

特別編 第11章

O/Rマッピング
フレームワーク
Hibernate

業務アプリケーションのように大量のデータを扱うシステムでは、データベースとの連携が不可欠です。最もよく用いられるデータベースはリレーショナルデータベース（RDB）ですが、表の構造をしたRDBとプログラム内部のオブジェクトとではデータ形式が異なるため、連携にはデータ間のマッピングをプログラムする必要があります^{注1}。

オブジェクトとRDBのデータ間でマッピングを行うためのフレームワークを「O/R（Object/Relational）マッピングフレームワーク」と呼びます。O/RマッピングフレームワークにはHibernate、Torque、Cayenne、iBatisなどがありますが、その中でもHibernateは特に知名度の高いフレームワークです。

Hibernateは、複数データベースへの対応やパフォーマンス最適化のための工夫といった、本格的なO/Rマッピングに必要な仕組みを知るための教材として読むことができます。同時に、Hibernateはイベント駆動型のアーキテクチャという一面も備えます。イベントベースのシステムを開発する際の指針として、Hibernateを参照することもできるでしょう。

本章で扱うソースコードのバージョンは、Hibernate 3.1.3です。

注1 RDBとオブジェクトとでデータ形式に違いがあることは「インピーダンスミスマッチ」と呼ばれます。

11.1 読み解くテーマ

O/Rマッピングの中核は、RDBからデータを取得し、それをオブジェクトにマッピングする部分にあります。しかし、こうした部分の実装はJDBC APIとリフレクションを使う基本的なものであり、汎用性を実現するために複雑になっています。本書の読解に取り上げるには、あまり適していません。

本章では、よりフレームワークの設計部分に焦点を当てた読解をすることにします。以下の2テーマを読みます。

- ① イベント駆動型システムの作り方
- ② データベースごとのSQL方言に対応するためのクラス設計

まずHibernateの動作原理を把握するために、①でそのイベントベースのアーキテクチャを理解します。②では、異なるSQL方言を持った複数のデータベースに対応するために、Hibernateがどのようなクラス設計を用いているかを読むことにします。

11.2 Hibernateの概要

Hibernateは、他のO/Rマッピングフレームワークと同様、POJO^{注2}をRDBのテーブルと関連付け、オブジェクトの操作だけでRDB上のレコードを追加/検索/更新/削除できる仕組みを提供するフレームワークです。MySQL、PostgreSQL、Oracleをはじめ、ほとんどのRDBサーバに対応しています。第9章で読解するHSQLDBにも対応しています。

オブジェクトとRDBテーブルとの関連付けは、XML形式のマッピングファイル(Xxx.hbm.xml)で定義します。オブジェクト間の一对一、一对多、多対多の関連に対応しているほか、ID自動生成の方法を指定することもできます。

オブジェクトの検索方法には、SQLを直接用いた検索のほか、HQL(Hibernate Query Language)^{注3}というSQLに似た独自の問い合わせ言語によるもの、純粋にオブジェクト指向のAPI(クワイテリアクエリ)によるものがあります。

11.2.1 ソースコードの構成

Hibernateは、ソースコードとバイナリをひとまとめに配布しています。配布アーカイブは、図1のような構成になります。

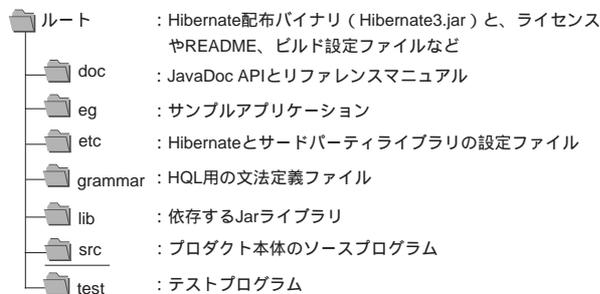


図1 Hibernateソースコードのフォルダ構成

ビルドツールはAntです。サンプルコードのフォルダが「eg」^{注4}という名前であるのが多少変わっているものの、それ以外は素直な構成です。

先述のとおり、HibernateはHQLという独自言語を持ちます。「grammar」フォルダには、HQL文法とパーサの定義ファイルが置かれています。Hibernateでは、ANTLR^{注5}というパーサジェネレータが使われています。

NOTE

注2 Plain Old Java Object。実装上の制約を持たない、通常のJavaオブジェクトのこと。

NOTE

注3 HQLは、一種のDSL(Domain Specific Language、ドメイン特化言語)と見ることもできます。本章では試みませんが、第7章のVelocityと同様、HibernateをDSL実装系として読むことも可能です。

NOTE

注4 英語で「e.g.」は「たとえば」の意味です。ラテン語の「exempli gratia」から来ています。

NOTE

注5 <http://www.antlr.org/>

11.2.2 パッケージ構成

パッケージ構成は、図2の通りです。

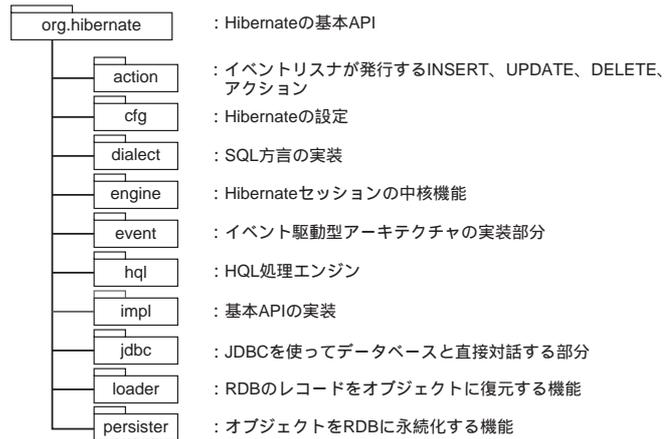


図2 主要パッケージの構成

Hibernateのパッケージは非常に多いので、図2では説明に支障のない範囲でパッケージを絞りました。

Hibernateの基本APIとして定められているのが、最上位パッケージのorg.hibernateとその下のcfgパッケージです。ユーザが普段用いるSessionFactory、Session、Transactionの3インタフェースは最上位パッケージに、Configurationクラスはcfgパッケージにあります。図2に挙げたその他のパッケージは、普段ユーザに意識されることはありません。

implパッケージには、最上位パッケージにある基本APIの実装クラスが置かれます。Sessionインタフェースの実装もimplパッケージが提供しますが、その内部の中心的な機能はengineパッケージに置かれています。

Hibernateの特徴であるイベントベースのアーキテクチャを実装するのが、eventパッケージです。オブジェクトの永続化、状態更新、削除などは、すべてイベントとして表現されます。発生したイベントは、イベントリスナがアクションを発行することで処理されます。アクションはすべて、actionパッケージのクラスです。

loaderパッケージとpersisterパッケージとは、対照的な役割を持ちます。前者はデータベースからオブジェクトを読み込み、後者はオブジェクトをデータベースに永続化します。どちらのパッケージも、実際のデータベース接続はjdbcパッケージを介して行います。

hqlパッケージには、検索時にHQLを解析するためのパーサと抽象構文木(AST)[※]とがあります。dialectパッケージは、データベースごとに微妙に異なるSQLの方言の

NOTE
注6 パーサと抽象構文木については、第7章を参照。

違いを吸収する仕組みを提供します。

11.2.3 主要クラス間の関係

Hibernateの構造は複雑なので、説明を2段階に分けます。まず、オブジェクトの永続化（追加／更新／削除）に絞って、設定からセッションまでのシステム全体を説明します（①）。次に、セッションのみに注目して、クエリ（検索）部分の構造を説明します（②）。

①システムの全体像

全体像は、図3のようになります。

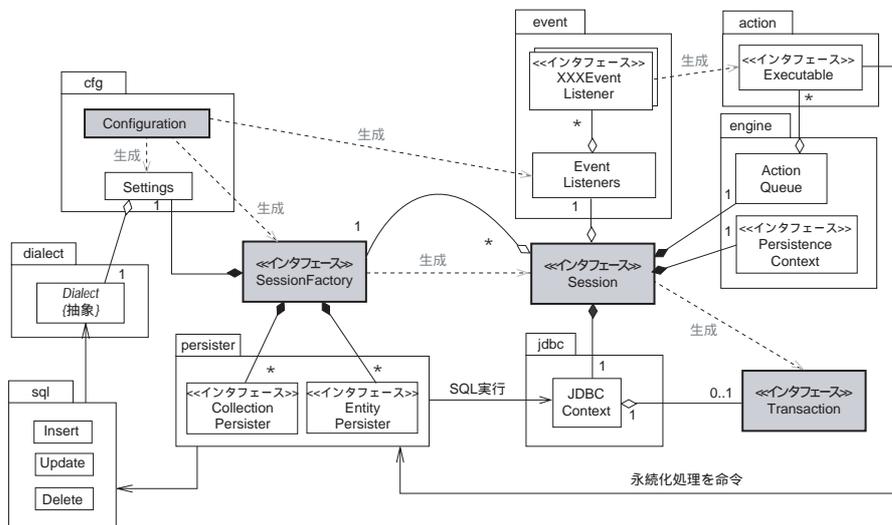


図3 全体的なクラス構造

Hibernateの設定情報を表すのが、Configurationクラス（図3左上）です。hibernate.propertiesやhibernate.cfg.xmlなどの設定ファイルから、情報を読み込みます。Configurationクラスの役目は、セッションファクトリの生成です。

Configurationオブジェクトはセッションファクトリ生成時に、DB接続やトランザクション管理、キャッシュなどの方法を定義したSettingsオブジェクトを生成し、それをセッションファクトリに渡します。Settingsオブジェクトには、SQL方言（Dialectクラス、図3左中）も含まれます。

ユーザプログラムは、Hibernateをセッションという単位で操作します。セッションを生成するのが、セッションファクトリ（SessionFactoryインタフェース、図3左

NOTE

注7 パーシスタはセッションでなくセッションファクトリに関連付けられており、複数セッションで共有される設計になります。そのため、パーシスタクラスはセッションに対してパラメータ化される必要があり、メソッドはすべてセッションを引数に持つことになります。

NOTE

注8 たとえば従業員 (Employee) クラスや会社 (Company) クラスのように、実世界に存在する実体をモデル化したクラスを「エンティティ」と呼びます。

NOTE

注9 実際にパーシスタが呼び出すのは、JDBCContextクラスが所有するBatcherインタフェースです。

NOTE

注10 そのほかDBのネイティブSQLを用いた検索も可能ですが、ここでは説明を割愛します。

中)です。セッションファクトリは、オブジェクト永続化を役目とするパーシスタを持ちます^{注7}。

パーシスタには、コレクション用 (CollectionPersisterインタフェース) とエンティティ^{注8}用 (EntityPersisterインタフェース) とがあります。パーシスタは、コレクションやエンティティとRDBテーブルとのマッピング定義ごとに用意されるため、セッションファクトリはそれぞれ複数のパーシスタを持つことになります。

セッション (Sessionインタフェース、図3中央) がHibernateの心臓部です。ユーザプログラムはセッションに対して、トランザクション (Transactionインタフェース、図3右) の開始と終了、オブジェクトの永続化やクエリの要求を行います。セッション中に読み出したり、新たに追加された永続化オブジェクトを一時的に記録したりしておくために、セッションには永続化コンテキスト (PersistenceContextインタフェース、図3右) があります。

セッションはイベントリスナの集合 (EventListenersクラス) を保持しており、セッションに対して追加/更新/削除の要求があると、そのイベントに対応するイベントリスナ (XXXEventListenerインタフェース、図3右上) に通知します。イベントを受け取ったイベントリスナは、イベント処理のためにアクション (Executableインタフェース、図3右上) を生成し、それをセッションのアクションキュー (ActionQueueクラス) に蓄えます。

トランザクションのコミット時、セッションはアクションキューに貯まったアクションをすべて実行します。アクションはパーシスタに対し、永続化処理の命令を出します。パーシスタはsqlパッケージのクラス (図3左下) を使ってSQLを生成し、セッションが管理するデータベース接続 (JDBCContextクラス、図3右下) ^{注9}を通してSQLを実行します。

②クエリ部分の構造

上記の説明では、クエリによってDBからオブジェクトを取得する部分の構造が省略されています。クエリ部分の構造を説明すると、セッションを中心として図4のようになります。

クエリの方法を大別すると、オブジェクトAPIによるクライテリア (Criteriaインタフェース) クエリと、SQLライクな検索を可能にするHQLクエリ (Queryインタフェース) との2つがあります^{注10}。どちらのクエリもセッションから生成されますが、処理は再びセッションに委譲され、実際の検索はセッションが行います。

クライテリアクエリの場合、セッションはクライテリア用のオブジェクトローダ (CriteriaLoaderクラス) を呼び出し、そのローダがクライテリアを元にSQLを構築します。

HQLクエリの場合、セッションは、HQLを解析して実行する役目のHQLQueryPlanオブジェクトに処理を渡します。HQLQueryPlanオブジェクトの内部では、QueryTranslatorオブジェクトによってHQLが解析され、構築されたSQLがHQLクエ

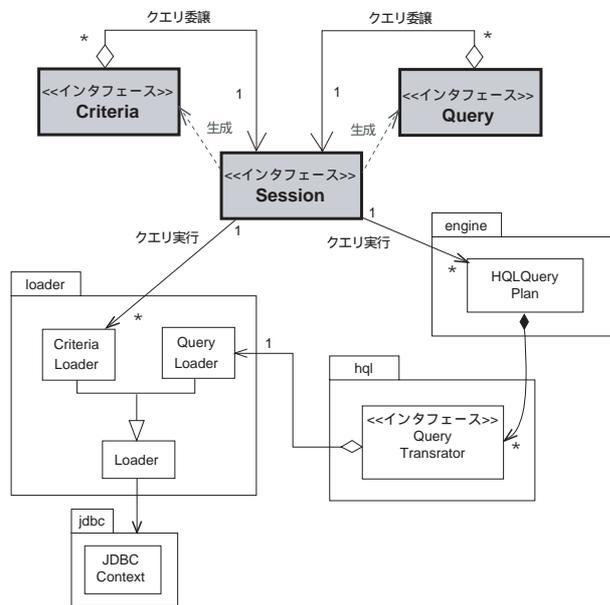


図4 クエリ部分の構造

リ用のオブジェクトローダ（QueryLoaderクラス）に渡されます。

クライテリア用、HQLクエリ用どちらのオブジェクトローダも、上位クラスのLoaderがSQLをHibernateのJDBC層（JDBCContextオブジェクト）^{注11}に対して実行し、結果セット（java.sql.ResultSetオブジェクト）を取得します。

クエリ部分の構造を概観すると、このようになります。

NOTE

注11 実際に呼び出されるのはBatcherインタフェースです。

■ C O L U M N ■

O/RマッピングフレームワークとData Mapperパターン

エンタープライズアプリケーションアーキテクチャパターンでは、データソースのアーキテクチャパターンとして4つのパターンを挙げています。Table Data Gateway、Row Data Gateway、Active Record、Data Mapperの4パターンです(表A)。

表A データソースアーキテクチャの4パターン

| パターン | 説明 |
|------------------------|--|
| Table Data Gatewayパターン | RDBの1テーブルごとにデータアクセス用のクラス(ゲートウェイ)を用意し、ロジックとデータアクセスとを分離する。J2EEパターン(参考文献[1])のDAO(Data Access Object)パターンと同じ |
| Row Data Gatewayパターン | テーブルの1レコードごとにゲートウェイオブジェクトを用意し、ロジックとデータアクセスとを分離する |
| Active Recordパターン | ロジックとデータアクセスとを1つのクラスにまとめる。EJB 2のエンティティBeanやRuby on Rails ^{注A} の例が有名 |
| Data Mapperパターン | データマップがロジックとデータベースとのマッピングを行うことで、ロジックからデータアクセス層への依存を完全に排除する |

注A スクリプト言語RubyのWebアプリケーションフレームワーク。

表の4パターンのうち、ロジックからデータアクセスのコードを完全に排除できるのはData Mapperパターンです。Data Mapperパターンのクラス図は、図Aの通りです。名前が示すとおり、Data MapperパターンはO/Rマッピングフレームワークの本質を捉えたパターンです。

Hibernateをアーキテクチャの視点で捉えるときは、Data Mapperパターンの枠組みで考えるとよいでしょう。



図A Data Mapperパターン

11.3 イベント駆動型システムの作り方



システムの拡張性を高める設計の1つに、イベント駆動型のアーキテクチャがあります。イベント駆動型とは、システムが重要な動作を起こしたタイミングがイベントとなり、イベントリスナに通知されるようなシステム（図5）です。

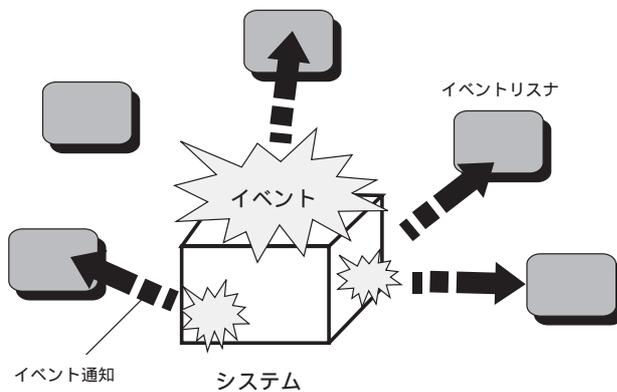


図5 イベント駆動型のシステム

イベント駆動型の特徴は、イベントがどう処理されるかということ、イベントの発生源（イベント源）は一切知らなくてよいことにあります。イベント源は単に、登録されているイベントリスナへイベントを通知する仕組みを持つだけです。イベントリスナを登録することで、イベント源はそのままに、システムに後からさまざまな振る舞いを追加していくことができます。

Hibernateは、バージョン3よりイベント駆動型のアーキテクチャになりました。オブジェクトの追加や更新、削除、読み込みは、すべてイベントとして処理されます。クエリを除くHibernate本来の機能までイベントリスナとして実装されており、徹底したアーキテクチャになっています。

イベント駆動型のアーキテクチャはHibernateの柔軟な拡張を可能にしましたが、とくにセキュリティを宣言的に設定可能になったことが大きな特徴です。認証用のリスナを、オブジェクトの更新や読み込みのイベントに登録することで、設定ファイル上にアクセス権限などの定義を書くことができます。

ここでは、イベント駆動型のシステムがどのように作られているかを、読み解くことにします。読解によって、Hibernateの動作原理を理解することができるでしょう。

11.3.1 イベント駆動型システムとObserverパターン

イベント駆動型のシステムを実装するには、GoFパターンの1つであるObserverパターンを使うのが定石です。Observerパターンとは、「あるオブジェクトに状態変化が起こったときに、それを監視するすべてのオブジェクトにその状態変化を通知する」仕組みです。状態変化をイベントと読みかえれば、Observerパターンは正にイベント駆動型のシステムを作るためのパターンといえます。

Observerパターンの構造は、図6のようなクラス図で表されます。

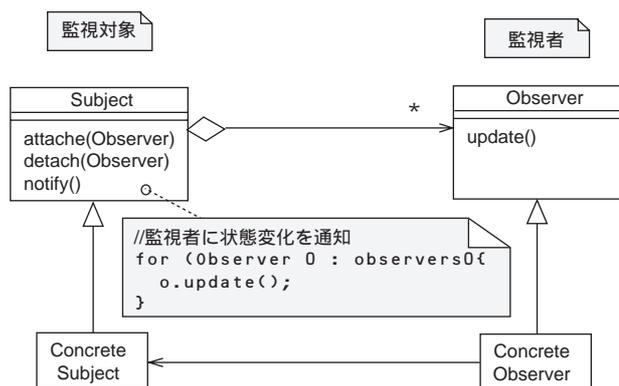


図6 Observerパターンのクラス図

Observerパターンには、「監視対象」(Subjectクラス)と「監視者」(Observerクラス)の2つの役割が存在します。監視対象が状態変化を起こすオブジェクトで、状態変化の通知を待つのが監視者です。

監視対象は、監視者を登録するための何らかのインタフェース(図6ではattach / detachメソッド)を持ちます。このインタフェースを通して、監視者の具体的な実装(ConcreteObserverクラス)が登録されます。

監視対象はまた、監視者に状態変化を知らせるための通知メソッド(図ではnotifyメソッド)を持ちます。監視対象の具体的な実装(ConcreteSubjectクラス)は、インスタンスが状態を変化させる度に通知メソッドを呼び出すことで、すべての監視者に状態変化を通知します。

Observerパターンの重要なポイントは、SubjectクラスとそのサブクラスはObserverクラスにしか依存せず、その実装クラスまでは知らないことです。監視対象は監視者が誰であるかを知らなくてよいので、さまざまな機能を持つ監視者を後から自由に登録できるわけです。

なおJavaの世界では、「監視者」には慣習的にリスナ(listener)という名前が付けられます。

1.1.3.2 ソースコードのポイント

読解するソースコードの流れをシーケンス図に表すと、図7のとおりです。

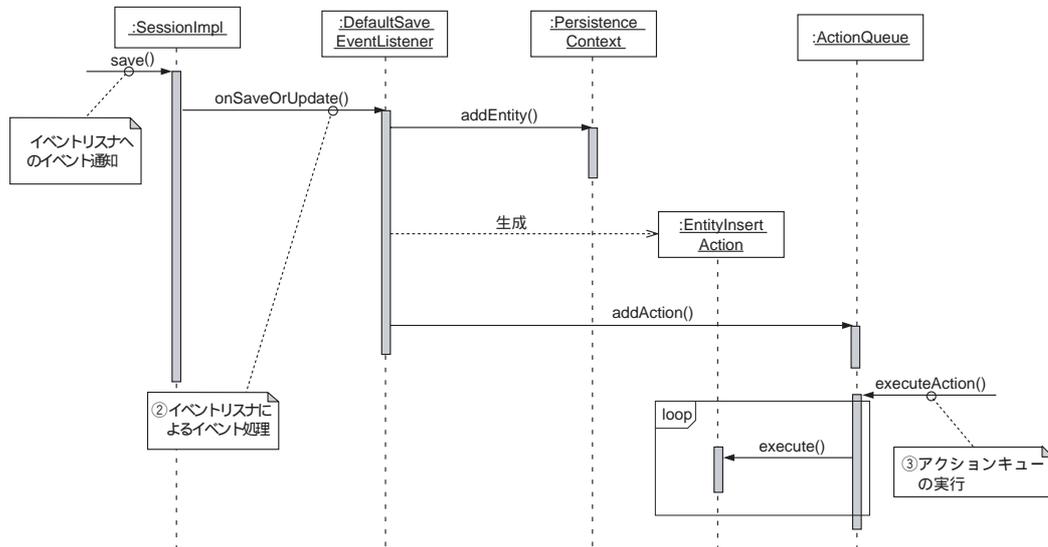


図7 イベント処理の流れ

次の3段階に分けて、読解を進めていきます。

- ① イベントリスナへのイベント通知
- ② イベントリスナによるイベント処理
- ③ アクションキューの実行

最初の段階だけ読めば、Observerパターンを使ってどうやってイベント駆動の仕組みが実装されているかがわかります。それ以降の段階は、Hibernateの動作メカニズムを知るためにイベント処理の続きを読みます。

11.3.3 ソースコード読解(イベント駆動型システムの作り方)

Hibernateを使ってオブジェクトをDBに新規保存するプログラミングは、リスト1のようになります。

```
SessionFactory sessionFactory = ... // セッションファクトリは生成済みとする
Session session = sessionFactory.openSession(); // セッションの生成
Transaction tx = null;
try {
    tx = session.beginTransaction(); // トランザクションの開始
    MyEntity entity = new MyEntity(); // エンティティ
    session.save(entity); // エンティティを永続化 ①
    tx.commit(); // トランザクションをコミット
} catch (Exception e) {
    // トランザクションをロールバック
} finally {
    session.close(); // セッションを閉じる
}
```

リスト1 Hibernateの使い方

① イベントリスナへのイベント通知

オブジェクトの保存をセッションに命令しているのは、リスト1-①です。Sessionインスタンスがイベント駆動型システムの「監視対象」であり、ここでオブジェクト保存のイベントが発生します。

Sessionインタフェースの実装クラスは、org.hibernate.impl.SessionImplクラスです。SessionImpl#saveメソッドは、メソッド内でもう1つのsaveメソッド(リスト2)を呼び出します。

```
public Serializable save(String entityName, Object object) throws HibernateException {
    // 通知メソッド呼び出し
    return fireSave( new SaveOrUpdateEvent(entityName, object, this) ); ①
}
```

リスト2 org.hibernate.impl.SessionImpl#saveメソッド

saveメソッドは、保存イベント(SaveOrUpdateEventインスタンス)を生成して

fireSaveメソッドを呼び出します(リスト2-①)。

fireSaveメソッドはリスト3です。

```
private transient EventListeners listeners; // 登録された「監視者」のストック場所 ②
...中略...
private Serializable fireSave(SaveOrUpdateEvent event) {
    ...中略...
    // 保存イベントのために登録された「監視者」の取得
    SaveOrUpdateEventListener[] saveEventListener = listeners.getSaveEventListeners();
    // すべての「監視者」へイベント通知
    for ( int i = 0; i < saveEventListener.length; i++ ) {
        saveEventListener[i].onSaveOrUpdate(event);
    }
    return event.getResultId();
}
```

リスト3 org.hibernate.impl.SessionImpl#fireSaveメソッド

fireSaveメソッドが、Observerパターンの通知メソッド(図8のnotifyメソッドに相当)です。保存イベントを監視する「監視者」を取得し、イベントの通知を行います(リスト3-①)。

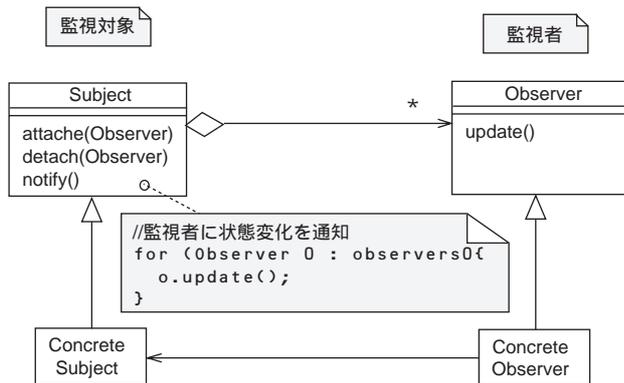


図8 Observerパターンのクラス図(図6の再掲)

EventListenersオブジェクト(リスト3-②)が、「監視者」を登録するインターフェース(図8のattach/detachメソッド)になっています。EventListenersクラスはリスト4のように、「監視者」を登録したり^{注12}取得したりするメソッドを持ちます。

デフォルトで登録されている「監視者」の実装は、DefaultSaveEventListenerオブジェクトです(リスト4-①)。

ここまでの読解を元に、セッションの構造をObserverパターンに当てはめると図9のようになります。

典型的なObserverパターンの構造と少し異なるのは、「監視対象」に相当するSessionImplクラスが上位クラスとサブクラスの2つに分かれておらず、イベントの発

NOTE
注12 ユーザは、設定ファイルhibernate.cfg.xmlか、あるいはConfigureクラスのメソッドを使って、イベントリスナを登録することができます。

```

// 保存イベント用の「監視者」
private SaveOrUpdateEventListener[] saveEventListeners = { new DefaultSaveEventListener() }; — ①
...中略...
// 「監視者」取得のメソッド
public SaveOrUpdateEventListener[] getSaveEventListeners() {
    return saveEventListeners;
}
// 「監視者」登録のメソッド
public void setSaveEventListeners(SaveOrUpdateEventListener[] saveEventListener) {
    this.saveEventListeners = saveEventListener;
}
    
```

リスト4 org.hibernate.event.EventListenersクラス (抜粋)

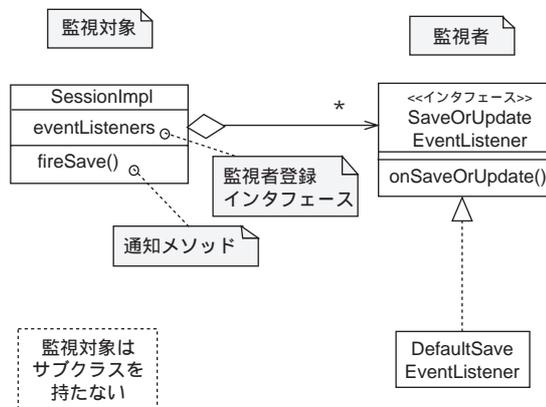


図9 HibernateのObserverパターン

生と通知が1つのクラスで定義されていることです。

② イベントリスナによるイベント処理

イベント駆動型システムの仕組みはここまでで理解できましたが、Hibernateがイベントを最終的にどのように処理するのかを、ついでに読み解くことにします。

保存イベントの通知によって呼び出されるonSaveOrUpdateメソッドは、リスト5のようになっています。

メソッド内では最終的にオブジェクトの保存が実行されて、新たに割り当てられたIDがイベント内に記録されます(リスト5-①)。

オブジェクトの保存を実行するperformSaveOrUpdateメソッドを読みます(リスト6)。

まずセッションの永続化コンテキストに、オブジェクトのエントリがすでに存在するかをチェック^{注14}します(リスト6-①)。エントリが見つからなければ新規のオブジェクトだと判明するので、オブジェクトを保存する処理に移ります(リスト6-②)。

entityIsTransientメソッドから先のメソッド呼び出しの流れは長いので、途中を少し省きます。最終的にスーパークラスに定義されているperformSaveOrReplicateメソ

NOTE

注14 オブジェクトは、いったん永続化されセッションの管理下に置かれると、永続化コンテキスト(Persistence Contextオブジェクト)にエントリが作成されます。

```

public void onSaveOrUpdate(SaveOrUpdateEvent event) throws HibernateException {
    final SessionImplementor source = event.getSession();
    final Object object = event.getObject();
    ...中略...
    // 遅延ロード注13を使っている場合の、プロキシの再設定処理
    if ( reassociateIfUninitializedProxy(object, source) ) {
        ...中略...
    }
    else {
        ...中略...
        // 通常の新規保存ではこちらが実行される
        // オブジェクトの保存を実行して、生成されたIDを取得
        event.setResultId( performSaveOrUpdate(event) ); ①
    }
}

```

リスト5 org.hibernate.event.def.DefaultSaveOrUpdateEventListener
#onSaveOrUpdateメソッド

```

protected Serializable performSaveOrUpdate(SaveOrUpdateEvent event) {
    ...中略...
    // セッションの永続化コンテキストにエン트리があるかを確認
    EntityEntry entry = event.getSession().getPersistenceContext().getEntry( event.getEntity() ); ①
    if ( entry!=null && entry.getStatus() != Status.DELETED ) {
        // すでに永続化されているオブジェクトの場合
        return entityIsPersistent(event);
    }
    else {
        // まだ永続化されていない新しいオブジェクトの場合
        return entityIsTransient(event); ②
    }
}

```

リスト6 org.hibernate.event.def.DefaultSaveEventListener#performSaveOrUpdateメソッド

ッド(リスト7)が呼び出されます。

performSaveOrReplicateメソッドでは、まずセッションの永続化コンテキストに保存対象のオブジェクトを登録します(リスト7-①)。次に、SQLのINSERT文に相当するINSERTアクション(EntityInsertActionオブジェクト)を生成し、それをセッションのアクションキューに追加します(リスト7-②)。

アクションキューにINSERTアクションを追加すると、後でまとめてデータベースへの保存が実行されます。アクションキューは、リスト8-①②のように各種のアクションを蓄えており、トランザクションのコミットなど一定のタイミングでまとめて実行されます。

NOTE

注13 遅延ロード(lazy load)とは、データが実際に参照されるまでデータをデータベースから取得しないことをいいます。遅延ロードされたオブジェクトは、初めにデータの代わりにプロキシを保持した状態で生成されます。Hibernate 3では、オブジェクトの関連とコレクションは、デフォルトで遅延ロードにより読み込まれます。

```

protected Serializable performSaveOrReplicate(
    Object entity,
    EntityKey key,
    EntityPersister persister,
    boolean useIdentityColumn,
    Object anything,
    EventSource source)
throws HibernateException {
    ...中略...
    Serializable id = key==null ? null : key.getIdentfier();
    ...中略...
    Object[] values = persister.getPropertyValuesToInsert(entity, getMergeMap(anything), source);
    ...中略...
    // セッションの永続化コンテキストにオブジェクトを登録
    // source/パラメータはセッション (SessionImplオブジェクト)
    Object version = Versioning.getVersion(values, persister);
    source.getPersistenceContext().addEntity(
        entity,
        Status.MANAGED,
        values,
        key,
        version,
        LockMode.WRITE,
        useIdentityColumn,
        persister,
        isVersionIncrementDisabled(),
        false
    );
    ...中略...
    // INSERTアクションをセッションのアクションキューに登録
    if ( !useIdentityColumn ) {
        source.getActionQueue().addAction(
            new EntityInsertAction(id, values, entity, version, persister, source)
        );
    }
    ...中略...
    return id;
}

```

リスト7 org.hibernate.event.def.AbstractSaveEventListener
#performSaveOrReplicateメソッド

```

private ArrayList insertions; // INSERTアクションのキュー
private ArrayList deletions; // DELETEアクションのキュー
private ArrayList updates; // UPDATEアクションのキュー
...中略...
public void addAction(EntityInsertAction action) {
    insertions.add( action );
}

```

リスト8 org.hibernate.engine.ActionQueue#addActionメソッド

③アクションキューの実行

アクションキューがアクションを実行するところまで、読んでおきましょう。アクションキューが一括実行される時は、executeActionsメソッドが呼び出されます(リスト9)。

```

private ArrayList insertions;
private ArrayList deletions;
private ArrayList updates;
...中略...
public void executeActions() throws HibernateException {
    executeActions( insertions );
    executeActions( updates );
    ...中略...
    executeActions( deletions );
}
...中略...
private void executeActions(List list) throws HibernateException {
    int size = list.size();
    for ( int i = 0; i < size; i++ ) {
        execute( (Executable) list.get(i) );
    }
    list.clear();
    session.getBatcher().executeBatch();
}

```

リスト9 org.hibernate.engine.ActionQueue#executeActionsメソッド

executeActionsメソッドは、もう1つのexecuteActionsメソッドを呼び出して、各アクションを実行していきます（リスト9-①）。2つ目のexecuteActionsメソッドでは、アクションが1つずつ実行されます（リスト9-②）。

executeメソッドはリスト10-①のとおり、アクション（Executableインスタンス）を実行します。

```

public void execute(Executable executable) {
    ...中略...
    executable.execute();
}

```

リスト10 org.hibernate.engine.ActionQueue#executeメソッド

この後、アクションがパーシスタを呼び出し、パーシスタによってオブジェクトの永続化処理が行われることは、Hibernateの概要で説明したとおりです。

イベント駆動型のシステムは、本質的なメカニズムはそれほど難しくはないものの、読解する側としては非常に難しいシステムと言えます。イベントの発生時に、どんなイベントリスナが登録されているかを把握しておかないと、何をやっているのか一向に理解できないからです。それは、イベント源（「監視対象」）がイベントリスナ（「監視者」）の実装をまったく知らなくてよいという、Observerパターンの拡張性の高さの裏返しです。

■ C O L U M N ■

小読解: ログ出力のちょっとしたテクニック

Commons Loggingでログ出力のレベルを柔軟にカスタマイズしたいときは、どうすればよいでしょうか？

Commons Loggingの一般的な使い方では、ログレベル (DEBUG、INFO、WARN、ERRORなど) はパッケージ (またはクラス) ごとに設定されます^{注A}。たとえば、ロギング実装にLog4Jを用いた場合、log4j.propertiesに

```
log4j.to.msn.wings.opensource.hibernate = INFO
```

と設定すると、to.msn.wings.opensource.hibernateパッケージにあるクラスはすべて、INFOレベル以上のログのみ出力されます。

しかし、あるクラスの一部の挙動がシステムの重要な役割を担っており、その部分だけ独自にログレベルを設定したいこともあります。Hibernateでは、マッピング情報から自動生成されるSQLや、HQLクエリの解析結果などは、フレームワークの挙動を調べるために他とは別にログを残したい箇所です。

Hibernateは、重要な挙動をもつクラスに2つのロガー (Logインスタンス) を用意することで、ログ出力の細かいカスタマイズを可能にしています。リストAは、HQLクエリを解析するクラスの一部です。

Commons Loggingの通常の使い方ではロガーは1つ用意するだけでいいのですが、QueryTranslatorImplクラスには2つのロガーが用意されます (リストA-①)。特別なロガー (AST_LOGフィールド) の生成時に、LogFactory #getLogメソッドにClassインスタンスでなく「org.hibernate.hql.ast.AST」という独自の文字列を渡していることがポイントです (Hibernateにはorg.hibernate.hql.ast.ASTというクラスは存在しません)。

そして特別な箇所のログ出力には、AST_LOGフィールドを使います (リストA-②)。もちろん他の部分では、通常のロガーを使ってログ出力をします。

Hibernateを使う際に、log4j.propertiesに

```
org.hibernate.hql.ast.AST = DEBUG
```

と設定すれば、QueryTranslatorImplクラスの他の挙動をログ出力することなく、HQLクエリ解析の様子だけをピンポイントで確認することができます。

注A 詳しい読者向けに説明すると、Commons Loggingではログのカテゴリをクラスの完全修飾名にするのが一般的な使い方になっているということです。

```

public class QueryTranslatorImpl implements FilterTranslator {
    // 通常のロガー
    private static final Log log = LogFactory.getLog( QueryTranslatorImpl.class );
    // 特別な箇所のためのロガー
    private static final Log AST_LOG = LogFactory.getLog( "org.hibernate.hql.ast.AST" );
    ...中略...
    private HqlSqlWalker analyze(HqlParser parser, String collectionRole)
        throws QueryException, RecognitionException {
        ...中略...
        // 特別な箇所のログ出力
        if ( AST_LOG.isDebugEnabled() ) {
            ASTPrinter printer = new ASTPrinter( SqlTokenTypes.class );
            AST_LOG.debug( printer.showAsString( w.getAST(), "---- SQL AST ----" ) );
        }
        ...中略...
    }
}

```

リストA org.hibernate.hql.ast.QueryTranslatorImplクラス (抜粋)

■ C O L U M N ■

Hibernate 3.2とJava Persistence API

Hibernateの次期バージョンとなる3.2は、本書の執筆時点(2006年8月)でリリース候補(CR2)が公開されています。

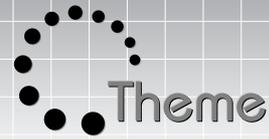
3.2の大きな目玉は、サブコンポーネントHibernate EntityManagerとの組み合わせによるJava Persistence API(JPA)の実装です。JPAは、EJB 3が採用するJava標準の永続化APIです。3.2の狙いは、永続化フレームワークとしては老舗のHibernateを、新しい標準API(EJB 3.0)に対応させることにあります。

そのほか3.2には、以下の変更点があります。

- ①CGLIB以外のバイトコード生成ライブラリ(Javassistなど)のサポート
- ②トランザクション外にあるオブジェクトの振る舞いの変更
- ③HQL、ネイティブSQL処理方法の改良

O/Rマッピングフレームワークとしての大きな変更点はないため、本章で読解したソースコードへの影響は少ないと考えられます。

11.4 データベースごとのSQL方言に対応するためのクラス設計



データベースシステムには、MySQL、HSQLDB、Oracle、Microsoft SQL Serverなど、オープンソースから商用までさまざまな製品があります。SQLには「SQL92」や「SQL99」といった標準規格があるものの、製品ごとに微妙な違い（SQL方言）が存在します。

データベースとアプリケーションとの間を取りもつフレームワークは、アプリケーションが特定のデータベースに依存しないように、データベースの種類による違いを吸収する必要があります。SQLの方言による違いは、以下のような点があります。

- ①カラム型の種類と名称
- ②関数の種類
- ③構文（INSERT、SELECT、UPDATE、DELETEなど）の違いや有無

こうしたSQL方言の違いを吸収するクラス設計の例を、Hibernateに学ぶことにします。

11.4.1 ソースコードのポイント

SQL方言の多様性をフレームワークで吸収しようとするには、多様性をどうやってモデル化するかがポイントになります。SQL方言を表現するクラスは、org.hibernate.dialect.Dialectクラスとそのサブクラスです。

Dialectクラスの設計上のポイントは、以下の3つです。

- ①カラム型、関数、予約語の違いはフィールドによりモデル化する
- ②他の構文上の細かな違いは、メソッドによりモデル化する
- ③SQL方言の定義は、Dialectのサブクラス化により行う

11.4.2 ソースコード読解（データベースごとのSQL方言に対応するためのクラス設計）

オブジェクトローダ（loaderパッケージ）やパーシスタ（persisterパッケージ）、SQL生成ヘルパークラス（sqlパッケージ）など、Hibernateの中でSQLを発行する役割のクラスが、実行時にDialectオブジェクトを参照することで方言に対応しています。

①Dialectクラスの設計

Dialectクラスの設計は、図10のようになります。

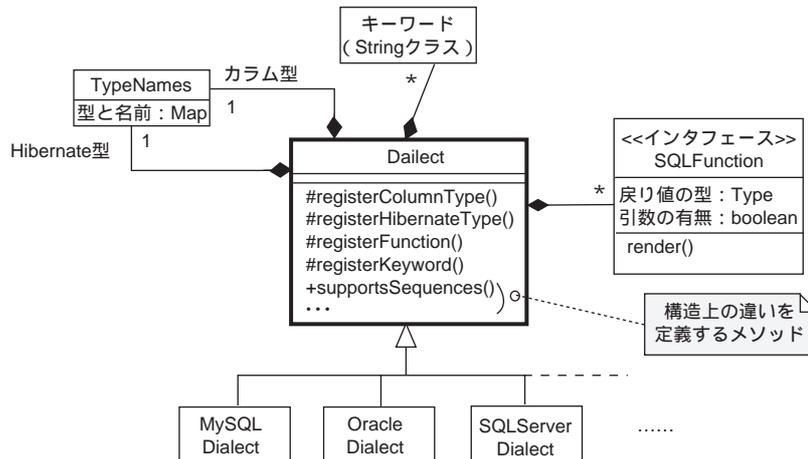


図10 Dialectクラスの設計

Dialectクラスはまず、SQLカラム（列）型とHibernate型の2つのマップ（TypeNamesクラス、図10左上）を持ちます。どちらのマップも、キーはJavaのコアAPIに用意されたSQL型（java.sql.Typesクラス）で、値はそのデータベースでの型名（「bigint」や「varchar」など）です。Hibernate型のマップは、データベースからデータを読み込む際に、そのデータ型を判別するときなどに利用されます。

またDialectクラスは、関数（SQLFunctionインタフェース、図10右）とキーワード（文字列、図10上）のセットを持ちます。関数のセットは、そのSQL方言でサポートされている関数を意味します。キーワードとは、SQL方言に固有の予約語（一般的なSQL構文は含まない）を指します。

各SQL方言へ対応するには、Dialectのサブクラス（図10下）を作ります。SQL方言毎に異なるカラム型、Hibernate型^{注15}、関数、キーワードは、サブクラスのコンストラクタ呼び出し時にDialect#registerXxxメソッドを通して登録されます。リスト11は、MySQL方言であるMySQLDialectクラスのコンストラクタです。

NOTE

注15 後述するように、データベースによるHibernate型の違いはまず存在しません。Hibernate内部の型と、コアAPIのSQL型との対応を定義するものだからです。

```

public MySQLDialect() {
    super();
    // カラム型登録
    registerColumnType( Types.BIT, "bit" );
    registerColumnType( Types.BIGINT, "bigint" );
    registerColumnType( Types.SMALLINT, "smallint" );
    ...中略...

    // 関数登録
    registerFunction("ascii", new StandardSQLFunction("ascii", Hibernate.INTEGER) );
    registerFunction("bin", new StandardSQLFunction("bin", Hibernate.STRING) );
    registerFunction("char_length", new StandardSQLFunction("char_length", Hibernate.LONG) );
    ...中略...
}

```

リスト11 org.hibernate.dialect.MySQLDialectコンストラクタ

カラム型が登録され(リスト11-①)、関数が登録されています(リスト11-②)。Hibernate型は方言による違いがないため、上位クラス(Dialectクラス)のコンストラクタで登録されます(リスト11-③)。また、Dialectクラスのコンストラクタでは、SQL92の標準的な関数の登録も行っています。Dialectクラスの読解は省略します。

キーワードの登録を見るために、もう1つ別のコンストラクタを読みます。SQL Serverでは、

```
SELECT TOP 5 * FROM my_table
```

のようにして、SELECT文の最初の5行だけを取得できる「TOP」という特殊な構文があります。このような特殊構文は、キーワードとして登録される必要があります(リスト12)。

```

public SQLServerDialect() {
    ...中略...
    // 「TOP」キーワード登録
    registerKeyword("top");
}

```

リスト12 org.hibernate.dialect.SQLServerDialectコンストラクタ

SQL方言のカラム型と関数、予約語の違いは、このようにモデル化されています。

②SQL方言への対応方法

SQLは、方言によって他にもさまざまな構文上の違いがあります。図10には1つしか掲載していませんが、Dialectクラスにはそうした細かな違いを設定するためのメソッドが数多く用意されています。Dialectクラスは標準的なSQLのためのデフォルトの実装を与えており、サブクラスで特定のメソッドをオーバーライドすることによって、方言の特徴を設定します。

たとえば、IDの自動生成を行う「シーケンス」機能の有無を判定する

supportsSequencesメソッドは、デフォルトではfalseを返し機能がサポートされない定義になっています(リスト13)。

```
public boolean supportsSequences() {  
    // デフォルト設定はシーケンスのサポートなし  
    return false;  
}
```

リスト13 org.hibernate.dialect.Dialect#supportsSequencesメソッド

多くのデータベースがシーケンスをサポートしていますが、たとえばHSQLDBはシーケンスをサポートしているため、リスト14のようにメソッドをオーバーライドしています。

```
public boolean supportsSequences() {  
    // HSQLDBはシーケンスをサポートしている  
    return true;  
}
```

リスト14 org.hibernate.dialect.HSQLDialect#supportsSequencesメソッド

一方、MySQLのようにシーケンスのないデータベースでは、supportsSequencesメソッドのオーバーライドはありません。

メソッド数が膨大なためここではすべてを掲載できませんが^{注16}、DialectクラスはまさにSQL方言を詳細にモデル化した成果です。アプリケーションの構築中にSQL方言への対応が必要になったら、DialectクラスのSQL方言モデルを手本として参照することができます。

NOTE

注16 HibernateのJavaDoc(http://www.hibernate.org/hib_docs/v3/api/)を参照してください。

■ C O L U M N ■

Adapterパターンとの比較

コンポーネントのインタフェースの違いを吸収する設計方法として、第2章「汎用ライブラリJakarta Commons Logging/Pool」でAdapterパターンを取り上げました。Adapterパターンが解決しようとする問題と、SQL方言に対応する問題とは、どちらも既存のシステムの差異を吸収しようとする点で似ています。しかし、前提にいくつか違いがあります。

まず、Adapterパターンは既存クラスを対象とするものですが、SQL方言の場合はSQLという言語を対象とします。クラスと言語とでは、適用できるアプローチの仕方が異なります。また、Adapterパターンでは、インタフェースの違いをクライアントコードから完全に隠蔽することが求められますが、Hibernateでは最適化のために、時として、特定のデータベースでしか使えないようなSQLをクライアントコードが発行したいこともあります。

AdapterパターンでなくDialectクラスのような解決策がHibernateに必要なのは、このような前提の違いがあるからです。問題が似通っていても前提が異なれば、異なる解決策が必要となります。