされていない）インスタンスと「既存の」 （以前のSessionでセーブまたはロードされた）インスタンスを区別します。 <id>（または <version>, または <timestamp> ）の unsaved-value 属性のマッピングで、 どの値が「新しい」インスタンスを表すかを指定します。

```
<id name="id" type="long" column="uid" unsaved-value="null">
    <generator class="hilo"/>
</id>
```

unsaved-value は以下の値を取ります：

- any - 常にセーブする
- none - 常に更新する
- null - 識別子がnullのときにセーブする（これがデフォルトです）
- 妥当な識別子の値 - 識別子がnullか値が与えられたときにセーブする
- undefined - version や timestamp のデフォルトで、そのとき識別子がチェックされる

```
// ###Session#
Cat cat = (Cat) firstSession.load(Cat.class, catID);

// ###############
Cat mate = new Cat();
cat.setMate(mate);

// #######Session#
secondSession.saveOrUpdate(cat);   // ############cat#null###id########
secondSession.saveOrUpdate(mate);  // ##################mate#null#id########
```

新しいユーザには、saveOrUpdate() の使いみちと意味が 混乱しているように思われるかもしれません。 あるSessionのインスタンスを他の新しいSessionで使おうとしない限りは、 update() や saveOrUpdate() を使う必要はありません。 これらのメソッドのどちらも、アプリケーション全体を通して全く使わないユーザもいるでしょう。

普通 update() や saveOrUpdate() は、 以下のようなシナリオで使われます：

- アプリケーションが最初のSessionで、オブジェクトをロードする
- オブジェクトがUI層へ渡される
- オブジェクトにいくつか更新が行われる
- オブジェクトがビジネス・ロジック層に返される
- アプリケーションが2番目のSessionで update() をコールして、 これらの更新を永続化する

saveOrUpdate() は以下のことを行います：

- オブジェクトがこのSessionですでに永続化されていれば、何もしない
- オブジェクトに識別子プロパティがなければ、save() する
- オブジェクトの識別子が unsaved-value で指定されたcriteriaにマッチすれば、 save() する
- オブジェクトがバージョン付けされていれば。 （ version または timestamp ）、 バージョンが unsaved-value="undefined" （デフォルト値）でない限り、 バージョンが識別子のチェックに優先する
- もしSessionに関連付けられている他のオブジェクトが同じ識別子を持つなら、例外がスローされる

最後のケースは、 saveOrUpdateCopy(Object o) を使うことで避けられます。 このメソッドは、与えられたオブジェクトの状態を、同じ識別子を持つ永続オブジェクトにコピーします。 現在のSessionに関連付けられた永続インスタンスが存在しなければ、ロードされます。 このメソッドは永続インスタンスを返します。 与えられたインスタンスが、セーブされていないかデータベースに存在しなければ、 Hibernateはそれをセーブして、新しく永続化したインスタンスとして返します。 そうでなければ、与えられたインスタンスはSessionに関連付けられません。 オブジェクトの切り離しを使うほとんどのアプリケーションでは、 saveOrUpdate() と saveOrUpdateCopy() の両方のメソッドが必要です。

9.4.3. 関連付けをやめたオブジェクトの再関連付け

lock() メソッドを使えば、 アプリケーションを新しいSessionで更新されていないオブジェクトに、 再び関連付けることができます。

```
// #########
sess.lock(fritz, LockMode.NONE);
// ######################
sess.lock(izi, LockMode.READ);
// SELECT ... FOR UPDATE########################
sess.lock(pk, LockMode.UPGRADE);
```

9.5. 永続オブジェクトの削除

Session.delete() はデータベースからオブジェクトの状態を削除します。 もちろんアプリケーションがまだそれへの参照を保持しているかもしれません。 そのため delete() は永続インスタンスを一時的にするものと考えるのが一番です。

```
sess.delete(cat);
```

Hibernateクエリ文字列を delete() に渡すと、 多くのオブジェクトを一度に削除できます。

今ではもう、外部キー制約に違反するリスクなしで、 好きな順序でオブジェクトを削除することができます。 もちろんオブジェクトを間違った順序で削除すると、 外部キーの NOT NULL 制約に違反する可能性はまだあります。

9.6. フラッシュ

JDBCコネクションの状態をメモリ内部のオブジェクトの状態と同期する必要があるSQL文を Session が実行することがたまにあります。 この処理 フラッシュ はデフォルトでは以下のポイントで起こります。

- find() や iterate() のいくつかの起動から
- net.sf.hibernate.Transaction.commit() から
- Session.flush() から

SQL文は以下の順番で発行されます

1. すべてのエンティティの挿入は Session.save() を使って、 対応するオブジェクトと同じ順序でセーブされる
2. すべてのエンティティの更新
3. すべてのコレクションの削除
4. すべてのコレクション要素の削除、更新、挿入
5. すべてのコレクションの挿入
6. すべてのエンティティの削除（Session.delete() を使い 対応するオブジェクトが削除されるのと同じ順序で）

（例外は native ID生成を使い、 それらがセーブされるときにオブジェクトが挿入される場合です。）

明示的に flush() されるときを除いて、 いつ Session がJDBCコールを実行するかの絶対的な保証はなく、 実行の 順序 だけが保証されます。 しかしHibernateは Session.find(..) メソッドが、

古いデータや間違ったデータを返すことは絶対にないことを保証しています。

フラッシュがそう頻繁に起こらないように、デフォルトの振る舞いを変更することはできます。 FlushMode クラスは3つの異なったモードを定義しています。 これは「読み込みのみ」トランザクションの場合に最も有用です。 それは（非常に）わずかなパフォーマンスの改善のために使われるかもしれません。

```
sess = sf.openSession();
Transaction tx = sess.beginTransaction();
sess.setFlushMode(FlushMode.COMMIT); // ###################
Cat izi = (Cat) sess.load(Cat.class, id);
izi.setName(iznizi);
// #############....
sess.find("from Cat as cat left outer join cat.kittens kitten");
// izi################
...
tx.commit(); // ###########
```

9.7. Sessionの終了

Sessionの終了には4つの明確なフェーズがあります：

- Sessionをフラッシュする
- トランザクションをコミットする
- Sessionをクローズする
- 例外に対処する

9.7.1. Sessionのフラッシュ

偶然 Transaction APIを使うことになったとしても、 このステップを心配する必要はありません。 これはトランザクションがコミットされるときに、暗黙的に実行されます。 そうでなければ、すべての変更がデータベースと同期を取ることを確実にするために、 Session.flush() をコールすべきです。

9.7.2. データベース・トランザクションのコミット

Hibernateの Transaction APIを使うなら、このようになります：

```
tx.commit(); // Session#########################
```

もしJDBCトランザクションを独自に管理するなら、 JDBCコネクションを手作業で commit() すべきです。

```
sess.flush();
sess.connection().commit();  // JTA###############
```

もし変更をコミット しないと決めたなら：

```
tx.rollback();  // ##################]></programlisting>

            <para>
                or:
            </para>
```

```
            <programlisting><![CDATA[// JTA############################
sess.connection().rollback();
```

トランザクションをロールバックするなら、 Hibernateの内部状態の整合性が取れていることを確実にするために、 現在のSessionを即座にクローズして捨てるべきです。

9.7.3. Sessionのクローズ

`Session.close()` へのコールはSessionの終了をマークします。 JDBCコネクションがSessionから放棄されることが `close()` の主な意味です。

```
tx.commit();
sess.close();
```

```
sess.flush();
sess.connection().commit();  // JTA###############
sess.close();
```

独自にコネクションを用意すれば、`close()` はそれへの参照を返すので、 手作業でクローズしてプールに戻すことができます。 さもなければ `close()` はそれをプールに戻します。

9.8. 例外処理

Hibernateを使っていると例外に出くわすことがあります。 それは大抵、チェックされた `HibernateException` です。 この例外は、入れ子状の根本原因（root cause）を持つことができます。 それにアクセスするには、`getCause()` メソッドを使ってください。

`Session` が例外をスローした場合は、 即座にトランザクションをロールバックし、`Session.close()` をコールし、`Session` インスタンスを捨ててください。 `Session` のメソッドの中には、 Sessionを整合性のある状態に しない ものがあります。 つまりこれは、Hibernateがスローする例外は全て致命的なものであるということです。 そのため、チェックされた `HibernateException` を `RuntimeException` に変換したい場合があるかもしれません （最も簡単なのは、`HibernateException.java` の `extends` を置き換えて、Hibernateを再コンパイルする方法です）。 チェック例外はHibernateの遺物であることに注意してください。 これはHibernateの将来のメジャーバージョンで変更されます。

データベースとのやりとりの最中にスローされたSQLExceptionは、 Hibernateによって JDBCExceptionのサブクラスへの変換が試みられます。 根本原因のSQLExceptionには、`JDBCException.getCause()` をコールすることでアクセスできます。 HibernateによるSQLExceptionから適切なJDBCExceptionのサブクラスへの変換は、 SessionFactoryに関連付けられたSQLExceptionConverterに基づいて行われます。 デフォルトでは、どのSQLExceptionConverterが使われるかは、 設定されたデータベースの方言によって決まります。 しかし、カスタムの実装をプラグすることも可能です （詳細については、SQLExceptionConverterFactoryクラスのjavadocを見てください）。 標準のJDBCExceptionのサブタイプは、以下のものです：

- `JDBCConnectionException` – ベースとなる JDBCコミュニケーションに関するエラーを示唆します。
- `SQLGrammarException` – 発行したSQLに関する 文法や構文の問題を示唆します。
- `ConstraintViolationException` – 何らかの 整合性制約違反を示唆します。
- `LockAcquisitionException` – リクエストされた操作の実行に 必要なロックレベルの取得に関す

るエラーを示唆します。
- `GenericJDBCException` – 他のカテゴリに該当しない 一般的な例外です。

常に全ての例外は、現時点の `Session` とトランザクションに対して致命的であると考えられます。HibernateがさまざまなSQLExceptionの型を区別できるようになったからといって、 Sessionに対してこれらの例外が復帰可能なものになったわけではありません。 この例外の型階層によって、例外の原因のカテゴリに対する プログラムの反応を容易にしただけです。

以下の例外処理イディオムは、Hibernateを使うアプリケーションでの典型例です：

```
Session sess = factory.openSession();
Transaction tx = null;
try {
    tx = sess.beginTransaction();
    // do some work
    ...
    tx.commit();
}
catch (Exception e) {
    if (tx!=null) tx.rollback();
    throw e;
}
finally {
    sess.close();
}
```

また、JDBCトランザクションを自分で管理するときは：

```
Session sess = factory.openSession();
try {
    // do some work
    ...
    sess.flush();
    sess.connection().commit();
}
catch (Exception e) {
    sess.connection().rollback();
    throw e;
}
finally {
    sess.close();
}
```

また、JTAに登録されたデータソースを使うときは：

```
SessionContext ctx = ... ;
Session sess = factory.openSession();
try {
    // do some work
    ...
    sess.flush();
}
catch (Exception e) {
    // ctx.setRollbackOnly();
    throw new EJBException(e);
}
finally {
    sess.close();
}
```

（JTAで管理される環境における）アプリケーション・サーバは、 `java.lang.RuntimeException` に対してのみ、 トランザクションを自動ロールバックするということを心に留めておいてください。アプリケーションの例外（つまり `HibernateException` ）が発生した場合は、 `EJBContext` において

setRollbackOnly() を自分でコールしなければいけません。 もしくは上の例のように、自動ロールバックするために RuntimeException （例えば、EJBException ）でラップしなければいけません。

9.9. ライフサイクルとオブジェクトのグラフ

関連オブジェクトのグラフの中のすべてのオブジェクトをセーブまたは更新するためには、 以下のどちらかを行わなければなりません

- 個々のオブジェクトに対して save() または saveOrUpdate() または update() するか
- cascade="all" または cascade="save-update" を使い、 関連オブジェクトをマッピングする。

同じように、グラフのすべてのオブジェクトを削除するには、以下のどちらかを行います

- 個々のオブジェクトを delete() するか
- cascade="all" または cascade="all-delete-orphan" または cascade="delete" を使い、関連オブジェクトをマッピングする。

おすすめ：

- もし子オブジェクトの生存期間が親オブジェクトの生存期間に制限を受けるなら、cascade="all" と指定して、 それを ライフサイクル・オブジェクトにしてください。
- さもなければ明示的にそれを save() し、 そしてアプリケーション・コードから delete() してください。 もし本当に余分なタイピングをしてセーブを行いたければ、 cascade="save-update" を使い、 明示的に delete() してください。

cascade="all" と関連をマッピングすると（many-to-oneまたはコレクション）、 その関連は 親 / 子 スタイルの関係としてマークされます。 それは親のセーブ/更新/削除を行うと、子のセーブ/更新/削除が行われる関係です。 さらに永続的な親から子への単なる参照すると、子のセーブ/更新が行われます。 しかしメタファーは不完全です。 cascade="all-delete-orphan" でマッピングされた <one-to-many> 関連の場合を除いて、 子が親から参照されなくなっても自動的には削除 されません。 操作のカスケードの正確な意味は以下のようになります：

- もし親がセーブされれば、すべての子は saveOrUpdate() に渡される
- もし親が update() または saveOrUpdate() に渡されれば、 すべての子は saveOrUpdate() に渡される
- もし一時的な子が永続的な親から参照されるようになれば、 子は saveOrUpdate() に渡される
- もし親が削除されれば、すべての子は delete() に渡される
- もし一時的な子が永続的な親から参照されなくなっても、 特別には何も起こらない。 （必要ならアプリケーションで明示的に子を削除すべきです） ただし cascade="all-delete-orphan" の場合は除きます。 この場合は「みなしご」は削除されます。

Hibernateは「到達可能性による永続化」を完全には実装していません。 それは（非効率的な）永続ガベージ・コレクションを含意するからです。 しかし一般的な要求のため、Hibernateは他の永続オブジェクトから参照されたときに、 エンティティが永続化されるようになる考えをサポートしています。 cascade="save-update" とマークされた関連はこのように振る舞います。 もしアプリケーションを通してこの方法を使いたければ、 単純に <hibernate-mapping> 要素の default-cascade 属性を指定するだけです。

9.10. インターセプタ

Interceptor インターフェイスはSessionからアプリケーションへの コールバックを用意しています。 それはセーブ、更新、削除、ロードの前に、 永続オブジェクトのプロパティを見たり操作したりできます。 これの使いみちで考えられるものは、コード監査のための情報を追跡することです。 例えば以下の Interceptor は、 Auditable が作成されたとき createTimestamp〉を自動的に設定し、 Auditable が更新されたとき lastUpdateTimestamp プロパティを更新します。

```
package net.sf.hibernate.test;

import java.io.Serializable;
import java.util.Date;
import java.util.Iterator;

import net.sf.hibernate.Interceptor;
import net.sf.hibernate.type.Type;

public class AuditInterceptor implements Interceptor, Serializable {

    private int updates;
    private int creates;

    public void onDelete(Object entity,
                         Serializable id,
                         Object[] state,
                         String[] propertyNames,
                         Type[] types) {
        // ######
    }

    public boolean onFlushDirty(Object entity,
                                Serializable id,
                                Object[] currentState,
                                Object[] previousState,
                                String[] propertyNames,
                                Type[] types) {

        if ( entity instanceof Auditable ) {
            updates++;
            for ( int i=0; i < propertyNames.length; i++ ) {
                if ( "lastUpdateTimestamp".equals( propertyNames[i] ) ) {
                    currentState[i] = new Date();
                    return true;
                }
            }
        }
        return false;
    }

    public boolean onLoad(Object entity,
                          Serializable id,
                          Object[] state,
                          String[] propertyNames,
                          Type[] types) {
        return false;
    }

    public boolean onSave(Object entity,
                          Serializable id,
                          Object[] state,
                          String[] propertyNames,
                          Type[] types) {

        if ( entity instanceof Auditable ) {
            creates++;
            for ( int i=0; i<propertyNames.length; i++ ) {
                if ( "createTimestamp".equals( propertyNames[i] ) ) {
                    state[i] = new Date();
                    return true;
                }
            }
```

```
        }
        return false;
    }

    public void postFlush(Iterator entities) {
        System.out.println("Creations: " + creates + ", Updates: " + updates);
    }

    public void preFlush(Iterator entities) {
        updates=0;
        creates=0;
    }

    ......
    ......

}
```

インターセプタはSessionを作成するときに指定します。

```
Session session = sf.openSession( new AuditInterceptor() );
```

また、Configuration を使い、グローバルレベルでインターセプタを設定することもできます：

```
new Configuration().setInterceptor( new AuditInterceptor() );
```

9.11. メタデータAPI

Hibernateはすべてのエンティティとバリューの型の非常にリッチなメタレベル・モデルを必要とします。 このモデルがアプリケーション自体にとってとても役に立つことが時折あります。 例えばアプリケーションが「賢い」ディープ・コピーアルゴリズムを実装するために、 Hibernateのメタデータを使うかもしれません。 それはどのオブジェクトをコピーすべきか（例 更新可能なバリュー型）、 またはすべきでないか（例 更新不能バリュー型、可能なら関連エンティティ）を 理解するために必要です。

Hibernateは ClassMetadata と CollectionMetadata インターフェイスと Type 階層を通して、メタデータを公開しています。 メタデータ・インターフェイスのインスタンスは SessionFactory から取得することができます。

```
Cat fritz = ......;
Long id = (Long) catMeta.getIdentifier(fritz);
ClassMetadata catMeta = sessionfactory.getClassMetadata(Cat.class);
Object[] propertyValues = catMeta.getPropertyValues(fritz);
String[] propertyNames = catMeta.getPropertyNames();
Type[] propertyTypes = catMeta.getPropertyTypes();
// ##################################
// TODO: #########?
Map namedValues = new HashMap();
for ( int i=0; i<propertyNames.length; i++ ) {
    if ( !propertyTypes[i].isEntityType() && !propertyTypes[i].isCollectionType() ) {
        namedValues.put( propertyNames[i], propertyValues[i] );
    }
}
```

第10章 トランザクションと同時並行性

Hibernate自体はデータベースではなく、ライトウェイトなオブジェクト/リレーショナル・ マッピング・ツールです。 そのためトランザクション管理は、ベースとなっているデータベース・コネクションへ任されます。 コネクションがJTAに登録されていれば、 Session が実行する操作は、より大きなJTAトランザクションの一部です。 オブジェクト指向の意味を追加したJDBCのシン・アダプタとして、Hibernateを捉えることもできます。

10.1. Configuration、Session、Factory

SessionFactory は生成することが高価で、 スレッドセーフなオブジェクトです。 これはすべてのアプリケーション・スレッドで共有すべきです。 それに対して Session は高価ではなく、スレッドセーフでもありません。 これは1つのビジネス・プロセスに対して1度だけ使われて、捨てられるべきです。 例えばサーブレット・ベースのアプリケーションでHibernateを使うとき、 サーブレットは以下のようにして SessionFactory を取得できます。

```
SessionFactory sf = (SessionFactory)getServletContext().getAttribute("my.session.factory");
```

サービス・メソッドへのコールのたびに、新しく Session が作成され、 flush() され、コネクションを commit() し、 close() して、最後には捨てられます。 (SessionFactory は、JNDIや、staticな Singleton のヘルパ変数に取っておくこともできます。)

ステートレス・セッション・ビーンでも同じようなアプローチを使えます。 ビーンは setSessionContext() で SessionFactory を取得します。 各ビジネス・メソッドで Session が作成され、flush() され、 close() されます。 当然、アプリケーションではコネクションを commit() すべきではありません。 (JTAに任せると、コンテナ管理トランザクションで、データベースが自動的に関連付けられます。)

以前述べたHiberante Transaction APIを使うことにします。 Hibernate Transaction の1つの commit() は、 状態をフラッシュし、(JTAトランザクションの特別な扱いで) データベース・コネクションをコミットします。

flush() の意味を確実に理解してください。 フラッシュはメモリ内部の変更を永続ストアに同期させますが、逆は成り立ち ません。 すべてのHibernateJDBCコネクション/トランザクションに対して、 コネクションのトランザクション分離レベルがHibernateが実行する すべての操作に適用されることに注意してください。

次節では、トランザクション・アトミシティを確実にするもう一つの方法を議論します (バージョン付けを使います)。 これは「高度な」方法なので、注意して使わなければなりません。

10.2. スレッドとコネクション

Hibernate Sessionを作成するときは、以下の規則を守るべきです:

- データベース・コネクションに対して、複数の同時並行 Session や Transaction インスタンスを作成しない。
- データベース・トランザクションに対して複数の Session を作成するときは、 極めて慎重に行う。 Session 自体はロードされたオブジェクトに加えられた更新を覚えていますが、 他の

Session は古いデータを見てしまうかもしれません。
- Session はスレッドセーフでは ありません 。 そのため、2つの同時並行スレッドで、同じ Session にアクセスしてはいけません。 普通 Session は、作業単位（unit-of-work）1つにすぎません。

10.3. オブジェクト・アイデンティティの考慮

アプリケーションは2つの異なる仕事単位で、同じ永続状態に並行にアクセスできます。 しかし永続クラスのインスタンスを、2つの Session インスタンスで共有することはできません。 つまりアイデンティティに対する、2つの異なる考え方があります：

データベース・アイデンティティ
```
foo.getId().equals( bar.getId() )
```

JVMアイデンティティ
```
foo==bar
```

オブジェクトが1つの 特定の Session に関連付けられている場合は、 2つの考え方は等価です。しかし2つの異なるSessionで、アプリケーションが「同じ」（永続アイデンティティ） ビジネス・オブジェクトに並行にアクセスする場合は、 2つのインスタンスは実際には「違うもの」です（JVMアイデンティティ）。

この方法はHibernateとデータベースに、同時並行性を任せます。 Session に対して1つのスレッドに関連付けられているか、 オブジェクト・アイデンティティである限りは、 アプリケーションがビジネス・オブジェクトを同期させる必要は全くありません。 （1つの Session で、アプリケーションはオブジェクトの比較のために == を安全に使うことができます。）

10.4. optimistic同時並行性制御

ビジネス・プロセスの多くはデータベース・アクセスとユーザ・インタリーブの 一連の相互作用全体を必要とします。 ウェブやエンタープライズアプリケーションでは、データベース・トランザクションが ユーザとの対話に及ぶことは許されません。

ビジネス・プロセスの分離の維持は、アプリケーション層の責務の一部になります。 そのためこのプロセスを 長い間実行される アプリケーション・トランザクション と呼びます。 1つのアプリケーション・トランザクションは、 普通いくつかのデータベース・トランザクションに及びます。もしこれらのデータベース・トランザクション（の最後の1つ）が更新されたデータを格納すれば、よりアトミックになり、他のすべてのトランザクションは単にデータを読むだけになります。

高い同時並行性とスケーラビリティを持つ唯一の方法は、 バージョン付けを使うoptimisticな同時並行性制御です。 Hibernateではoptimistic同時並行性を使うアプリケーション・コードを書くための方法が 3種類用意されています。

10.4.1. 自動バージョン付けを使う長いSession

1つの Session インスタンスとその永続インスタンスは、 アプリケーション・トランザクション全体に渡って使われます。

Session はバージョン付けを使った楽観的ロックを使い、 データベース・トランザクションの多くが1つの論理的なアプリケーション・トランザクションに 現れることを確実にします。 ユーザとの対話を待つ間、Session はベースとなる JDBCコネクションと切断されています。 この方法は、データベース・アクセスという点では最も効率的です。 アプリケーションは自身のバージョンをチェックすることも、 切り離したインスタンスの再接続も考慮する必要はありません。

```
// foo####Session##############
session.reconnect();
foo.setProperty("bar");
session.flush();
session.connection().commit();
session.disconnect();
```

foo オブジェクトは、それをロードした Session をまだ知っています。 Session がJDBCコネクションを持つとすぐに、 オブジェクトに対する変更をコミットします。

Session がユーザが考える時間で格納するには大きすぎるなら、 このパターンは問題になります。 例えば HttpSession はできるだけ小さく保たなければなりません。 Session も（強制的な）ファーストレベル・キャッシュで、 すべてのロードされたオブジェクトを含んでいるように、 この戦略はリクエスト/レスポンス・サイクルが少ないときだけに使うことができます。 Session> はすぐに古いデータを持つようになるので、 これを強くおすすめします。

10.4.2. 自動バージョン付けを使う多くのSession

永続ストアとの対話はそれぞれ、新しい Session で起こります。 しかし同じ永続インスタンスが、データベースとの各対話で再利用されます。 元は他の Session でロードされた切り離されたオブジェクトの状態を、 アプリケーションが操作します。 そして Session.update() や Session.saveOrUpdate() を使い、 それらを「再接続」します。

```
// foo####Session##############
foo.setProperty("bar");
session = factory.openSession();
session.saveOrUpdate(foo);
session.flush();
session.connection().commit();
session.close();
```

オブジェクトが更新されていないことが確実なら、 update() の代わりに lock() を、 そして LockMode.READ を使うことができます。 （すべてのキャッシュを使わずに、バージョン・チェックを実行します）

10.4.3. アプリケーション・バージョン・チェック

データベースとの対話は新しい Session で起こります。 そこでは操作する前にデータベースからすべての永続インスタンスをリロードします。 この方法は、アプリケーション・トランザクションの分離を確実にするために、 アプリケーションが自分のバージョンチェックを実行することを強制します。 （もちろんHibernateはあなたのためにバージョン番号を 更新 します。） この方法はデータベース・アクセスの点では最も効率が悪いです。 これはエンティティEJBに最もよく似ています。

```
// foo####Session##############
session = factory.openSession();
int oldVersion = foo.getVersion();
session.load( foo, foo.getKey() );
```

```
if ( oldVersion!=foo.getVersion ) throw new StaleObjectStateException();
foo.setProperty("bar");
session.flush();
session.connection().commit();
session.close();
```

もちろん、もしデータ同時並行性の低い環境でバージョン・チェックが必要なければ、 この方法を使いバージョン・チェックをスキップできます。

10.5. Sessionの切断

上で述べた最初の方法は、ユーザの考慮時間に及ぶビジネス・プロセス全体に対して、 1つの `Session` を維持するものです。 （例えば、サーブレットはユーザの `HttpSession` で1つの `Session` を保持します。） パフォーマンスを考慮すると以下のようにすべきです。

1. `Transaction` （またはJDBCコネクション）をコミットし、
2. JDBCコネクションから `Session` を切断する

ユーザの行動を待つ前に、これらを行います。 `Session.disconnect()` メソッドは、 JDBCコネクションからSessionを切断し、 （もしコネクションを用意していなければ）プールに返します。

`Session.reconnect()` は新しいコネクションを取得し （または自分で用意し）、Sessionを再開します。 再接続したあと、更新していないデータのバージョン・チェックを強制するために、 他のトランザクションで更新されたどんなオブジェクトに対しても `Session.lock()` をコールできます。 更新中の どんなデータもロックする必要はありません。

以下はその例です：

```
SessionFactory sessions;
List fooList;
Bar bar;
....
Session s = sessions.openSession();

Transaction tx = null;
try {
    tx = s.beginTransaction();

    fooList = s.find(
            "select foo from eg.Foo foo where foo.Date = current date"
        // DB2#date#######
    );
    bar = (Bar) s.create(Bar.class);

    tx.commit();
}
catch (Exception e) {
    if (tx!=null) tx.rollback();
    s.close();
    throw e;
}
s.disconnect();
```

その後で：

```
s.reconnect();

try {
    tx = s.beginTransaction();
```

```
    bar.setFooTable( new HashMap() );
    Iterator iter = fooList.iterator();
    while ( iter.hasNext() ) {
        Foo foo = (Foo) iter.next();
        s.lock(foo, LockMode.READ);    // foo###############
        bar.getFooTable().put( foo.getName(), foo );
    }

    tx.commit();
}
catch (Exception e) {
    if (tx!=null) tx.rollback();
    throw e;
}
finally {
    s.close();
}
```

これから Transaction と Session の間がどのようなmany-to-oneであるか、 Session がどのようにアプリケーションとデータベースの間の 対話を表現するかがわかるでしょう。 Transaction は対話をデータベース・レベルの仕事単位に分解します。

10.6. 悲観的ロック

ユーザがロックの戦略について時間をかけて考えるとは期待していません。 普通はJDBCコネクションに対する分離レベルを指定して、 データベースにすべての仕事を任せることで十分です。 しかし高度なユーザなら、排他的な悲観的ロックや、新しいトランザクションの開始時に ロックを再取得したいと思うことあるかもしれません。

Hibernateは常にデータベースのロック機構を使います。 そのため、メモリ中にオブジェクトをロックしないでください。

LockMode クラスは、さまざまなロックのレベルを定義しています。 それらはHibernateによって取得されるかもしれません。 ロックは以下のような機構によって取得できます：

- LockMode.WRITE はHibernateが行を更新や挿入するとき、自動的に取得されます。
- LockMode.UPGRADE はこの構文をサポートするデータベースに対して、 SELECT ... FOR UPDATE を使い、明示的なユーザ・リクエストを取得できます。
- LockMode.UPGRADE_NOWAIT はOracleに対して、 SELECT ... FOR UPDATE NOWAIT を使い、明示的なユーザ・リクエストを取得できます。
- LockMode.READ はRepeatable ReadまたはSerializableの 分離レベルで、Hibernateがデータを読み込むときに自動的に取得されます。 明示的なユーザ・リクエストによる再取得かもしれません。
- LockMode.NONE はロックされていないことを表します。 すべてのオブジェクトは Transaction の終了時に、 このロック・モードに切り替えられます。 update() または saveOrUpdate() のコールを通して Sessionに関連付けられたオブジェクトも、このロック・モードで開始されます。

「明示的なユーザ・リクエスト」は以下の方法の1つとして表現されます：

- LockMode を指定した、Session.load() のコール。
- Session.lock() のコール。
- Query.setLockMode() のコール。

Session.load() が UPGRADE または UPGRADE_NOWAIT でコールされ、 リクエストされたオブジェクト

がまだSessionによってロードされていなければ、 オブジェクトは `SELECT ... FOR UPDATE` を使ってロードされます。 リクエストしたものよりも低い制限のロックですでにロードされていたオブジェクトに対して `load()` がコールされれば、 Hibernateはそのオブジェクトに対して `lock()` をコールします。

ロック・モードが `READ`, `UPGRADE`, `UPGRADE_NOWAIT` と指定されていれば、 `Session.lock()` はバージョン番号をチェックします （ `UPGRADE` または `UPGRADE_NOWAIT` の場合は、 `SELECT ... FOR UPDATE` が使われます。）。

データベースがリクエストしたロック・モードに対応していなければ、 Hibernateは（例外をスローする代わりに）他の適切なモードを使います。 これによりアプリケーションがポータブルになります。

第11章 HQL: Hibernateクエリ言語

HibernateはSQLに非常によく似た(意図的に似せた)強力な問い合わせ言語を備えています。 しかしSQLに似た構文に惑わされないでください。HQLは完全にオブジェクト指向であり、継承、ポリモーフィズム、関連といった概念を理解してください。

11.1. 大文字と小文字の区別

クエリはJavaのクラス名とプロパティ名を除いて大文字、小文字を区別しません。従ってSeLeCTはsELEct と同じで、かつSELECT とも同じですがnet.sf.hibernate.eg.FOOはnet.sf.hibernate.eg.Fooとは違い、かつfoo.barSet はfoo.BARSETとも違います。

このマニュアルでは小文字のHQLキーワードを使用します。大文字のキーワードのクエリの方が読みやすいと感じるユーザーもいると思います。しかし我々はJavaのコードにHQLを埋め込むときには、この習慣はコードを見づらくすると考えています。

11.2. from句

可能な、もっとも単純なHibernateクエリは次の形式です。:

```
from eg.Cat
```

これは単純にeg.Catクラスのインスタンスをすべて返します。

ほとんどの場合クエリのほかの部分でCatを参照するので、エイリアスを割り当てる必要があるでしょう。

```
from eg.Cat as cat
```

このクエリはCatインスタンスにエイリアスのcatを割り当てます。そのため我々はあとのクエリでこのエイリアスを使用できます。asキーワードはオプションです。つまりこのように書けます。 :

```
from eg.Cat cat
```

直積、つまりクロスジョインによって多数のクラスが出現することもあります。

```
from Formula, Parameter
```

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

ローカル変数のJavaのネーミング基準と一致した、頭文字に小文字を使ったクエリ・エイリアスをつけることはいい習慣です(例えばdomesticCat)。

11.3. 関連とジョイン

関連するエンティティあるいは値コレクションの要素にさえ、####を使ってエイリアスを割り当てることが出来ます。

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

from eg.Cat as cat left join cat.mate.kittens as kittens

from Formula form full join form.parameter param
```

サポートしているジョインのタイプはANSI SQLと同じです。

- #########
- #############
- #############
- #######（たいていの場合使いづらい）

#########、#############、#############の構造は省略されることもあります。

```
from eg.Cat as cat
    join cat.mate as mate
    left join cat.kittens as kitten
```

加えて、ジョインによるフェッチは関連や値のコレクションを親オブジェクトと一緒に1度のselect句で初期化します。これは特にコレクションの場合に有用です。関連とコレクションに対するマッピング定義ファイルのアウター・ジョインとlazy初期化の宣言を効果的に無効にします。

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

ジョインによるフェッチは関連するオブジェクトがwhere句(または他のどんな句でも)で使われてはならないので、通常エイリアスを割り当てる必要がありません。また関連オブジェクトは問い合わせ結果として直接返されません。代わりに親オブジェクトを通してアクセスできます。

現在の実装では、1つのクエリ中ではただひとつのコレクションのロールだけがフェッチされることに注意してください(その他のものはすべて事前にフォーマットされません)。またfetch構造はscroll()やiterate()を使ったクエリ呼び出しで使用できないことにも注意してください。 最後に##############と###############は有用ではないことに注意してください。

11.4. Select句

select句は以下のようにどのオブジェクトと属性をクエリ・リザルトセットに返すかを選択します。:

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

上のクエリは他のCatsのmatesを選択します。実際には次のように、より簡潔に表現できます。:

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

特別のelements関数を利用して、コレクション要素を選択できます。次のクエリは任意のCatのkittensをすべて返します。

```
select elements(cat.kittens) from eg.Cat cat
```

クエリはコンポーネント型のプロパティを含む、あらゆるバリュー・タイプのプロパティも返せます。:

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

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

クエリは多数のオブジェクトと(または)プロパティを`Object[]`型の配列として返せます。

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

あるいは`Family`クラスが適切なコンストラクタを持っているとするならば、

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

タイプ・セーフなJavaオブジェクトを返せます。

11.5. 集約関数

HQLのクエリはプロパティの集約関数も返せます。

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

コレクションは`select`句の集約関数の内部にも記述できます。

```
select cat, count( elements(cat.kittens) )
from eg.Cat cat group by cat
```

サポートしている集約関数は以下のものです。

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

`distinct`と`all`キーワードはSQLと同じ意味で使われ、同じ意味を持っています。

```
select distinct cat.name from eg.Cat cat

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

11.6. ポリモーフィックなクエリ

下のクエリは`Cat`だけでなく、`DomesticCat`のようなサブクラスのインスタンスも返します。

```
from eg.Cat as cat
```

Hibernateクエリーはfrom句中で、どんなJavaクラスあるいはインターフェースも指定できます。クエリはそのクラスを拡張した、もしくはインターフェースを実装した全ての永続クラスを返します。次のクエリは永続オブジェクトをすべて返します:

```
from java.lang.Object o
```

Namedインターフェースは様々な永続クラスによって実装されます。:

```
from eg.Named n, eg.Named m where n.name = m.name
```

最後の2つのクエリは、2つ以上のSQL SELECTを要求していることに注意してください。 このことはorder by句がリザルトセット全体を正確には整列しないことを意味します(さらにそれは、Query.scroll()を使用してこれらのクエリを呼ぶことができないことを意味します。)。

11.7. where句

where句は返されるインスタンスのリストを絞ることができます。

```
from eg.Cat as cat where cat.name='Fritz'
```

上のHQLはFritzという名前のCatを返します。

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

上のHQLは、Fooのプロパティと等しいdateプロパティを持ったbarインスタンスが存在する、すべてのFooインスタンスを返します。コンパウンド・パス表現はwhere句を非常に強力にします。

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

上のクエリはテーブル・ジョイン(インナー・ジョイン)を持つSQLクエリに変換します。

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

もし上のクエリを記述したらクエリ内に4つのテーブル・ジョインを必要とするSQLクエリに変換されます。

=オペレーターは以下のように、プロパティだけでなくインスタンスを比較するためにも使われます。

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

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

id(小文字)は特別なプロパティであり、オブジェクトのユニークな識別子に参照を付けるために使

用できます。(さらに、そのプロパティ名を使用できます。)

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

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

2番目のクエリは効率的です。テーブル・ジョインを要求されません！

また複合識別子のプロパティも使用できます。ここでPersonがcountryとmedicareNumberからなる複合識別子を持つと仮定します。

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

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

もう一度言いますが、2番目のクエリはテーブル・ジョインを要求されません。

同様にclassは特別なプロパティであり、ポリモーフィックな永続化におけるインスタンスの識別値にアクセスします。where句に埋め込まれたJavaのクラス名はその識別値に変換されます。

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

またコンポーネントやコンポジット・ユーザ・タイプ(またそのコンポーネントのコンポーネントなど)のプロパティも指定できます。しかし決してコンポーネント・タイプのプロパティ中で終了するパス表現をしようとしないでください。例えばもしstore.ownerがaddressコンポーネントを持つエンティティならば以下のような結果となります。

```
store.owner.address.city    // OK
store.owner.address         // ###!
```

"any"型は特別なプロパティであるidとclassを持ち、以下の方法でジョインを表現することを可能にします(ただしAuditLog.itemは<any>でマッピングされたプロパティです)。

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

log.item.classとpayment.classが上記のクエリ中で全く異なるデータベース・カラムの値を参照するということに注意してください。

11.8. 式

SQLのwhere句で記述することが出来る式のほとんどをHQLでも記述できます。:

- 数学的なオペレーター：+, -, *, /
- 2項比較演算：=, >=, <=, <>, !=, like
- 論理演算：and, or, not
- ストリングの連結：||
- SQLのスカラー関数：upper() , lower()など
- グループ分けを表す括弧：( )
- in, between, is null

- JDBC入力パラメータ：?
- 名前付きパラメータ:name, :start_date, :x1
- SQL文字列：'foo', 69, '1970-01-01 10:00:01.0'
- Javaのpublic static final定数：eg.Color.TABBY

in と betweenは以下のように使用できます。:

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

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

また、否定の形式は下のように記述されます。

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

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

同様にis null や is not nullはnull値をテストするために使用できます。

booleanはHibernateの設定ファイルでHQL query substitutionsを宣言することで、式として簡単に使用できます。:

```
<property name="hibernate.query.substitutions">true 1, false 0</property>
```

こうすることで下記のHQLをSQLに変換するときにtrue , falseキーワードを1 , 0に置き換えます。:

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

特別なプロパティsize、または特別な関数size()を使ってコレクションのサイズをテストできます。:

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

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

インデックスが付けられたコレクションについては、minIndex と maxIndexを使用して最小および最大のインデックスが参照できます。同様にminElement と maxElementを使用して、基本型のコレクション要素の最小と最大が参照できます。

```
from Calendar cal where cal.holidays.maxElement > current date
```

さらに関数形式もあります。(それらは上記の構造と異なり、大文字小文字を区別しません。):

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

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

コレクションの要素やインデックスの集合(elements , indices関数)、またはサブクエリの結果が渡される場合には、SQL関数any, some, all, exists, inがサポートされます。

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

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

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

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

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

これらsize, elements, indices, minIndex, maxIndex, minElement, maxElementの構造には、以下の使用上の制限があることに注意してください。:

- where句では：副問い合わせを備えたデータベースでのみ使用可能。
- select句では:##および##だけが使用可能。

インデックスが付けられたコレクション(配列、list、map)の要素はindexによって引用できます(where句でのみ)。

```
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
```

[]の内部の表現は演算式も可能です。

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

HQLはさらにmany-to-one関連の要素、あるいは値のコレクションに対してindex()関数を用意しています。

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

ベースとなるデータベースがサポートしているスカラーSQL関数が使用できます

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

もしまだ全てを理解していないなら、下のクエリをSQLでどれだけ長く、読みづらく出来るか考えてください。:

```
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)
```

Hint:例えばこのように出来ます。

```
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
    )
```

11.9. order by句

クエリによって返されたlistは、返されたクラスやコンポーネントの属性によって整列できます。:

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

オプションのasc と descはそれぞれ昇順か降順の整列を示します

11.10. group by句e

集約値を返すクエリは、返されたクラスやコンポーネントの任意のプロパティによってグループ化できます。:

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

select foo.id, avg( elements(foo.names) ), max( indices(foo.names) )
from eg.Foo foo
group by foo.id
```

注:副問い合わせのないデータベースでも、select節内部を構成するelements および indicesを使用できます。

having句も同様に可能です。

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

もしベースとなるデータベースによってサポートされているなら、having 句と order by句中でSQL関数および集計関数の使用は可能です(つまりMySQLでは使用不可能です)。

```
select cat
from eg.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句もorder by句も演算式を含むことが出来ないことに注意してください。

11.11. 副問い合わせ

副問い合わせをサポートするデータベースについてはHibernateはクエリ内でサブクエリをサポートします。括弧(SQLの集約関数呼び出しによる事が多い)でサブクエリを囲まなければなりません。関連副問い合わせ(外部クエリ中の別名を参照するサブサブクエリのこと)さえ許可されます。

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

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

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

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

11.12. HQLの例

Hibernateクエリは非常に強力で複雑にできます。実際、クエリ言語の威力はHibernateの主要なセールス・ポイントの一つです。ここに最近のプロジェクトで使用したクエリと非常によく似た例があります。ほとんどのクエリはこれらの例より簡単に記述できることに注意してください!

以下のクエリは特定の顧客と与えられた最小の合計値に対する未払い注文の注文ID、商品の数、注文の合計を合計値で整列して返します。結果として返されるSQLクエリはORDER, ORDER_LINE, PRODUCT, CATALOG および PRICEテーブルに対し４つのinner joinと(関連しない)副問い合わせを持ちます。

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

何て巨大なクエリなのでしょう!普段私は副問い合わせをあまり使いません。したがって私のクエリは実際には以下のようにします。:

```
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
```

次のクエリは各状態の支払い数をカウントします。ただしすべての支払いが現在の利用者による最新の状態変更であるAWAITING_APPROVAL状態である場合を除きます。このクエリは2つのインナー・ジョインとPAYMENT, PAYMENT_STATUS および PAYMENT_STATUS_CHANGEテーブルに対する関連副問い合わせを備えた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
```

もし私がsetの代わりにlistとしてstatusChangesコレクションをマッピングしたならば、はるかに簡単にクエリを記述できるでしょう。

```
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
```

次のクエリは現在のユーザが所属する組織に対するアカウントおよび未払いの支払いをすべて返すMS SQLサーバーのisNull()関数を使用しています。このクエリは３つのインナー・ジョインと1つのアウター・ジョイン 、そしてACCOUNT, PAYMENT, PAYMENT_STATUS, ACCOUNT_TYPE, ORGANIZATION および ORG_USERテーブルに対する副問い合わせ持ったSQLに変換されます。

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

いくつかのデータベースについては、(関連させられた)副問い合わせの使用を避ける必要があるでしょう。

```
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
```

11.13. Tips & Tricks

実際に結果を返さなくてもクエリの結果数を数えることができます。:

```
( (Integer) session.iterate("select count(*) from ....").next() ).intValue()
```

コレクションのサイズにより結果を整列するため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)
```

使用しているデータベースが副問い合わせをサポートする場合、クエリのwhere句でサイズによる選択条件を設定できます:

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

使用しているデータベースが副問い合わせをサポートしない場合は、次のクエリを使用してください:

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

インナー・ジョインをしているせいで上の解決法がmessageの件数がゼロのUserを返すことができないならば、以下の形式が使えます。:

```
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
```

JavaBeanのプロパティは、名前付きのクエリ・パラメータに結びつけることが出来ます。 :

```
Query q = s.createQuery("from foo in class Foo where foo.name=:name and foo.size=:size");
q.setProperties(fooBean); // fooBean#getName()#getSize()#####
List foos = q.list();
```

コレクションはフィルタ付きQueryインターフェースを使用することでページをつけることができます。:

```
Query q = s.createFilter( collection, "" ); // the trivial filter
q.setMaxResults(PAGE_SIZE);
q.setFirstResult(PAGE_SIZE * pageNumber);
List page = q.list();
```

コレクションの要素はクエリ・フィルターを使って整列、グループ分けすることが出来ます。:

```
Collection orderedCollection = s.filter( collection, "order by this.amount" );
Collection counts = s.filter( collection, "select this.type, count(this) group by this.type" );
```

コレクションを初期化せずにコレクションのサイズを得ることができます。:

```
( (Integer) session.iterate("select count(*) from ....").next() ).intValue();
```

第12章 Criteriaクエリ

Hibernateは現在、直感的で拡張可能なcriteriaクエリAPIを用意しています。今ではこのAPIは、成熟したHQLクエリの機能に比べて力不足です。特にcriteriaクエリは射影や集約をサポートしていません。

12.1. Criteriaインスタンスの作成

net.sf.hibernate.Criteriaインタフェースは特定の永続性クラスに対するクエリを表現しています。SessionはCriteriaインスタンスのファクトリです。

```
Criteria crit = sess.createCriteria(Cat.class);
crit.setMaxResults(50);
List cats = crit.list();
```

12.2. リザルトセットの絞込み

個別のクエリ・クライテリオン(問い合わせの判定基準)はインターフェイス net.sf.hibernate.expression.Criterion のインスタンスです。net.sf.hibernate.expression.Expressionクラスはある組み込みのCriterion型を取得するためのファクトリメソッドを定義します。

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "Fritz%") )
    .add( Expression.between("weight", minWeight, maxWeight) )
    .list();
```

Expressionは論理的にグループ化されます。

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "Fritz%") )
    .add( Expression.or(
        Expression.eq( "age", new Integer(0) ),
        Expression.isNull("age")
    ) )
    .list();
```

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.in( "name", new String[] { "Fritz", "Izi", "Pk" } ) )
    .add( Expression.disjunction()
        .add( Expression.isNull("age") )
            .add( Expression.eq("age", new Integer(0) ) )
            .add( Expression.eq("age", new Integer(1) ) )
            .add( Expression.eq("age", new Integer(2) ) )
    ) )
    .list();
```

元々あるcriterion型(Expressionのサブクラス)はかなりの範囲に及びますが、特に有用なのはSQLを直接指定できるものです。

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.sql("lower({alias}.name) like lower(?)", "Fritz%", Hibernate.STRING) )
    .list();
```

プレースホルダ{alias}は、問い合わせを受けたエンティティの行の別名によって置き換えられます。

12.3. 結果の整列

net.sf.hibernate.expression.Orderを使って結果を整列することができます。

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "F%")
    .addOrder( Order.asc("name") )
    .addOrder( Order.desc("age") )
    .setMaxResults(50)
    .list();
```

12.4. 関連

createCriteria()を使い関連をナビゲートすることで、容易に関係するエンティティに制約を指定できます。

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "F%")
    .createCriteria("kittens")
        .add( Expression.like("name", "F%")
    .list();
```

2番目のcreateCriteria()は、kittensコレクションの要素を参照する新しいCriteriaインスタンスを返すことに注意してください。

以下のような方法も、状況により有用です。

```
List cats = sess.createCriteria(Cat.class)
    .createAlias("kittens", "kt")
    .createAlias("mate", "mt")
    .add( Expression.eqProperty("kt.name", "mt.name") )
    .list();
```

(ここでcreateAlias()は新しいCriteriaインスタンスを作成しません。)

前の2つのクエリによって返されるCatインスタンスによって保持されるkittensコレクションは、criteriaによって事前にフィルタリングされないことに注意してください。もしcriteriaと対応するkittenを取得したいなら、returnMaps()を使わなければなりません。

```
List cats = sess.createCriteria(Cat.class)
    .createCriteria("kittens", "kt")
        .add( Expression.eq("name", "F%") )
    .returnMaps()
    .list();
Iterator iter = cats.iterator();
while ( iter.hasNext() ) {
    Map map = (Map) iter.next();
    Cat cat = (Cat) map.get(Criteria.ROOT_ALIAS);
    Cat kitten = (Cat) map.get("kt");
}
```

12.5. 動的関連のフェッチ

setFetchMode()を使い、実行時に関連の復元方法を指定することができます。

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "Fritz%") )
    .setFetchMode("mate", FetchMode.EAGER)
    .setFetchMode("kittens", FetchMode.EAGER)
    .list();
```

このクエリはアウター・ジョインでmate と kittensの両方をフェッチします。

12.6. クエリの例

net.sf.hibernate.expression.Exampleクラスを使って、与えられたインスタンスから クエリ・クライテリオンを構築することができます。

```
Cat cat = new Cat();
cat.setSex('F');
cat.setColor(Color.BLACK);
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .list();
```

バージョン・プロパティ、識別子、関連は無視されます。デフォルトではnull値のプロパティは除外されます。

どのようにExampleを適用するか、状況に応じて調整することができます。

```
Example example = Example.create(cat)
    .excludeZeroes()           //###############
    .excludeProperty("color")  //"color"#################
    .ignoreCase()              //############################
    .enableLike();             //#######like######
List results = session.createCriteria(Cat.class)
    .add(example)
    .list();
```

関連オブジェクトにcriteriaを指定するExampleを使うことも可能です。

```
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .createCriteria("mate")
        .add( Example.create( cat.getMate() ) )
    .list();
```

第13章 ネイティブSQLクエリ

データベースのネイティブSQL方言を使ってクエリを表現することもできます。Oracleにおける CONNECTキーワードのように、そのデータベースに特有の機能を利用するときに有用です。これにより直接の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が可能になります。

13.1. QueryベースSQLの作成

SQLクエリは、普通のHQLクエリと同じQueryインターフェイスを通して公開されています。 唯一の違いはSession.createSQLQuery()を使うことです。

```
Query sqlQuery = sess.createSQLQuery("select {cat.*} from cats {cat}", "cat", Cat.class);
sqlQuery.setMaxResults(50);
List cats = sqlQuery.list();
```

以下の3つのパラメータがcreateSQLQuery()に提供されます。 :

- SQLクエリ文字列
- テーブルの別名
- クエリにより返される永続クラス

別名はマッピングするクラスのプロパティを参照するSQL文字列の内部で使用されます(このケースではCat)。別名のString配列と対応するクラスのClass配列を与えることで、ひとつの行に つき複数のオブジェクトを取得できます。

13.2. 別名とプロパティ参照

上で使われている{cat.*}は「すべてのプロパティ」を意味する短絡表記です。明示的にプロパティを連ねることもできますが、それにはプロパティそれぞれにSQLカラムの別名を与えるようにしなければなりません。これらのカラムの別名のプレースホルダはテーブルの別名で修飾されるプロパティ名です。以下の例では、異なったテーブル（cat_log）からマッピング･メタデータで定義されたクラスへCatを復元します。where句の中でもプロパティの別名を使用できることに注意してください。

```
String sql = "select cat.originalId as {cat.id}, "
    + "  cat.mateid as {cat.mate}, cat.sex as {cat.sex}, "
    + "  cat.weight*10 as {cat.weight}, cat.name as {cat.name}"
    + "     from cat_log cat where {cat.mate} = :catId"
List loggedCats = sess.createSQLQuery(sql, "cat", Cat.class)
    .setLong("catId", catId)
    .list();
```

注意:明示的にプロパティを列挙するときは、クラスとそのサブクラスのすべてのプロパティを含めなければなりません！

13.3. 名前付きSQLクエリ

名前付きのクエリはマッピング・ドキュメントで定義することができ、名前付きHQLクエリと全く同じ方法でコールすることができます。

```
List people = sess.getNamedQuery("mySqlQuery")
    .setMaxResults(50)
    .list();
```

```
<sql-query name="mySqlQuery">
    <return alias="person" class="eg.Person"/>
    SELECT {person}.NAME AS {person.name},
           {person}.AGE AS {person.age},
           {person}.SEX AS {person.sex}
    FROM PERSON {person} WHERE {person}.NAME LIKE 'Hiber%'
</sql-query>
```

第14章 パフォーマンスの改善

14.1. コレクションのパフォーマンスの理解

すでにコレクションの話題に結構な時間を割きました。この節ではコレクションが実行時にどのように振る舞うかについて話題を2，3取り上げます。

14.1.1. 分類

Hibernateは以下の3つの種類のコレクションを定義しています。

- 値のコレクション
- one-to-many関連
- many-to-many関連

この分類法ではさまざまなテーブルと外部キー関係を分類しました。しかしこれは関係モデルについて知る必要のあることについては、ほとんど何も教えてはくれません。関係構造とパフォーマンスの特徴を完全に理解するためには、コレクションの行を更新、削除するためにHibernateが使用する主キーの構造を考えなくてはなりません。このことは以下の分類を示唆しています。

- インデックス付きのコレクション
- set
- bag

インデックス付きのコレクション（map、list、配列）はすべて、`<key>` と `<index>`カラムからなる主キーがあります。この場合コレクションの更新は通常、非常に効率的です。主キーは効率的にインデックスを付けられ、Hibernateが更新または削除しようとするとき、特定の行は効率的に検索されます。

setは`<key>`で構成される主キーと要素のカラムを持っています。これはコレクション要素の型のいくつかについては効率的ではないかもしれません。特に複合要素、ラージ･テキスト、バイナリ･フィールドなどは非効率です。それはデータベースが複合主キーを効率的に索引付けできないことがあるからです。一方で、one-to-many関連やmany-to-many関連、特に人工識別子の場合は同じくらい効率的です。（余談：`SchemaExport`に`<set>`の主キーを実際に生成させたいなら、すべてのカラムを`not-null="true"`と定義しなければなりません。）

bagは最悪のケースです。bagは重複した要素の値を許し、インデックスカラムを持たないので、主キーは定義されません。Hibernateには重複した行同士を区別する方法がありません。Hibernateはこの問題を、変更が行なわれたときには常に完全にコレクションを削除し(一度の`DELETE`で)、もう一度作成しなおすことで解決します。 これは非常に非効率です。

one-to-many関連にとって、「主キー」がデータベーステーブルの物理的な主キーではないかもしれないことに注意してください。しかしこのケースでは以上の分類はまだ有用です。（この分類は、まだHibernateがどのようにコレクションの個別の行を「検索」しているかを反映しています。）

14.1.2. コレクションの更新に最も効率的なlist、map、set

上での議論から、インデックス付きのコレクションと(大抵は)setが要素の追加、削除、更新などの操作を最も効率的に行うことは明らかです。

まず間違いなく、many-to-many関連や値のコレクションにおいて、インデックス付きのコレクションのSetに対するアドバンテージが１つ以上あります。Setはその構造のため、要素が「変更」されたときHibernateは決して行をUPDATEしません。setへの変更は必ず(個別の行に対する)INSERT とDELETEを通して行われます。繰り返しになりますが、これはone-to-many関連には当てはまりません。

配列がlazyではいけないという決まりなので、list、map、setが最も効率の良いコレクション型であると結論付けるでしょう(setはいくつかの値のコレクションに対しては効率的ではありませんが)。

Hibernateのアプリケーションで、setは最も一般的な種類のコレクションであると思われます。

今回のリリースではドキュメント化されていない特徴があります。 *<idbag>*を使ったマッピングは値のコレクションやmany-to-many関連のためのbagセマンティクスの実装であり、このケースにおいては他のどのスタイルのコレクションよりも効率的です！

14.1.3. インバース･コレクションに最も効率的なbagとlist

bagを見放してしまう前に、bag（そしてlist）がsetよりずっと効率的である特別なケースをご紹介します。inverse="true"であるコレクションに対して(例えば、普通のone-to-many双方向関連)、初期化（フェッチ）の 必要なしにbagまたはlistに要素を追加することができます！これはCollection.add() や Collection.addAll()がSetとは違って、bagまたはListには必ずtrueを返さなければならない からです。これにより、以下のようなよくあるコードが非常に高速になります。

```
Parent p = (Parent) sess.load(Parent.class, id);
    Child c = new Child();
    c.setParent(p);
    p.getChildren().add(c);  //#######################
    sess.flush();
```

14.1.4. 一括削除

コレクションの要素を一つ一つ削除するのが極めて非効率であることがあります。Hibernateは全くの愚か者というわけではありませんから、新しい空のコレクションのケース（例えばlist.clear()をコールするケース）では一つ一つの要素を削除すべきでないとわかっています。このケースでは、HibernateはDELETEを一度だけ発行し、それですべて済みます！

サイズが20のコレクションに1つ要素を追加し、そして2つ要素を削除するとします。HibernateはINSERTステートメントを1つとDELETEステートメントを2つ発行します（もしコレクションがbagでなければ）。これは確かに望ましい動作です。

しかし、2つの要素を残して18個を削除し、それから新しい要素を3つ追加するとします。この場合2つの方法がありえます。

- 1ずつ18個の行を削除し、そして3つの行を挿入する。

- コレクション全体を削除し（一つのSQL DELETEで）、そして5つの要素をすべて（一つ一つ）挿入する。

このケースにおいておそらく2番目の方法の方が高速だろうとわかるほど、Hibernateは賢くありません(そして恐らく、Hibernateがそのように賢いのは望ましいことではないでしょう。例えば、そのような振る舞いでデータベーストリガが混乱してしまうなどということがあるかもしれません)。

幸い元のコレクションを捨て（つまり参照をやめる）、現在の要素すべて持つような新しくインスタンス化したコレクションを返すことで、いつでもこの振る舞い(つまり2番目の戦略)を強制することができます。時にこれは非常に有用で強力な方法となることがあります。

コレクションのマッピングについての章で、永続性コレクションに対してどのようにlazy初期化を使用できるか説明しました。CGLIBを使用することで、通常のオブジェクト参照に対しても同様の効果を得ることが出来ます。また、Sessionレベルでどのように永続オブジェクトをキャッシュするかについても言及しました。より積極的なキャッシュ戦略はclass-by-class戦略に基づいた設定が可能です

次の節で、必要に応じてより高いパフォーマンスを達成するための機能の使い方について説明します。

14.2. lazy初期化のためのプロキシ

Hibernateは実行時バイトコードエンハンスメントを使い、永続オブジェクトに対してlazy初期化プロキシを実装しています(CGLIBという素晴らしいライブラリを使用しています)。

マッピング定義ファイルで、そのクラスのプロキシ･インタフェースとしてクラスやインタフェースを定義します。以下のように、クラス自身を指定するのがお勧めの方法です：

```
<class name="eg.Order" proxy="eg.Order">
```

プロキシの実行時の型はOrderのサブクラスになります。プロキシとして指定するクラスは、少なくともパッケージレベルの可視性のデフォルト･コンストラクタを実装しなければならないことに注意してください。

このアプローチをポリモーフィックなクラスに拡張するとき気をつけるべきことがあります。

```
<class name="eg.Cat" proxy="eg.Cat">
    ......
    <subclass name="eg.DomesticCat" proxy="eg.DomesticCat">
        .....
    </subclass>
</class>
```

まず初めに、Catのインスタンスはは DomesticCatにキャストできません。たとえ基となるインスタンスがDomesticCatのインスタンスだとしてもです。

```
Cat cat = (Cat) session.load(Cat.class, id);  // instantiate a proxy (does not hit the db)
if ( cat.isDomesticCat() ) {                  // ##############db#######
    DomesticCat dc = (DomesticCat) cat;       // ####
    ....
}
```

２番目に、 == が成立しない可能性があります。

```
Cat cat = (Cat) session.load(Cat.class, id);            // Cat########
DomesticCat dc =
    (DomesticCat) session.load(DomesticCat.class, id);  // ###DomesticCat###!
System.out.println(cat==dc);                            // false
```

しかし、これは見かけほど悪い状況というわけではありません。以下のように、たとえ異なったプロキシ・オブジェクトに対して二つの参照があったとしても、基となるインスタンスは同じオブジェクトです：

```
cat.setWeight(11.0);  // ##############db#######
System.out.println( dc.getWeight() );  // 11.0
```

3番目に、finalクラスやfinalメソッドを持つクラスにCGLIBプロキシを使えません。

最後に、もし永続オブジェクトにインスタンス化時にリソースが必要となるなら(例えば、イニシャライザやデフォル・トコンストラクタ内で)、そのリソースもまたプロキシを通して取得されます。実際は、そのプロキシクラスは永続クラスのサブクラスです。

これらの問題はJavaの単一継承モデルの原理上の制限のためです。もしこれらの問題を避けたいのなら、永続クラスひとつひとつに、ビジネス・メソッドを定義したインターフェイスを実装させなければなりません。そしてこれらのインターフェイスをマッピング定義ファイルでプロキシとして指定します。：

```
<class name="eg.Cat" proxy="eg.ICat">
    ......
    <subclass name="eg.DomesticCat" proxy="eg.IDomesticCat">
        .....
    </subclass>
</class>
```

CatはインターフェイスICatを、DomesticCatはインターフェイスIDomesticCatを実装します。その際、CatとDomesticCatのインスタンスはload() や iterate()によって返されます(find()はプロキシを返さないことに注意してください)。

```
ICat cat = (ICat) session.load(Cat.class, catid);
Iterator iter = session.iterate("from cat in class eg.Cat where cat.name='fritz'");
ICat fritz = (ICat) iter.next();
```

関係もまたlazyに初期化されます。これはCatではなく ICat型でプロパティを定義しなければならないということです。

以下のようなプロキシの初期化を必要としない操作も存在します。

- 永続クラスがequals()をオーバーライドしないときのequals()。
- 永続クラスがhashCode()をオーバーライドしないときのhashCode()
- 識別子のgetterメソッド

Hibernateはequals() や hashCode()をオーバーライドする永続クラスを検知します。

プロキシを初期化するときに発生する例外はLazyInitializationExceptionでラップされます

Sessionをクローズする前に、確実にプロキシやコレクションを初期化しなければならないときがあります。もちろん、例えばcat.getSex() や cat.getKittens().size()をコールして、強制的に初期化することは常に可能です。しかしこれはソースを読む人を混乱させますし、汎用的なコードの観点

からも不便です。staticメソッドである`Hibernate.initialize()` と `Hibernate.isInitialized()`を使うと、アプリケーションでlazyに初期化されたコレクションやプロキシを便利に使用できます。`Hibernate.initialize(cat)`は`Session`がオープンである限り、`cat`プロキシを強制的に初期化します。`Hibernate.initialize( cat.getKittens() )`はkittenコレクションに対して同様の効果を持ちます。

14.3. バッチ・フェッチングの使用

Hibernateではバッチ・フェッチングを効率的に使用できます。すなわち、1つのプロキシがアクセスされる場合、Hibernateはいくつかの初期化されていないプロキシをロードすることができます。バッチ・フェッチングはレイジー・ローディング戦略に対する最適化です。バッチ・フェッチングの調整にはクラス・レベルとコレクション・レベルの2つの方法があります。

クラス/エンティティに対するバッチ・フェッチングは理解しやすいです。実行時に以下の状況にあると創造してください。：`Session`に25の`Cat`インスタンスがロードされていて、それぞれの`Cat`は`Person`型である`owner`への参照を持っています。`Person`クラスはプロキシによりマッピングされており、`lazy="true"`です。もしすべてのcatsをイテレートしそれぞれ`getOwner()`をコールするならば、Hibernateはデフォルトではプロキシのownersを読み出すために、25の`SELECT`文を実行します。`Person`のマッピング定義の`batch-size`を指定することで、この振る舞いを調整することができます。：

```
<class name="Person" lazy="true" batch-size="10">...</class>
```

Hibernateはパターンが10つ，10つ，5つであるクエリを3つだけ実行します。パフォーマンスの最適化に関する限り、バッチ・フェッチングは当て推量であり、それは特定の`Session`中での初期化されていないプロキシの数に依存します。

コレクションのバッチ・フェッチングも可能です。例えば各`Person`が`Cat`のlazyコレクションを持ち、現在10の`Person`インスタンスが`Session`にロードされています。すべての`Person`インスタンスをイテレートすることで、すべての`getCats()`コールにつき1つ、計10の`SELECT`が発生します。もし`Person`のマッピング定義で`cats`コレクションに対してバッチ・フェッチングの指定が可能ならば、Hibernateはコレクションのプリ・フェッチが可能です。

```
<class name="Person">
    <set name="cats" lazy="true" batch-size="3">
        ...
    </set>
</class>
```

`batch-size`が3なので、Hibernateは4回の`SELECT`文で3つ，3つ，3つ，1つのコレクションをそれぞれロードします。属性の値は特定の`Session`中で期待される初期化されていないコレクションの数に依存します。

コレクションのバッチ・フェッチングは、アイテムのネストツリーの場合、つまり典型的なbill-of-materialsパターンの場合に有用です。

14.4. 第2レベル・キャッシュ

Hibernateの`Session`は永続データのトランザクションレベルのキャッシュです。class-by-classとcollection-by-collectionごとの、クラスタレベルやJVMレベル（`SessionFactory`レベル）のキャッシュを設定することも可能です。クラスタ化されたキャッシュにつなぐことさえ可能です。キャッ

シュは他のアプリケーションによって永続ストアに加えられた変更は考慮しないことに注意してください(キャッシュデータを定期的に期限切れに設定することはできます)。

デフォルトでは、HibernateはJVMレベルキャッシュにEHCacheを使います(JCSサポートは現在推奨されておらず、Hibernateの将来のバージョンで削除されます)。hibernate.cache.provider_classプロパティを使い、net.sf.hibernate.cache.CacheProviderを実装したクラスの名前を指定することで他の実装を選ぶことができます。

表 14.1. キャッシュ・プロバイダ

キャッシュ	プロバイダクラス	タイプ	クラスタセーフ	クエリキャッシュサポート
Hashtable (製品用として意図されていません)	net.sf.hibernate.cache.HashtableCacheProvider	メモリ		yes
EHCache	net.sf.hibernate.cache.EhCacheProvider	メモリ、ディスク		yes
OSCache	net.sf.hibernate.cache.OSCacheProvider	メモリ、ディスク		yes
SwarmCache	net.sf.hibernate.cache.SwarmCacheProvider	クラスタ(IPマルチキャスト)	yes (clustered invalidation)	
JBoss TreeCache	net.sf.hibernate.cache.TreeCacheProvider	クラスタ(IPマルチキャスト), トランザクション	yes (複製)	yes (clock sync req.)

14.4.1. キャッシュのマッピング

クラスまたはコレクションのマッピングの<cache>要素は以下の形式です：

```
<cache
    usage="transactional|read-write|nonstrict-read-write|read-only"  (1)
/>
```

(1) usage はキャッシュ戦略を指定します： transactional, read-write, nonstrict-read-write または read-onlyです

または(より良い方法として?)、<class-cache>と<collection-cache>要素をhibernate.cfg.xmlの中で指定することができます。

usage属性はcache concurrency strategyを指定します。

14.4.2. 戦略: read only

もしアプリケーションが、読み込みはするが決して永続クラスのインスタンスを修正しないならば、read-onlyキャッシュを使うことができます。これは最も単純で最もパフォーマンスの良い戦略です。クラスタでの使用も完全に安全です。

```
<class name="eg.Immutable" mutable="false">
    <cache usage="read-only"/>
    ....
</class>
```

14.4.3. 戦略: read/write

もしアプリケーションがデータを更新する必要があるなら、read-writeキャッシュが適切かもしれません。このキャッシュ戦略は、もしserializable transaction isolationレベルが要求されるなら、決して使うべきではありません。 もしJTA環境でキャッシュが使われるなら、JTAのTransactionManagerを取得する戦略を示すhibernate.transaction.manager_lookup_classプロパティを指定しなければなりません。他の環境ではSession.close() や Session.disconnect()がコールされたとき、確実にトランザクションが完了していなければなりません。もしクラスタでこの戦略を使いたいのなら、基となるキャッシュの実装が確実にロックをサポートしていなければなりません。組み込みのキャッシュ・プロバイダはこれをサポートしません。

```
<class name="eg.Cat" .... >
    <cache usage="read-write"/>
    ....
    <set name="kittens" ... >
        <cache usage="read-write"/>
        ....
    </set>
</class>
```

14.4.4. 戦略: nonstrict read/write

もしアプリケーションがたまにしかデータの更新を必要とせず(つまり、2つのトランザクションが同時に同じitemを更新するようなことがほとんど起きない)、厳密なトランザクションの隔離が要求されないならば、nonstrict-read-writeキャッシュが適当かもしれません。もしキャッシュがJTA環境で使われるなら、hibernate.transaction.manager_lookup_classを指定しなければなりません。他の環境ではSession.close() や Session.disconnect()がコールされたとき、確実にトランザクションが完了していなければなりません。

14.4.5. 戦略: transactional

transactionalキャッシュ戦略はJBossのTreeCacheのような、完全にトランザクショナルなキャッシュのサポートを 提供します。そのようなキャッシュはJTA環境だけで使用され、hibernate.transaction.manager_lookup_classを指定しなければなりません。

すべての同時並行性キャッシュ戦略をサポートしているキャッシュ・プロバイダはありません。以下のテーブルはどのプロバイダがどの同時並行性戦略と対応するかを示しています。

表 14.2. 同時並行性キャッシュ戦略のサポート

Cache	read-only	nonstrict-read-write	read-write	transactional
Hashtable（製品用として意図されていません）	yes	yes	yes	
EHCache	yes	yes	yes	
OSCache	yes	yes	yes	
SwarmCache	yes	yes		
JBoss TreeCache	yes			yes

14.5. Sessionキャッシュの取り扱い

オブジェクトをsave(), update() や saveOrUpdate()に渡すとき、そしてload(), find(), iterate(), や filter()を使ってオブジェクトを取り出すときはいつでも、そのオブジェクトはSessionの内部キャッシュに追加されます。続いてflush()がコールされるとき、そのオブジェクトの状態はデータベースと同期されます。 もしこの同期を望まなかったり、膨大な数のオブジェクトを処理し効率的にメモリを扱う必要があるなら、evict()メソッドを使ってキャッシュからオブジェクトやそのコレクションを削除することができます。

```
Iterator cats = sess.iterate("from eg.Cat as cat"); //##########
while ( cats.hasNext() ) {
    Cat cat = (Cat) iter.next();
    doSomethingWithACat(cat);
    sess.evict(cat);
}
```

もし関連がcascade="all" や cascade="all-delete-orphan"でマッピングされているならば、Hibernateは関連するエンティティを自動的にキャッシュから削除します。

Sessionには、インスタンスがセッション･キャッシュに含まれるかどうかを判断するためのcontains()メソッドもあります。

セッション･キャッシュからすべてのオブジェクトを完全に取り除くためには、Session.clear()をコールしてください。

第2レベルのキャッシュのために、SessionFactoryにはインスタンス、クラス全体、コレクション･インスタンスまたはコレクション･ロール全体をキャッシュから削除するメソッドが定義されています。

14.6. クエリ･キャッシュ

クエリ･リザルトセットもキャッシュすることができます。これは同じパラメータで度々実行されるクエリについてのみ役立ちます。 クエリキャッシュを使うためには、まずプロパティhibernate.cache.use_query_cache=trueを設定して使用可能にしておかなければなりません。これにより2つのキャッシュ領域が作成されます。1つはキャッシュされたクエリ･リザルトセットを保持し（net.sf.hibernate.cache.QueryCache）、もう1つは最新のクエリ･テーブルへの更新のタイムスタンプを保持するもの（net.sf.hibernate.cache.UpdateTimestampsCache）です。クエリ･キャッシュは

リザルトセットの中のどんなエンティティの状態もキャッシュしないことに注意してください。それは識別子 の値と、バリュー・タイプの結果のみキャッシュします。そのためクエリ・キャッシュは通常、第2レベルのキャッシュと一緒に使われます。

ほとんどのクエリはキャッシュしても意味がないので、デフォルトではクエリはキャッシュされません。キャッシュを有効にするには、`Query.setCacheable(true)`をコールしてください。そうすればクエリが既存のキャッシュ結果を見つけ、実行時にその結果をキャッシュに追加するようになります。

もしクエリ・キャッシュ終了方針の粒度の良い制御を必要とするなら、`Query.setCacheRegion()`をコールすることで 特定のクエリに対するキャッシュ領域を指定することができます。

```
List blogs = sess.createQuery("from Blog blog where blog.blogger = :blogger")
    .setEntity("blogger", blogger)
    .setMaxResults(15)
    .setCacheable(true)
    .setCacheRegion("frontpages")
    .list();
```

もしクエリがそのクエリキャッシュの範囲をリフレッシュさせるべきなら、`Query.setForceCacheRefresh()`をコールして`true`にすることができます。この操作は特にベースとなっているデータが別のプロセスによって更新されるケース(つまりHibernateを通した修正では無いケース)のときに有用であり、アプリケーションがこれらのイベントの知識をベースにして選択的にクエリキャッシュの範囲をリフレッシュできるようになります。これはクエリキャッシュ領域のevictの代替方法です。もしクエリに対してきめの細かいリフレッシュ制御が必要ならば、個々のクエリに新しい領域割り当てル代わりにこの機能を使ってください。

第15章 ツールセット・ガイド

Hibernateを使ったラウンドトリップ・エンジニアリングはHibernateプロジェクトの一部として保守されているコマンドライン・ツールのセットを使うことで可能で、XDoclet、Middlegen、AndroMDAがHibernateサポートに組み込まれています。

Hibernateメインパッケージは最も重要なツールになっています(実行時にHibernateの「中」からも使われます)。:

- マッピングファイルからDDLスキーマを生成(`SchemaExport`, `hbm2ddl`)

Hibernateプロジェクトにより直接提供された他のツール群は、Hibernate Extensionsという別のパッケージになっています。このパッケージは以下のようなタスクのためのツールを含んでいます。:

- マッピングファイルからJavaソースファイルを生成(`CodeGenerator`, `hbm2java`)
- コンパイル済みJavaクラス、またはXDocletのマークアップを施されたJavaソースからマッピングファイルを生成(`MapGenerator`, `class2hbm`)

実はHibernate Extentionsにはもうひとつ、実用的なツール`ddl2hbm`がありますが、非推奨であり、もはや保守されていません。Middlegenの方が同じタスクに対してより良い仕事を行ないます。

以下にHibernateをサポートするサード・パーティ・ツールを挙げます。:

- Middlegen (既存のデータベース・スキーマからマッピング・ファイルを生成)
- AndroMDA (UML図とそのXML/XMI表現から永続クラスのコードを生成するMDA(モデル駆動アーキテクチャ)アプローチ)

これらサード・パーティ・ツールはこのリファレンスには文書化されていません。最新の情報はHibernateの ウェブサイトを参照してください。(サイトのスナップショットはHibernateメイン・パッケージに含まれています。

15.1. スキーマ生成

DDLはコマンドライン・ユーティリティによりマッピング・ファイルから生成することができます。バッチファイルはHibernateのコア・パッケージの`hibernate-x.x.x/bin`ディレクトリにあります。

生成されるスキーマはエンティティとコレクション・テーブルへの参照完全性制約(主キーと外部キー)を含みます。また、マッピングする識別子ジェネレータに対しテーブルとシーケンスが作成されます。

このツールを使うときは、`hibernate.dialect`プロパティでSQLの##を指定しなければなりません。

15.1.1. スキーマのカスタマイズ

多くのHibernateのマッピング要素ではオプションの`length`という名の属性を定義しています。この属性でカラム長を設定することができます(または数値/小数型のデータの精度を設定できます)。

not-null属性（テーブルのカラムへのNOT NULL制約を生成する）とunique属性（テーブルのカラムへのUNIQUE制約を生成する）が設定できるタグもあります。

カラムのインデックスの名前を特定するためのindex属性を指定できるタグもあります。unique-key属性はカラムをシングル・ユニットのキー制約でグループ化するために使われます。現在、unique-key属性で指定された値は制約の命名には使われず、マッピング・ファイルでカラムをグループ化することにのみ使われます。

例:

```
<property name="foo" type="string" length="64" not-null="true"/>

<many-to-one name="bar" foreign-key="fk_foo_bar" not-null="true"/>

<element column="serial_number" type="long" not-null="true" unique="true"/>
```

もうひとつの方法として、これらの要素は子の<column>要素も持つことができます。これは特にマルチ・カラム型に対して有効です。

```
<property name="foo" type="string">
    <column name="foo" length="64" not-null="true" sql-type="text"/>
</property>

<property name="bar" type="my.customtypes.MultiColumnType"/>
    <column name="fee" not-null="true" index="bar_idx"/>
    <column name="fi" not-null="true" index="bar_idx"/>
    <column name="fo" not-null="true" index="bar_idx"/>
</property>
```

sql-type属性で、デフォルトのHibernate型のマッピングをSQLのデータ型にすることができます。

check属性でチェック制約を指定することができます。

```
<property name="foo" type="integer">
    <column name="foo" check="foo > 10"/>
</property>

<class name="Foo" table="foos" check="bar < 100.0">
    ...
    <property name="bar" type="float"/>
</class>
```

表 15.1. まとめ

まとめ	値	説明
length	number	カラム長/小数の精度
not-null	true|false	カラムがnull値を取らないことを指定す
unique	true|false	カラムがユニーク制約を持つことを指定する
index	index_name	(マルチ・カラム)インデックスの名前を指定する
unique-key	unique_key_name	マルチ・カラムのユニーク制約の名前を指定する
foreign-key	foreign_key_name	<one-to-one>、<many-to-one>、<many-to-many>マッピング要素を使って、関連に対し生成された外部キー制約の名前を指定する。inverse="true"側は

まとめ	値	説明
		SchemaExportにより考慮されないことに注意してください。
sql-type	column_type	デフォルトのカラム型をオーバーライドする(<column>要素の属性のみ)
check	SQL expression	カラムやテーブルにSQLのチェック制約を作成する

15.1.2. ツールの実行

SchemaExportは標準出力に対してDDLスクリプトを書き出し、またはDDL文を実行します。

java -cp hibernate_classpaths net.sf.hibernate.tool.hbm2ddl.SchemaExport options mapping_files

表 15.2. **SchemaExport**のコマンドライン･オプション

オプション	説明
--quiet	スクリプトを標準出力に出力しません
--drop	テーブルを削除するだけです
--text	データベースにエクスポートしません
--output=my_schema.ddl	DDLスクリプトをファイルに出力します
--config=hibernate.cfg.xml	XMLファイルからHibernateの定義フィファイルを読み込みます
--properties=hibernate.properties	ファイルからデータベースプロパティを読み込みます
--format	スクリプト用に生成されたSQLをフォーマットします
--delimiter=x	スクリプトの行区切り文字を設定します

アプリケーションにSchemaExportを組み込むこともできます：

```
Configuration cfg = ....;
new SchemaExport(cfg).create(false, true);
```

15.1.3. プロパティ

データベースのプロパティを指定することができます。

- システム・プロパティとして-D<property>
- hibernate.properties内で指定する
- --propertiesで指定されたファイル内で指定する

必要なプロパティは以下のようになります。：

表 15.3. SchemaExportのコネクション・プロパティ

プロパティ名	説明
hibernate.connection.driver_class	JDBCドライバーのクラス
hibernate.connection.url	JDBCのURL
hibernate.connection.username	データベースのユーザ
hibernate.connection.password	ユーザのパスワード
hibernate.dialect	方言

15.1.4. Antの使用

Antのビルド・スクリプトからSchemaExportを呼ぶことができます。:

```
<target name="schemaexport">
    <taskdef name="schemaexport"
        classname="net.sf.hibernate.tool.hbm2ddl.SchemaExportTask"
        classpathref="class.path"/>

    <schemaexport
        properties="hibernate.properties"
        quiet="no"
        text="no"
        drop="no"
        delimiter=";"
        output="schema-export.sql">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaexport>
</target>
```

もし#####ファイルや#####ファイルを指定しなければ、SchemaExportTaskは通常のなAntプロジェクトのプロパティを代わりに使用します。言いかえれば、外部の設定ファイルやプロパティ・ファイルを望まないし必要としなければ、build.xmlあるいはbuild.propertiesの設定プロパティをhibernate.*としてもよい。

15.1.5. インクリメンタルなスキーマ更新

SchemaUpdateツールは既存のスキーマをインクリメンタルに更新します。SchemaUpdateはJDBCメタデータAPIに強く依存します。そのためすべての、JDBCドライバでうまくいくとは限らないことに注意してください。

java -cp hibernate_classpaths net.sf.hibernate.tool.hbm2ddl.SchemaUpdate options mapping_files

表 15.4. **SchemaUpdate**のコマンドライン・オプション

オプション	説明
--quiet	標準出力にスクリプトを出力しません

オプション	説明
--properties=hibernate.properties	ファイルからデータベース･プロパティを読み込みます

アプリケーションにSchemaUpdateを組み込むことができます。 :

```
Configuration cfg = ....;
new SchemaUpdate(cfg).execute(false);
```

15.1.6. インクリメンタルなスキーマ更新に対するAntの使用

AntスクリプトからSchemaUpdateをコールすることができます :

```
<target name="schemaupdate">
    <taskdef name="schemaupdate"
        classname="net.sf.hibernate.tool.hbm2ddl.SchemaUpdateTask"
        classpathref="class.path"/>

    <schemaupdate
        properties="hibernate.properties"
        quiet="no">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaupdate>
</target>
```

15.2. コード生成

Hibernateのコードジェネレータを使い、Hibernateのマッピングファイルから Javaでの実装クラスのスケルトンを生成することができます。このツールはHibernate Extensionパッケージに含まれています。(別途ダウンロード)

hbm2javaはマッピング･ファイルを構文解析し、十分に機能するJavaソースファイルを生成します。このように hbm2javaを使い、単に.hbmファイルを作成するだけで、Javaファイルをハンドコードする煩わしさから解放されます。

java -cp hibernate_classpaths net.sf.hibernate.tool.hbm2java.CodeGenerator options mapping_files

表 15.5. コードジェネレータのコマンドライン･オプション

オプション	説明
--output=output_dir	生成されるコードのルートディレクトリ
--config=config_file	hbm2javaを設定するオプションのファイル

15.2.1. 設定ファイル(オプション)

設定ファイルはソースコードの複数の「レンダラ」を指定する方法と、「グローバル」スコープの<meta>属性を宣言する方法を提供します。これについての詳細は<meta>属性の節を見てください。

```
<codegen>
    <meta attribute="implements">codegen.test.IAuditable</meta>
    <generate renderer="net.sf.hibernate.tool.hbm2java.BasicRenderer"/>
    <generate
        package="autofinders.only"
        suffix="Finder"
        renderer="net.sf.hibernate.tool.hbm2java.FinderRenderer"/>
</codegen>
```

この構成ファイルはグローバル・メタ属性の"implements"を宣言し、デフォルトのレンダラ（BasicRenderer）と ファインダを生成するレンダラの二つを指定しています(詳しくは次の「BasicFinder生成」をご覧ください)。

２番目のレンダラはパッケージ属性とサフィックス属性とともに提供されます。

パッケージ属性では、.hbmファイル内で指定されたパッケージスコープの代わりに、このレンダラから生成されたソースファイルの配置場所を指定します。

サフィックス属性では、生成されるファイルのサフィックスを指定します。例えばFoo.javaというファイルを、代わりにFooFinder.javaとすることができます。

<generate>要素に<param>属性を付け加えることで、任意のパラメータを(注：)投げることも可能です。

hbm2java は現在 generate-concrete-empty-classes パラメータをサポートします。これはBasicRendererに対し、ベースクラスを拡張する、空の具象クラスだけを生成するという情報を与えます。以下のonfig.xmlの例はこの特徴を示します。

```
<codegen>
  <generate prefix="Base" renderer="net.sf.hibernate.tool.hbm2java.BasicRenderer"/>
  <generate renderer="net.sf.hibernate.tool.hbm2java.BasicRenderer">
    <param name="generate-concrete-empty-classes">true</param>
    <param name="baseclass-prefix">Base</param>
  </generate>
</codegen>
```

このconfig.xmlファイルは2つのレンダラを設定することに注意してください。一つは基底クラスを生成するもので、もう一つは空の具象クラスを生成するものです。

15.2.2. ##属性

<meta>タグはhbm.xmlに情報を付加する簡単な方法です。これによりツールがHibernateのコアに直接関係しない情報を保存し、読みこむための自然な場所を保持します。

"protected"なセッタだけを生成する、あるインタフェースの集合を常に実装する、またはある基底クラスを常に拡張するといった事をhbm2javaに知らせるために<meta>タグを使用することができます。:

下の例は以下のようなコードを生成します(わかりやすいようコードは短くしてあります）。

```
<class name="Person">
    <meta attribute="class-description">
        Javadoc for the Person class
        @author Frodo
    </meta>
    <meta attribute="implements">IAuditable</meta>
```

```
    <id name="id" type="long">
        <meta attribute="scope-set">protected</meta>
        <generator class="increment"/>
    </id>
    <property name="name" type="string">
        <meta attribute="field-description">The name of the person</meta>
    </property>
</class>
```

Javadocコメントとprotectedなセット・メソッドに注目してください。：

```
// default package

import java.io.Serializable;
import org.apache.commons.lang.builder.EqualsBuilder;
import org.apache.commons.lang.builder.HashCodeBuilder;
import org.apache.commons.lang.builder.ToStringBuilder;

/**
 *         Javadoc for the Person class
 *         @author Frodo
 *
 */
public class Person implements Serializable, IAuditable {

    /** identifier field */
    public Long id;

    /** nullable persistent field */
    public String name;

    /** full constructor */
    public Person(java.lang.String name) {
        this.name = name;
    }

    /** default constructor */
    public Person() {
    }

    public java.lang.Long getId() {
        return this.id;
    }

    protected void setId(java.lang.Long id) {
        this.id = id;
    }

    /**
     * The name of the person
     */
    public java.lang.String getName() {
        return this.name;
    }

    public void setName(java.lang.String name) {
        this.name = name;
    }

}
```

表 15.6. サポートしているメタ・タグ

属性	説明
class-description	クラスのJavadocとして挿入されます

属性	説明
field-description	フィールド・プロパティのJavadocとして挿入されます
interface	trueなら、クラスの代わりにインターフェイスが生成されます
implements	クラスが実装すべきインターフェイス
extends	クラスが拡張すべきクラス（サブクラスには無視されます)
generated-class	生成される実際のクラスの名前を上書きします
scope-class	クラスのスコープ
scope-set	セッタメソッドのスコープ
scope-get	ゲッタメソッドのスコープ
scope-field	実際のフィールドのスコープ
use-in-tostring	toString()にこのプロパティを含めます
implement-equals	このクラスにequals() と hashCode()メソッドを含めます
use-in-equals	equals() と hashCode()メソッド内にこのプロパティを含めます
bound	プロパティにpropertyChangeListenerサポートを追加します
constrained	プロパティへのbound + vetoChangeListenerサポート
gen-property	falseならプロパティが生成されません(注意して使いましょう)
property-type	プロパティのデフォルトの型をオーバーライドします。単なるObjectクラスの代わりに 具体的な型を指定するタグとともに使ってください。
class-code	クラスの最後に挿入する特別なコード
extra-import	すべてのimportの後に挿入する特別なimport
finder-method	以下の｢基本ファインダ･ジェネレータ｣を見てください
session-method	以下の｢基本ファインダ･ジェネレータ｣を見てください

<meta>を通して定義された属性はhbm.xmlファイル内のデフォルトの"inherited"ごとにあります。

これはどういうことでしょうか？これは例えば、もしすべてのクラスでIAuditableを実装したいならばhbm.xmlファイルの先頭で、<hibernate-mapping>のすぐ後に、<meta attribute="implements">IAuditable</meta>を追加するだけでいいということです。これで、そのhbm.xmlファイルで定義されているすべてのクラスはIAuditableを実装することになります！（ただし、そのクラスが"implements"メタ属性を持っているときは除きます。というのはローカルで指定されたメタタグは継承されたメタタグを無効にし、置き換えるからです。）

注意：これはすべての <meta>タグに当てはまります。そのため例えば、デフォルトであるprivateの代わりにすべてのフィールドをprotectedと指定するなどに使うことができます。このことは、たとえば<class>タグのすぐ後に<meta attribute="scope-field">protected</meta>を追加することで、クラスのすべてのフィールドがプロテクティドになります。

継承された<meta>タグを無効にするには、単に属性をinherit="false"と指定するればよく、例えば<meta attribute="scope-class" inherit="false">public abstract</meta>は"クラススコープ"を現在のクラスに制限し、サブクラスには適用されません。

15.2.3. BasicFinderジェネレータ

今ではHibernateプロパティに対しhbm2javaを使って、BasicFinderを生成させることが可能になりました。これによりhbm.xmlファイルに二つのことが必要になります。

一つめは、どのフィールドに対してファインダを生成したいのか指定する必要があります。以下のようにプロパティ・タグ中のメタ・ブロックを使って指定します。：

```
<property name="name" column="name" type="string">
     <meta attribute="finder-method">findByName</meta>
</property>
```

ファインダ・メソッドの名前はメタ・タグで囲まれたテキストになります。

二つめは、以下のフォーマットのようなhbm2java用の構成ファイルを作成する必要があります。

```
<codegen>
    <generate renderer="net.sf.hibernate.tool.hbm2java.BasicRenderer"/>
    <generate suffix="Finder" renderer="net.sf.hibernate.tool.hbm2java.FinderRenderer"/>
</codegen>
```

そしてhbm2java --config=xxx.xmlパラメータを使います。ただしxxx.xmlは作成した構成ファイルです。

オプションのパラメータはクラスレベルにおいてのメタタグで、以下のようなフォーマットです：

```
<meta attribute="session-method">
    com.whatever.SessionTable.getSessionTable().getSession();
</meta>
```

もしThread Local Sessionパターンを使うなら、Sessionを得る方法を指定します。(Hibernateのウェブサイトのデザインパターン・エリアに文書化されています)。

15.2.4. Velocityベースのレンダラ/ジェネレータ

今では、代替のレンダリング・メカニズムとしてvelocityが利用可能になりました。以下のconfig.xmlは、velocityレンダラを使うためにはhbm2javaをどのように設定すればよいかを示しています。

```
    <codegen>
     <generate renderer="net.sf.hibernate.tool.hbm2java.VelocityRenderer">
      <param name="template">pojo.vm</param>
     </generate>
    </codegen>
```

templateという名前のパラメータは、使いたいvelocityのマクロ・ファイルへのリソース・パスです。このファイルはhbm2javaのクラスパス上に配置することで利用可能になります。このようにpojo.vmがあるディレクトリをantのタスクやシェルスクリプトに追加するのを忘れないようにしてください(デフォルトのロケーションは./tools/src/velocityです)。

カレントのpojo.vmがJavaBeansの最も基本的な部分しか生成しないことに注意してください。Velocityベースのレンダラ/ジェネレータはデフォルトのレンダラほど完全でも特徴豊かではありません。基本的に多くの##タグはサポートされていません。

15.3. マッピング・ファイルの生成

マッピング・ファイルのスケルトンはMapGeneratorというコマンドライン・ユーティリティを使ってコンパイルされた永続クラスから生成できます。このユーティリティはHibernate Extensionパッケージの一部です。

Hibernateのマッピング・ジェネレータはコンパイルされたクラスからマッピングを生成するメカニズムを提供します。これにはプロパティを見つけるのにJavaのリフレクションを使い、プロパティの型から適切なマッピングを推測するために経験則を用いています。生成されたマッピングは開始点としてのみ意図されています。ユーザからの追加入力なしでは、完全なHibernateマッピングを生成することはできません。しかしながら、このツールはマッピングを生成する"退屈な"繰り返しをいくらか減らします。

クラスは一度に1つのマッピングに追加されます。このツールはHibernate persistableではないと判断したクラスを無視します。

クラスをHibernate persistableにするには、以下のようにします。

- プリミティブ型ではいけません
- 配列であってはいけません
- インターフェイスであってはいけません
- 入れ子クラスではいけません
- デフォルトコンストラクタ(引数なし)を持たなければなりません。

実際にはインタフェースと入れ子クラスはHibernateで永続化できることに注意してください。しかしこれは普通ユーザが永続化したいとは思わないものです。

MapGeneratorは、同じデータベースのテーブルにできるだけ多くのHiberante Persistableなスーパークラスを追加するよう意図された、すべてのクラス階層をたどっていきます。検索はプロパティがcandidate UID namesのリストに名前が見つかるとすぐに終了します。

デフォルトのcandidate UID プロパティ名のリストは： uid, UID, id, ID, key, KEY, pk, PKです。

クラス内の2つのメソッド、セッタとゲッタが、セッタの第一引数が引数なしのゲッタの戻り値と同じであり、セッタの戻り値がvoidである場合、プロパティは発見されます。さらには、セッタの名前はsetで始まらなければならず、ゲッタの名前はgetか、プロパティの型がbooleanのときはisで始まらなければなりません。どちらの場合も、名前の残りの部分は一致しなければなりません。もし二文字めが小文字なら、プロパティ名の頭文字が小文字になることを除けば、この一致する部分はプロパティ名になります。

それぞれのプロパティのデータベースの型を決定する規則はこのようになります：

1. もしJavaの型が`Hibernate.basic()`ならば、プロパティはその型のカラムです。
2. `hibernate.type.Type` カスタム型と `PersistentEnum`型についても同様に、単純なカラムが使われます。
3. プロパティの型が配列なら、Hibernateの配列が使われ`MapGenerator`は配列の要素の型をリフレクトするよう試みます。
4. もしプロパティが`java.util.List`, `java.util.Map`, or `java.util.Set`の型を持つならば、対応するHibernate型が使われますが、`MapGenerator`は決してこれらの型の内部は処理しません。
5. もしプロパティの型が他のクラスなら、`MapGenerator`はすべてのクラスが実行されるまでデータベースの表現の決定保留します。この時点で、もし上述のようなスーパークラスの検索を通してクラスが発見されるとすれば、プロパティは`many-to-one`関連です。もしクラスがどれかプロパティを持つなら、それは`component`としてマッピングします。さもなければ、それはシリアライズ可能でも永続化可能でもありません。

15.3.1. ツールの実行

ツールはXMLマッピングを標準出力と/またはファイルに書き込みます。

ツールを起動するとき、コンパイルされたクラスをクラスパスに通しておかなければなりません。

`java -cp` hibernate_and_your_class_classpaths `net.sf.hibernate.tool.class2hbm.MapGenerator` options and classnames

コマンドラインまたはインタラクティブの二つの操作モードがあります。

インタラクティブモードはコマンドラインの引数`--interact`を一つ指定することで選択されます。このモードはプロンプトの応答をコンソールに出力します。それを使って、`XXX`がUIDプロパティのとき`uid=XXX`コマンドを使ってそれぞれのクラスにUIDのプロパティ名を設定することができます。他の代替コマンドは単に完全限定クラス名で、XMLを生成するコマンドや終了するコマンドがあります。

コマンドライン・モードにおいて、引き数は処理されるクラスの完全限定クラス名が組み込まれているオプションです。オプションのほとんどは複数回使われるものと意図されていて、どの使い方も続くクラスに影響を及ぼします。

表 15.7. MapGeneratorのコマンドライン・オプション

オプション	説明
`--quiet`	標準出力にO-Rマッピングを出力しません
`--setUID=uid`	UIDの候補とするリストをひとつのUIDに設定します
`--addUID=uid`	候補のUIDのリストの先頭にUIDを追加します
`--select=mode`	続いて追加されたクラスに対して、選択モードmode(例えば, distinct または all)を使うモード
`--depth=<small-int>`	続いて追加されたクラス対して、コンポーネント・データの再帰の深さを制限します

オプション	説明
--output=my_mapping.xml	O-R Mappingをファイルに出力します
full.class.Name	クラスをマッピングに追加します
--abstract=full.class.Name	以下を見てください

抽象スイッチはマップ・ジェネレータツールが特定のスーパークラスを無視するようにします。 それは共通の継承を持つクラスが一つの大きなテーブルにマッピングされないようにするためです。例えば、これらのクラスの階層構造を考えてみてください：

```
Animal-->Mammal-->Human

Animal-->Mammal-->Marsupial-->Kangaroo
```

もし--abstractスイッチが使われなければ、すべてのクラスはAnimalのサブクラスとしてマッピングされます。 その結果、どのサブクラスが実際に格納されたのかを示すディスクリミネータ・カラムと、すべてのクラスのすべてのプロパティを含んだ一つの大きなテーブルが付け加えることになります。もし MammalがabstractとマークされるならHuman と Marsupialは別の<class>定義にマッピングされ、別のテーブルに格納されます。Marsupialがaabstracttとしてマークされなければ、KangarooはMarsupialのサブクラスになってしまいます。

第16章 例: 親/子関係

新規ユーザがHibernateを使いまず最初に扱うモデルの一つは、親子型の関係です。これには異なった2つのアプローチが存在します。とりわけ新規ユーザにとって、さまざまな理由から最も便利だと思われるアプローチは、#から#への<one-to-many> 関連により#と#の両方をエンティティクラスとしてモデリングする方法です。（もう一つの方法は、#を<composite-element>として定義するものです。）これまでで、(Hibernateでは)one-to-many関連のデフォルトの意味が、通常複合要素のマッピングよりも親子関係の意味にかなり近いことがわかりました。 それでは親子関係を効率的かつエレガントにモデリングするために、どのようにしてカスケード操作を使った双方向one-to-many関連を扱うのか説明します。それはまったく難しいものではありません！

16.1. コレクションについての注意

Hibernateのコレクションは所持するエンティティの論理的な部分と考えられ、それは所持されるエンティティのでは決してありません。これは致命的な違いです！以下のような結果になります：

- オブジェクトをコレクションから削除、またはコレクションに追加するとき、コレクションのオーナのバージョン番号はインクリメントされます。
- もしコレクションから削除されるオブジェクトがバリュー･タイプのインスタンス(例えばコンポジット･エレメント)だったならば、そのオブジェクトは永続化されず、その状態はデータベースから完全に削除されます。同じように、バリュー･タイプのインスタンスをコレクションに追加すると、その状態は即座に永続化されます。
- その一方、もしエンティティがコレクション(one-to-manyまたはmany-to-many関連)から削除されても、デフォルトではそれは削除されません。この振る舞いは完全に首尾一貫しています。すなわち、他のエンティティの内部状態への変更によって、関連するエンティティが消滅すべきではないということです！同様に、エンティティがコレクションに追加されても、デフォルトではそのエンティティは永続化されません。

その代わりに、デフォルトの振る舞いでは、エンティティをコレクションに追加すると単に2つのエンティティ間のリンクを作成し、一方でエンティティを削除するとリンクを削除します。これはすべてのケースにおいて、非常に適切だと言えます。しかし子が親のライフサイクルに制限される親子関係のケースでは、全く適切ではありません。

16.2. 双方向one-to-many

ParentからChildへの単純な<one-to-many>関連から始めるとします。

```
<set name="children">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

もし以下のコードを実行すると、

```
Parent p = .....;
Child c = new Child();
p.getChildren().add(c);
```

```
session.save(c);
session.flush();
```

Hibernateは、2つのSQL文を発行します：

- cに対するレコードを生成するINSERT
- pからcへのリンクを作成するUPDATE

これは非効率的なだけではなく、parent_idカラムにおいてNOT NULL制約に違反します。

pからcへのリンク（外部キーparent_id)はChildオブジェクトの状態の一部とは考えられず、そのためINSERTによって外部キーは作成されないことが原因です。ですから、解決策はChildマッピングの一部にリンクすることです。

```
<many-to-one name="parent" column="parent_id" not-null="true"/>
```

(またChildクラスにparentプロパティを追加する必要があります。

さあそれではChildエンティティがリンクの状態をマッピングするようになったので、コレクションがリンクを更新しないようにしましょう。inverse属性を使います。

```
<set name="children" inverse="true">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

以下のコードを使えば、新しいChildを追加することができます。

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
c.setParent(p);
p.getChildren().add(c);
session.save(c);
session.flush();
```

そして今、たった一つのSQL INSERTだけが発行されるようになりました！

厳しくするには、ParentのaddChild()メソッドを作成します。

```
public void addChild(Child c) {
    c.setParent(this);
    children.add(c);
}
```

Childを追加するコードはこのようになります。

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
session.save(c);
session.flush();
```

16.3. ライフサイクルのカスケード

明示的にsave()をコールするのはまだ煩わしいものです。これをカスケードを使って対処します。

```
<set name="children" inverse="true" cascade="all">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

これは上のコードをこのように単純化します

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
session.flush();
```

同様にParentをセーブまたは削除するときに、子を一つ一つ取り出して扱う必要はありません。以下のコードはpを削除し、そしてデータベースからその子をすべて削除します。

```
Parent p = (Parent) session.load(Parent.class, pid);
session.delete(p);
session.flush();
```

しかし、次のコードはデータベースからcを削除しません。pへのリンクを削除する（そしてこのケースではNOT NULL制約違反を引き起こす）だけです。

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
c.setParent(null);
session.flush();
```

以下のように明示的に子をdelete()する必要があります。 delete() the Child.

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
session.delete(c);
session.flush();
```

さあ私たちのケースでは実際にChildが親なしでは存在できないようになりました。そのため、もしコレクションからChildを取り除けば、本当に削除したいものが取り除かれます。 このためにcascade="all-delete-orphan"を使わなければなりません。

```
<set name="children" inverse="true" cascade="all-delete-orphan">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

注意：コレクションのマッピングでinverse="true"と指定しても、コレクションの要素の繰り返しによってカスケードが依然実行されます。それゆえもしカスケードでオブジェクトをセーブ、削除、更新する必要があるなら、それをコレクションに追加しなければなりません。単にsetParent()をコールするだけでは不十分です。

16.4. カスケードupdate()の使用

ParentがあるSessionでロードされ、UIのアクションで変更が加えられ、この変更を新しいSessionにおいて永続化(update()をコールして)したいとします。Parentがのコレクションを持ち、カスケード

更新が有効になっているため、 Hibernateはどの子が新しくインスタンス化されたか、どれがデータベースに行として表現されているかということを 知る必要があります。 ParentとChildの両方がjava.lang.Long型の（人工的）識別プロパティを持つと仮定しましょう。Hibernateはどの子が新しいのかを決定するために識別プロパティの値を使います(versionやtimestampプロパティも使えます。項9.4.2. 「関連付けをやめたオブジェクトの更新」参照)。

unsaved-value属性は新しくインスタンス化されたインスタンスの識別子の値を特定するために使われます。unsaved-value のデフォルト値はnullで、これは Long識別子型にとって十分です。もしプリミティブ識別プロパティを使うならば、Childマッピングに以下を指定する必要があります。 :

```
<id name="id" type="long" unsaved-value="0">
```

(versionとtimestampプロパティ・マッピングのunsaved-value属性も存在します。)

以下のコードはparentとchildを更新し、newChild. を挿入します。

```
//parent#child#######Session##########n
parent.addChild(child);
Child newChild = new Child();
parent.addChild(newChild);
session.update(parent);
session.flush();
```

これらは生成された識別子の場合には非常に良いのですが、割り当てられた識別子と複合識別子の場合はどうでしょうか？これはunsaved-valueが、（ユーザにより割り当てられた識別子を持つ）新しくインスタンス化されたオブジェクトと前のSessionでロードされたオブジェクトを区別できないためより難しいです。この場合多分Hibernateにヒントを与える必要があります。それらは

- <version>においてunsaved-value="null"またはunsaved-value="negative"、またはクラスに対して<timestamp>プロパティ・マッピングを定義します。
- unsaved-value="none"を設定し、update(parent)をコールする前に新しくインスタンス化された子を明示的にsave()します。
- unsaved-value="any"を設定し、update(parent)をコールする前に以前の永続の子を明示的にupdate()します。

noneは割り当てられた識別子と複合識別子のデフォルトのunsaved-valueです。

一つの更なる可能性があります。isUnsaved()という名前の新しいInterceptorメソッドがあります。このメソッドは新しくインスタンス化されたオブジェクトを区別するための独自の戦略をアプリケーションに実装させることを可能にします。例えば、永続クラスに対して基本クラスを定義することもできます。

```
public class Persistent {
    private boolean _saved = false;
    public void onSave() {
        _saved=true;
    }
    public void onLoad() {
        _saved=true;
    }
    ......
    public boolean isSaved() {
        return _saved;
    }
}
```

(savedプロパティは永続的ではありません。)それでは以下のようにonLoad() と onSave()と一緒にisUnsaved()を実装しましょう。

```
public Boolean isUnsaved(Object entity) {
    if (entity instanceof Persistent) {
        return new Boolean( !( (Persistent) entity ).isSaved() );
    }
    else {
        return null;
    }
}

public boolean onLoad(Object entity,
    Serializable id,
    Object[] state,
    String[] propertyNames,
    Type[] types) {

    if (entity instanceof Persistent) ( (Persistent) entity ).onLoad();
    return false;
}

public boolean onSave(Object entity,
    Serializable id,
    Object[] state,
    String[] propertyNames,
    Type[] types) {

    if (entity instanceof Persistent) ( (Persistent) entity ).onSave();
    return false;
}
```

16.5. 結論

ここではかなりの量を要約したので、最初の頃は混乱しているように思われるかもしれません。しかし実際は、すべて非常に良く動作します。ほとんどのHibernateアプリケーションでは多くの局面でで親子パターンを使用しています。

最初の段落で代わりの方法につい触れました。それらは<composite-element>マッピングの場合は 存在せず、それは確かに親子関係の意味を持ちます。不幸にも複合要素クラスには二つの大きな制限があります。 : 一つは複合要素がコレクションを持つことができないこと。もう一つは他のユニークな親ではないエンティティの子となるべきではないということです。(しかし、それらは<idbag>マッピングを使って、代理の主キーを持てます。)

第17章 例: Weblogアプリケーション

17.1. 永続クラス

永続クラスがweblogと、weblogに掲示されたアイテムを表しています。それらは通常の親子関係としてモデリングされますが、setではなく順序を持ったbagを使用します。

```
package eg;

import java.util.List;

public class Blog {
    private Long _id;
    private String _name;
    private List _items;

    public Long getId() {
        return _id;
    }
    public List getItems() {
        return _items;
    }
    public String getName() {
        return _name;
    }
    public void setId(Long long1) {
        _id = long1;
    }
    public void setItems(List list) {
        _items = list;
    }
    public void setName(String string) {
        _name = string;
    }
}
```

```
package eg;

import java.text.DateFormat;
import java.util.Calendar;

public class BlogItem {
    private Long _id;
    private Calendar _datetime;
    private String _text;
    private String _title;
    private Blog _blog;

    public Blog getBlog() {
        return _blog;
    }
    public Calendar getDatetime() {
        return _datetime;
    }
    public Long getId() {
        return _id;
    }
    public String getText() {
        return _text;
    }
    public String getTitle() {
        return _title;
    }
    public void setBlog(Blog blog) {
        _blog = blog;
```

```
    }
    public void setDatetime(Calendar calendar) {
        _datetime = calendar;
    }
    public void setId(Long long1) {
        _id = long1;
    }
    public void setText(String string) {
        _text = string;
    }
    public void setTitle(String string) {
        _title = string;
    }
}
```

17.2. Hibernateのマッピング

XMLマッピングは直接的です

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping package="eg">

    <class
        name="Blog"
        table="BLOGS"
        lazy="true">

        <id
            name="id"
            column="BLOG_ID">

            <generator class="native"/>

        </id>

        <property
            name="name"
            column="NAME"
            not-null="true"
            unique="true"/>

        <bag
            name="items"
            inverse="true"
            lazy="true"
            order-by="DATE_TIME"
            cascade="all">

            <key column="BLOG_ID"/>
            <one-to-many class="BlogItem"/>

        </bag>

    </class>

</hibernate-mapping>
```

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
```

```
<hibernate-mapping package="eg">

    <class
        name="BlogItem"
        table="BLOG_ITEMS"
        dynamic-update="true">

        <id
            name="id"
            column="BLOG_ITEM_ID">

            <generator class="native"/>

        </id>

        <property
            name="title"
            column="TITLE"
            not-null="true"/>

        <property
            name="text"
            column="TEXT"
            not-null="true"/>

        <property
            name="datetime"
            column="DATE_TIME"
            not-null="true"/>

        <many-to-one
            name="blog"
            column="BLOG_ID"
            not-null="true"/>

    </class>

</hibernate-mapping>
```

17.3. Hibernateのコード

以下のクラスは、Hibernateでこれらのクラスを使ってできることをいくつか示しています。

```
package eg;

import java.util.ArrayList;
import java.util.Calendar;
import java.util.Iterator;
import java.util.List;

import net.sf.hibernate.HibernateException;
import net.sf.hibernate.Query;
import net.sf.hibernate.Session;
import net.sf.hibernate.SessionFactory;
import net.sf.hibernate.Transaction;
import net.sf.hibernate.cfg.Configuration;
import net.sf.hibernate.tool.hbm2ddl.SchemaExport;

public class BlogMain {

    private SessionFactory _sessions;

    public void configure() throws HibernateException {
        _sessions = new Configuration()
            .addClass(Blog.class)
            .addClass(BlogItem.class)
            .buildSessionFactory();
```

```
    }

    public void exportTables() throws HibernateException {
        Configuration cfg = new Configuration()
            .addClass(Blog.class)
            .addClass(BlogItem.class);
        new SchemaExport(cfg).create(true, true);
    }

    public Blog createBlog(String name) throws HibernateException {

        Blog blog = new Blog();
        blog.setName(name);
        blog.setItems( new ArrayList() );

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.save(blog);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return blog;
    }

    public BlogItem createBlogItem(Blog blog, String title, String text)
                        throws HibernateException {

        BlogItem item = new BlogItem();
        item.setTitle(title);
        item.setText(text);
        item.setBlog(blog);
        item.setDatetime( Calendar.getInstance() );
        blog.getItems().add(item);

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.update(blog);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return item;
    }

    public BlogItem createBlogItem(Long blogid, String title, String text)
                        throws HibernateException {

        BlogItem item = new BlogItem();
        item.setTitle(title);
        item.setText(text);
        item.setDatetime( Calendar.getInstance() );

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
```

```
            Blog blog = (Blog) session.load(Blog.class, blogid);
            item.setBlog(blog);
            blog.getItems().add(item);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return item;
    }

    public void updateBlogItem(BlogItem item, String text)
                    throws HibernateException {

        item.setText(text);

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.update(item);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
    }

    public void updateBlogItem(Long itemid, String text)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            BlogItem item = (BlogItem) session.load(BlogItem.class, itemid);
            item.setText(text);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
    }

    public List listAllBlogNamesAndItemCounts(int max)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        List result = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "select blog.id, blog.name, count(blogItem) " +
                "from Blog as blog " +
                "left outer join blog.items as blogItem " +
                "group by blog.name, blog.id " +
                "order by max(blogItem.datetime)"
            );
```

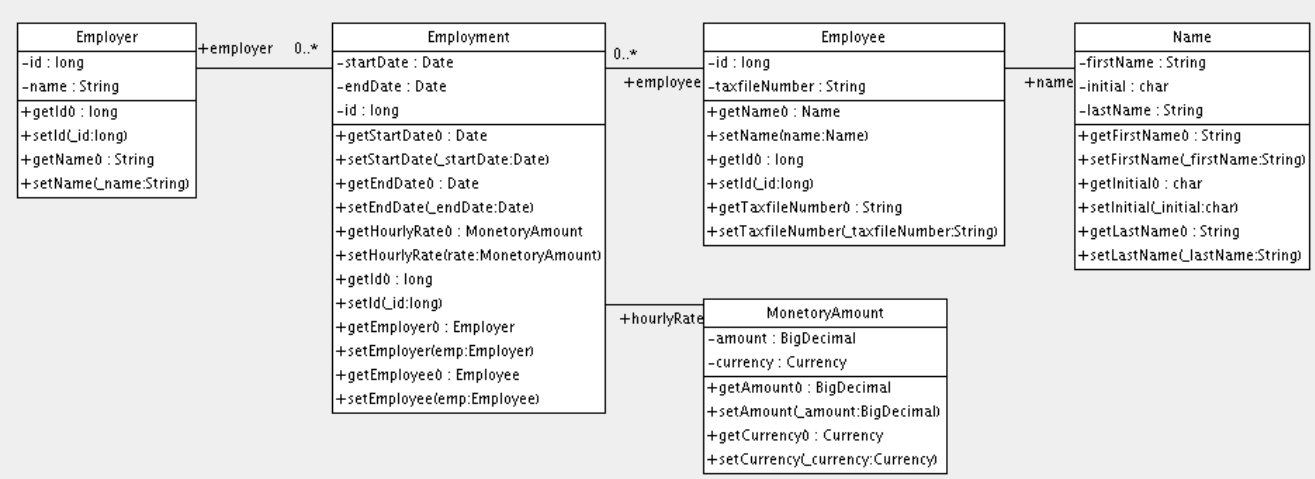
```
            q.setMaxResults(max);
            result = q.list();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return result;
    }

    public Blog getBlogAndAllItems(Long blogid)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        Blog blog = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "from Blog as blog " +
                "left outer join fetch blog.items " +
                "where blog.id = :blogid"
            );
            q.setParameter("blogid", blogid);
            blog  = (Blog) q.list().get(0);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return blog;
    }

    public List listBlogsAndRecentItems() throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        List result = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "from Blog as blog " +
                "inner join blog.items as blogItem " +
                "where blogItem.datetime > :minDate"
            );

            Calendar cal = Calendar.getInstance();
            cal.roll(Calendar.MONTH, false);
            q.setCalendar("minDate", cal);

            result = q.list();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return result;
    }
}
```

第18章 例：いろいろなマッピング

この章では、より複雑な関連のマッピングをいくつかお見せします。

18.1. 雇用者/従業員

`Employer` （雇用者）と `Employee` （従業員）の関係を表現する以下のモデルでは、それらの間の関連を表す具体的なエンティティ・クラス （`Employment`）（雇用）を使っています。 このようにしたのは、同じ組織に（ある一時期だけではなく）何度も雇用される場合が考えられるからです。 また、お金のバリューと雇用者の名前にはコンポーネントを使いました。



以下のようなマッピング・ドキュメントが考えられます：

```
<hibernate-mapping>

    <class name="Employer" table="employers">
        <id name="id">
            <generator class="sequence">
                <param name="sequence">employer_id_seq</param>
            </generator>
        </id>
        <property name="name"/>
    </class>

    <class name="Employment" table="employment_periods">

        <id name="id">
            <generator class="sequence">
                <param name="sequence">employment_id_seq</param>
            </generator>
        </id>
        <property name="startDate" column="start_date"/>
        <property name="endDate" column="end_date"/>

        <component name="hourlyRate" class="MonetoryAmount">
            <property name="amount">
                <column name="hourly_rate" sql-type="NUMERIC(12, 2)"/>
            </property>
            <property name="currency" length="12"/>
        </component>

        <many-to-one name="employer" column="employer_id" not-null="true"/>
        <many-to-one name="employee" column="employee_id" not-null="true"/>

    </class>
```

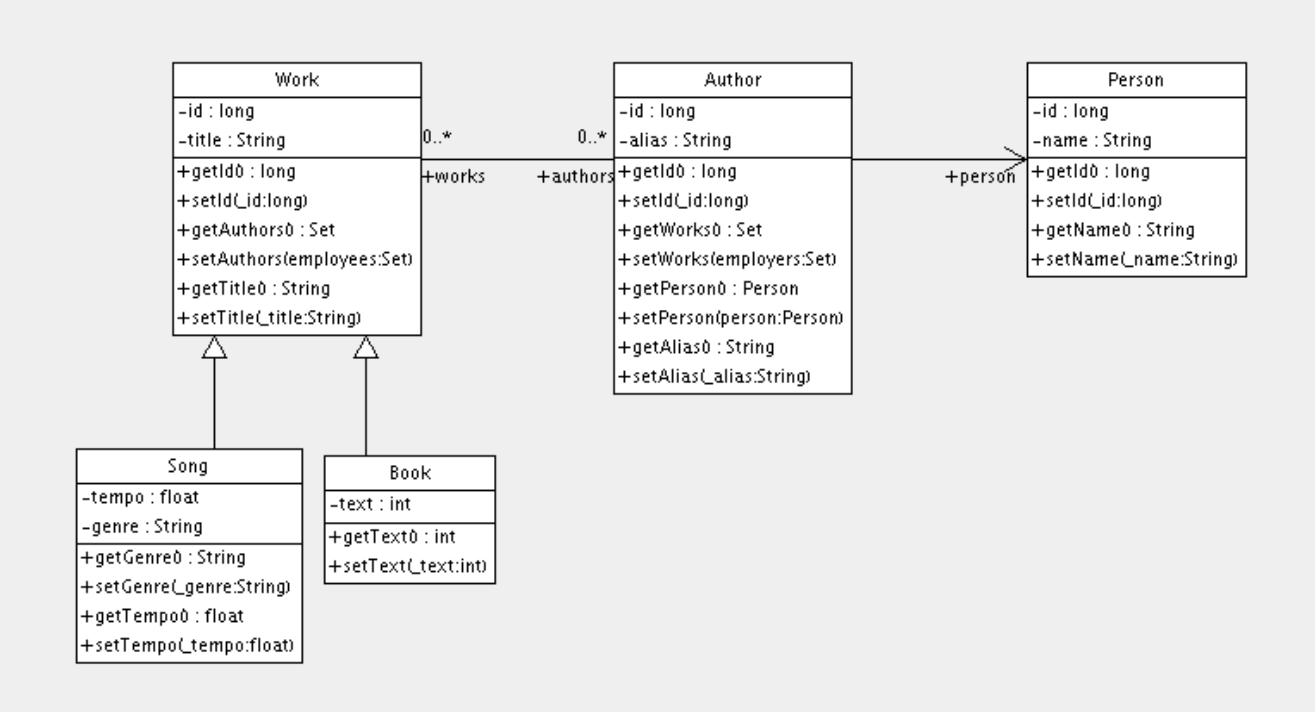
```
    <class name="Employee" table="employees">
        <id name="id">
            <generator class="sequence">
                <param name="sequence">employee_id_seq</param>
            </generator>
        </id>
        <property name="taxfileNumber"/>
        <component name="name" class="Name">
            <property name="firstName"/>
            <property name="initial"/>
            <property name="lastName"/>
        </component>
    </class>

</hibernate-mapping>
```

そして、これが SchemaExport で生成したテーブル・スキーマです。

```
create table employers (
    id BIGINT not null,
    name VARCHAR(255),
    primary key (id)
)

create table employment_periods (
    id BIGINT not null,
    hourly_rate NUMERIC(12, 2),
    currency VARCHAR(12),
    employee_id BIGINT not null,
    employer_id BIGINT not null,
    end_date TIMESTAMP,
    start_date TIMESTAMP,
    primary key (id)
)

create table employees (
    id BIGINT not null,
    firstName VARCHAR(255),
    initial CHAR(1),
    lastName VARCHAR(255),
    taxfileNumber VARCHAR(255),
    primary key (id)
)

alter table employment_periods
    add constraint employment_periodsFK0 foreign key (employer_id) references employers
alter table employment_periods
    add constraint employment_periodsFK1 foreign key (employee_id) references employees
create sequence employee_id_seq
create sequence employment_id_seq
create sequence employer_id_seq
```

18.2. 作者/作品

Work （作品）と Author （作者）と Person （人）の関係を表現する、以下のモデルについて考えてみましょう。 Work と Author の関係は、 many-to-many関連で表現しています。 また、 Author と Person の関係は、 one-to-one関連で表現しています。 他には Author が Person を拡張する方法なども考えられます。



以下のマッピング・ドキュメントでは、これらの関係が正しく表現されています：

```
<hibernate-mapping>

    <class name="Work" table="works" discriminator-value="W">

        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <discriminator column="type" type="character"/>

        <property name="title"/>
        <set name="authors" table="author_work" lazy="true">
            <key>
                <column name="work_id" not-null="true"/>
            </key>
            <many-to-many class="Author">
                <column name="author_id" not-null="true"/>
            </many-to-many>
        </set>

        <subclass name="Book" discriminator-value="B">
            <property name="text"/>
        </subclass>

        <subclass name="Song" discriminator-value="S">
            <property name="tempo"/>
            <property name="genre"/>
        </subclass>

    </class>

    <class name="Author" table="authors">

        <id name="id" column="id">
            <!-- The Author must have the same identifier as the Person -->
            <generator class="assigned"/>
        </id>

        <property name="alias"/>
        <one-to-one name="person" constrained="true"/>

        <set name="works" table="author_work" inverse="true" lazy="true">
            <key column="author_id"/>
            <many-to-many class="Work" column="work_id"/>
```

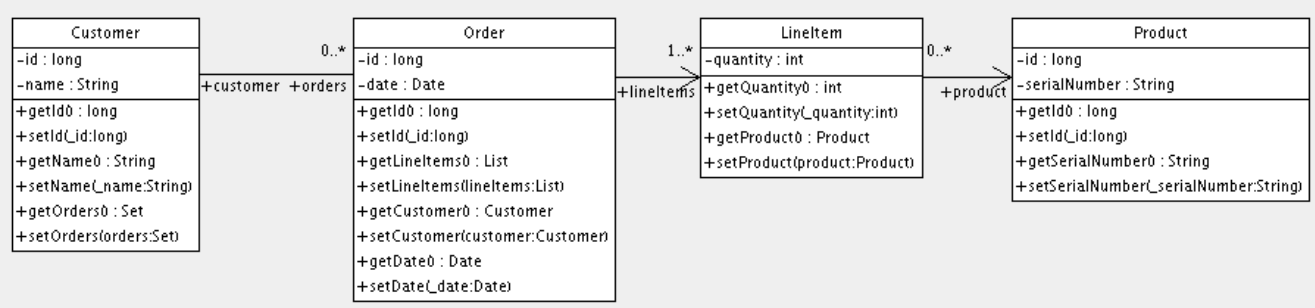
```
        </set>

    </class>

    <class name="Person" table="persons">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

このマッピングにはテーブルが4つあります。 works, authors, persons は、 作品、作者、人のデータをそれぞれ格納します。 author_work は作者と作品をリンクする関連テーブルです。 そして、これが SchemaExport で生成したテーブル・スキーマです。

```
create table works (
    id BIGINT not null generated by default as identity,
    tempo FLOAT,
    genre VARCHAR(255),
    text INTEGER,
    title VARCHAR(255),
    type CHAR(1) not null,
    primary key (id)
)

create table author_work (
    author_id BIGINT not null,
    work_id BIGINT not null,
    primary key (work_id, author_id)
)

create table authors (
    id BIGINT not null generated by default as identity,
    alias VARCHAR(255),
    primary key (id)
)

create table persons (
    id BIGINT not null generated by default as identity,
    name VARCHAR(255),
    primary key (id)
)

alter table authors
    add constraint authorsFK0 foreign key (id) references persons
alter table author_work
    add constraint author_workFK0 foreign key (author_id) references authors
alter table author_work
    add constraint author_workFK1 foreign key (work_id) references works
```

18.3. 顧客/注文/製品

さて今度は、Customer （顧客）と Order （注文）と LineItem （注文明細）と Product （製品）の関係を 表現するモデルについて考えてみましょう。 Customer と Order の関係は、 one-to-many 関連で表現しています。 それでは、Order / LineItem / Product はどのように表現すべきでしょうか。 私は Order と Product の間の many-to-many関連を表現する関連クラスとして、 LineItem をマッピングすることにしました。Hibernateではこれをコンポジット要素と呼んでいます。



マッピング・ドキュメントです：

```
<hibernate-mapping>

    <class name="Customer" table="customers">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
        <set name="orders" inverse="true" lazy="true">
            <key column="customer_id"/>
            <one-to-many class="Order"/>
        </set>
    </class>

    <class name="Order" table="orders">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="date"/>
        <many-to-one name="customer" column="customer_id"/>
        <list name="lineItems" table="line_items" lazy="true">
            <key column="order_id"/>
            <index column="line_number"/>
            <composite-element class="LineItem">
                <property name="quantity"/>
                <many-to-one name="product" column="product_id"/>
            </composite-element>
        </list>
    </class>

    <class name="Product" table="products">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="serialNumber"/>
    </class>

</hibernate-mapping>
```

customers, orders, line_items, products は 顧客、注文、注文明細、製品のデータをそれぞれ格納します。 line_items は、注文と製品をリンクする関連テーブルとしても働きます。

```
create table customers (
    id BIGINT not null generated by default as identity,
    name VARCHAR(255),
    primary key (id)
)

create table orders (
    id BIGINT not null generated by default as identity,
    customer_id BIGINT,
    date TIMESTAMP,
    primary key (id)
)

create table line_items (
```

```
    line_number INTEGER not null,
    order_id BIGINT not null,
    product_id BIGINT,
    quantity INTEGER,
    primary key (order_id, line_number)
)

create table products (
    id BIGINT not null generated by default as identity,
    serialNumber VARCHAR(255),
    primary key (id)
)

alter table orders
    add constraint ordersFK0 foreign key (customer_id) references customers
alter table line_items
    add constraint line_itemsFK0 foreign key (product_id) references products
alter table line_items
    add constraint line_itemsFK1 foreign key (order_id) references orders
```

第19章 ベスト・プラクティス

細かい粒度でクラスを書き、<component> でマッピングしましょう。
: 例えば street（通り）, suburb （都市）, state 州）, postcode （郵便番号）で構成される Address（住所）クラスを使いましょう。 そうすればコードが再利用しやすくなり、リファクタリングも簡単になります。

永続クラスには識別子プロパティを定義しましょう。
: 識別子プロパティはオプションになっていますが、いくつもの理由から使うべきです。 「人工的」で（被生成の意味。ビジネス上の意味を持たないこと）、 プリミティブ型でない識別子をおすすめします。柔軟性を最大にするためには、 java.lang.Long または java.lang.String を使いましょう。

マッピングはクラスごとの別のファイルに分けましょう。
: 大きい一枚岩のようなマッピング・ドキュメントはいけません。 例えば com.eg.Foo クラスは、 com/eg/Foo.hbm.xml ファイルにマッピングしましょう。 そうすれば、特にチームで開発を行う場合に、良い結果をもたらします。

リソースとしてマッピング・ファイルをロードしましょう。
: マッピング・ファイルは対応するクラス・ファイルと一緒に配置しましょう。

クエリ文字列を外に出しましょう。
: クエリ文字列がANSI標準ではないSQL関数をコールするなら、これは良い習慣です。 クエリ文字列を外に出すと、アプリケーションを他へ移植することが簡単になります。

バインド変数を使いましょう。
: JDBCのように、変化する値は必ず"?"で置き換えましょう。 具体的な値の代入に文字列操作を使っては、絶対にいけません。 名前つきのパラメータを使うとさらに良いです。

JDBCコネクションを自分で管理してはいけません。
: Hibernateでは、アプリケーションがJDBCコネクションを管理することができます。 しかし、これは最後の手段だと思ってください。 もし組み込みのコネクション・プロバイダを使うことができなければ、 net.sf.hibernate.connection.ConnectionProvider の実装を考えてください。

カスタム型の使用を考えましょう。
: 例えば、どこかのライブラリから持ってきたJava型を永続化する必要があるとしましょう。 しかしその型には、コンポーネントとしてマッピングするために必要なアクセサが用意されていないとします。 このような場合は、net.sf.hibernate.UserType を実装することを考えるべきです。 そうすれば、Hibernate型との実装変換を心配せずに、アプリケーションのコードを扱えます。

ボトルネックを解消するためには、JDBCをハンド・コードしましょう。
: システムのパフォーマンス・クリティカルな領域では、 （例えば大量の update / delete のような）ある種の操作には、 直接JDBCを利用すると良いかもしれません。 しかし、何がボトルネックなのかが わかる までは待ってください。 そして、直接JDBCを利用するからといって、必ずしも速くなるとは限らないことも理解してください。 直接JDBCを利用する必要があれば、Hibernateの Session をオープンして、 SQLコネクションを使うと良いかもしれません。それならまだ同じトランザクション戦略と、 コネクション・プロバイダを使うことができるからです。

`Session` のフラッシュを理解しましょう。
たまに、Sessionが永続状態をデータベースと同期させることがあります。 しかし、もしこれが頻繁に起こるようでは、パフォーマンスに影響が出てきてしまいます。 自動フラッシュを無効にしたり、特定のトランザクションにおけるクエリや操作の順番を変更することで、 不必要なフラッシュを最小限にすることができます。

3層アーキテクチャでは `saveOrUpdate()` の使用を考えましょう。
サーブレット / セッションビーン・アーキテクチャを使うとき、 セッションビーンでロードされた永続オブジェクトを、 サーブレット / JSP層との間でやりとりすることができます。 その際、各リクエストに対して新しいSessionを使ってください。 そしてオブジェクトの永続状態を更新するためには、 `Session.update()` や `Session.saveOrUpdate()` を使ってください。

2層アーキテクチャではSessionの切断の使用を考えましょう。
最高のスケーラビリティを得るためには、データベース・トランザクションをできるだけ短くしなければいけません。 しかし、ユーザから見た1つの仕事に対して、長いアプリケーション・トランザクションを実装する 必要があることがしばしばです。アプリケーション・トランザクションは、 クライアントのリクエストとレスポンスのサイクルを、いくつかまたぐものになるかもしれません。 切り離されたオブジェクトを使うか、2層アーキテクチャにおいては、 単にJDBCコネクションからHibernate Sessionを切断し、各後続のリクエストにはそれを再接続してください。 複数のアプリケーション・トランザクション・ユースケースに対して、 1つのSessionを使いまわすことはやめてください。 そうでなければ、古くなったデータを使うことになってしまいます。

例外を復帰可能なものとして扱ってはいけません。
これは「最も良い」習慣以上の、必須の習慣です。 例外が発生したときは、`Transaction` をロールバックして、 `Session` をクローズしてください。 そうでなければ、Hibernateはメモリの状態が永続状態を正確に表現していることを保証できません。 特別な場合として、与えられた識別子のインスタンスがデータベースに存在するかどうかを判定するために、 `Session.load()` を使うことはやめてください。 その代わりに `find()` を使ってください。 しかし、例えば `StaleObjectStateException` や `ObjectNotFoundException` のように、復帰可能な例外もいくつか存在します。

関連をフェッチするためには、できるだけlazyを選びましょう。
eagerな（アウター･ジョイン）フェッチは控えめに使ってください。 JVMレベルでキャッシュされないクラスへのほとんどの関連には、プロキシやlazyコレクションを使ってください。 キャッシュされるクラスへの関連、つまりキャッシュがヒットする可能性が高い関連には、`outer-join="false"` として明示的にeagerなフェッチを無効にしてください。 ある特定のユースケースに対してアウター･ジョインによるフェッチが適切なときは、 `left join` を用いたクエリを使ってください。

ビジネス・ロジックをHibernateから抽象化することを考えましょう。
インターフェイスを使い、（Hibernateの）データ・アクセスコードを隠蔽してください。 DAO と Thread Local Session パターンを組み合わせてください。 `UserType` を通してHibernateに関連付けることで、 ハンド・コーディングしたJDBCで永続化するクラスを持つこともできます。 （このアドバイスは「十分大きな」アプリケーションに対してのものです。 つまりテーブルを5個しか使わない（小さな）アプリケーションに対しては当てはまりません。）

ユニークなビジネス・キーを使い、`equals()` と `hashCode()` を実装しましょう。
セッション・スコープの外でオブジェクトを比較するなら、 `equals()` と `hashCode()` を実装しなければいけません。 セッション・スコープの内部では、Javaオブジェクトのアイデンティテ

ィは保証されています。 これらのメソッドを実装するなら、決してデータベース識別子を使ってはいけません。 一時的なオブジェクトは識別子の値を持たないため、Hibernateはオブジェクトがセーブされるときに値を代入します。 もしセーブ時にオブジェクトがsetの中にあると、ハッシュ・コードが変更され、 契約が破られてしまいます。 `equals()` と `hashCode()` を実装するには、 ユニークなビジネス・キーを使ってください。 つまりクラスのプロパティのユニークな組み合わせを比較するようにしてください。 オブジェクトがsetの内部にある間、このキーは安定でユニークでなければならないことを覚えておいてください （ライフタイム全体でなくても構いません。データベースの主キーと同じように安定ではありません。）。 `equals()` の比較（lazyローディング）の中では、絶対にコレクションを使わないでください。 そして、プロキシを介する他の関連クラスについても、注意してください。

外来の関連マッピングは使わないでください。

現実には、many-to-many関連の使いどころはめったにありません。 ほとんどの場合「リンク・テーブル」に付加的な情報が必要になります。 この場合、間のリンク・クラスに2つのone-to-many関連を使う方がずっと良いです。 実際に私たちは、ほとんどの関連がone-to-manyかmany-to-oneだと考えています。 他の関連を使うなら、注意深く、それからそれが本当に必要かどうかを考えてみてください。