

## 第2章

# 启动项目

### 本章内容

- 用Hibernate和Java Persistence实现“Hello World”
- 正向和反向工程的工具集
- Hibernate配置和整合

你想开始使用Hibernate和Java Persistence，想通过一个循序渐进的示例进行学习，想看到这两个持久化API，以及如何从原生的Hibernate或者标准化的JPA中受益。这就是本章的内容：一次简单的“Hello World”应用之旅。

然而，在Hibernate参考文档中已经有了一个公开可用的很棒的完整教程，因此为了避免重复，这里将给出有关Hibernate整合和配置的更为详细的指导。如果你想从一个更为简短的教程开始，以便能在一小时之内完成，建议你考虑阅读Hibernate参考文档。它引导你使用Hibernate：从一个简单独立的Java应用程序，到最本质的映射概念，最后示范了在Tomcat上部署的一个Hibernate Web应用程序。

本章将介绍如何给一个整合了Hibernate的简单Java应用程序建立项目基础，你会看到在这样的环境中配置Hibernate的更多细节。我们也会讨论Hibernate在一个托管环境中（即在提供了Java EE服务的环境中）的配置和整合。

将Ant作为“Hello World”项目的一个构建工具，并创建不仅能够编译和运行项目，而且能够利用Hibernate Tools的构建脚本。根据开发过程，可以使用Hibernate工具集自动导出数据库Schema，或者甚至从一个现有（遗留）的数据库Schema中通过反向工程得出一个完整的应用程序。

就像每一位好的工程师，在开始第一个真正的Hibernate项目之前，你应该准备工具，并决定采用什么样的开发过程。并且，根据所选择的过程，你自然会喜欢使用不同的工具。让我们看看这个准备阶段，以及你有哪些选择，然后开始一个Hibernate项目。

## 2.1 启动 Hibernate 项目

在有些项目中，应用程序的开发是通过开发人员以面向对象方式分析业务领域来驱动的。在其他项目中，则深受现有关系数据模型的影响，比如一个遗留的数据库或者由专业的数据建模者

设计的一个全新模式。有几种选择，并且要在开始之前回答下列问题：

- 可以从头开始简洁地设计一个新的业务需求吗？还是有遗留数据和（或）遗留应用程序代码？
- 某些必要的部分可以从现有的文件中自动生成吗（例如，从现有的数据库Schema中产生Java源代码）？数据库Schema可以从Java代码和Hibernate映射元数据中产生吗？
- 哪种工具可用来支持这项工作？其他工具能支持整个开发周期吗？

当我们在接下来的几节中建立基础的Hibernate项目时，将讨论这些问题。下面是你建立基础的Hibernate项目的过程：

- (1) 选择开发过程。
- (2) 建立项目基础。
- (3) 编写应用代码和映射。
- (4) 配置和启动Hibernate。
- (5) 运行应用。

读完接下来的几节，你就能为自己的项目准备正确的方法了，并且也具备了了解本章稍后要介绍的更复杂场景的背景信息。

### 2.1.1 选择开发过程

我们先从总体上看看可用的工具、用作源输入的制品，以及生成的输出。图2-1展现了Ant的各种导入和导出任务，所有功能对Eclipse的Hibernate Tools插件都可用。阅读本章时请参照这张图<sup>①</sup>。

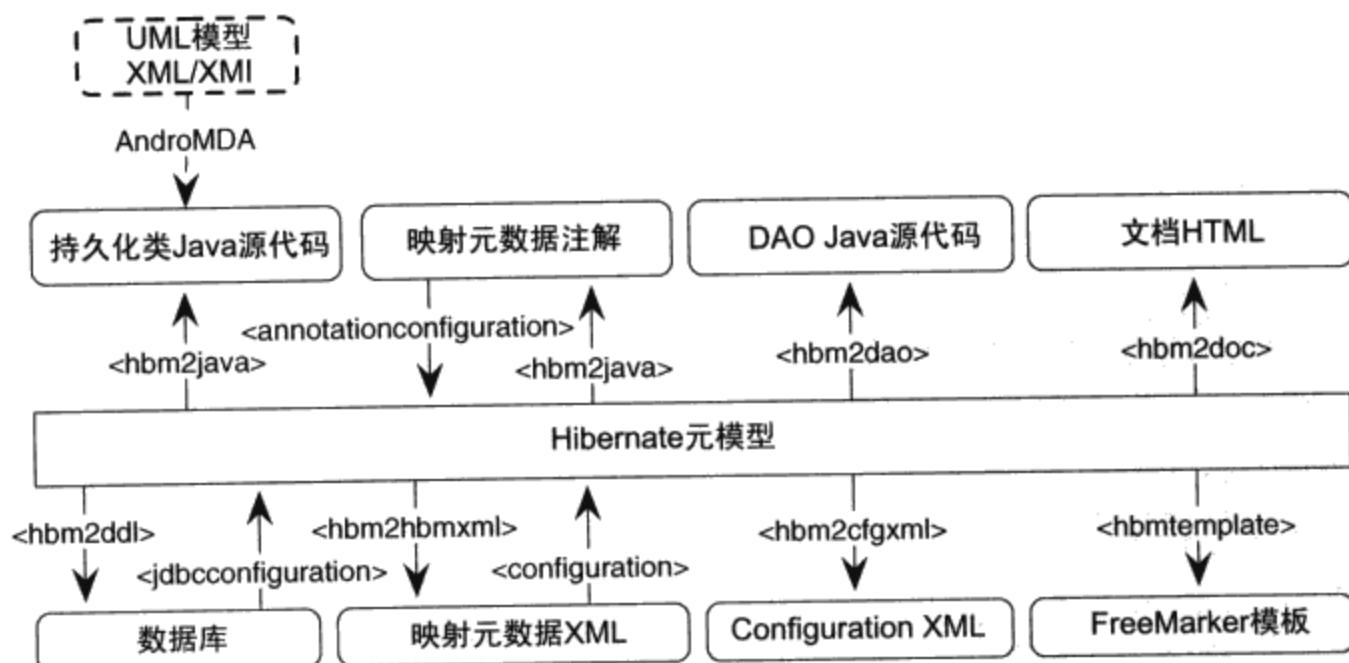


图2-1 用于Hibernate开发的工具的输入和输出

<sup>①</sup> AndroMDA是一种从UML图文件产生POJO源代码的工具，严格来说不是常用Hibernate工具箱的一部分，因此不在本章讨论。更多有关用于AndroMDA的Hibernate模块信息，请见Hibernate网站上的社区区域。

---

**说明** Eclipse IDE的Hibernate Tools是Eclipse IDE的插件（Eclipse 的JBoss IDE的一部分——一组向导、编辑器和Eclipse中的额外视图，帮助开发EJB3、Hibernate、JBoss Seam和基于JBoss中间件的其他Java应用）。正向和反向工程的特性相当于基于Ant的工具。另外的Hibernate Console视图允许对数据库执行特别的Hibernate查询（HQL和Criteria），并以图表形式浏览结果。Hibernate Tools XML编辑器支持映射文件的自动完成，包括类、属性，甚至表格和列名称。在撰写本书期间，图形工具仍在开发中，并以beta版的形式发行使用。然而，软件的未来发行版将废弃所有的屏幕截图。Hibernate Tools的文档包含许多屏幕截图和详细的项目构建指导，可以轻松地使用Eclipse IDE构建你的第一个“Hello World”程序。

---

以下为常见的开发场景：

- **自顶向下 (top-down)** —— 在自顶向下的开发中，从一个现有的领域模型开始，使用Java完成领域模型的实现，并（理想地）完成数据库Schema方面的自主。你必须构建映射元数据——用XML文件或者通过注解Java源代码，然后选择性地让Hibernate的hbm2ddl工具产生数据库Schema。没有现成的数据库Schema，对于大部分Java开发人员来说，是最舒适的开发风格。你甚至可以使用Hibernate Tools在每个应用程序于开发过程中重启时，自动刷新数据库Schema。
- **自底向上 (bottom-up)** —— 反之，自底向上开发始于一个现有的数据库Schema和数据模型。此时，最容易进行的方式就是使用反向工程工具从数据库中抽取元数据。这个元数据可被用来生成XML映射文件，例如用hbm2hbmxml。利用hbm2java，Hibernate映射元数据用来生成Java持久化类，甚至数据访问对象——换句话说，是Java持久层的一个骨架。或者，不编写到XML映射文件，被注解的Java源代码（EJB 3.0实体类）可以直接由工具生成。但是，并非所有的类关联细节和Java专有的元信息都可以用这种策略自动从SQL数据库Schema中生成，因此需要一些手工的工作。
- **起自中间层 (middle out)** —— Hibernate XML映射元数据提供充分的信息来完全推导出数据库Schema，并给应用的持久层生成Java源代码。此外，XML映射文档不会太冗长。因此，有些架构师和开发人员更喜欢起自中间层开发，他们在那开始手写Hibernate XML映射文件，然后用hbm2ddl生成数据库Schema，用hbm2java生成Java类。Hibernate XML映射文件在开发期间被不断更新，且其他所要创建的东西从这个主定义中生成。另外的业务逻辑或者数据库对象通过子类化和辅助的DDL添加。这种开发风格只建议经验丰富的Hibernate专家使用。
- **在中间会合 (meet in the middle)** —— 最难的场景是合并现有的Java类和现有的数据库Schema。此时，Hibernate工具集就无能为力了。当然，不可能把任意的Java领域模型映射给一个给定的Schema，因此这个场景通常至少需要重构一些Java类、数据库Schema或者兼而有之。映射元数据几乎一定要在XML文件中用手工编写（虽然它可能使用注解，如果很匹配的话）。这会是个难以置信的痛苦场景，的确痛苦，幸运的是这种情况非常罕见。我们现在要更深入地探讨工具和它们的配置选项，并给典型的Hibernate应用程序开发建立一

个工作环境。你可以按照我们循序渐进的指导，构建同样的环境；或者可以只是零碎地选取一些你所需要的，例如Ant构建脚本。

首先假设开发过程是自顶向下的，并且将快速完成一个不涉及任何遗留数据Schema或者Java代码的Hibernate项目。之后，将把代码迁移到JPA和EJB 3.0，然后从一个现有的数据库Schema反向工程，自底向上地开始一个项目。

## 2.1.2 建立项目

假设你已经从Hibernate网站（<http://www.hibernate.org/>）下载了Hibernate最新的正式版本，并且已经打开了档案文件。你还需要在开发机器上安装Apache Ant。你还应该从<http://hsqldb.org/>下载HSQLDB的最新版本，并解压文件包；你将用这个数据库管理系统进行测试。如果你已经安装了另一个数据库管理系统，只要给它获得一个JDBC驱动程序即可。

不是你将在本书后面开发的完善应用，而是从“Hello World”例子开始。这样，你就可以关注开发过程，而不会被Hibernate的细节弄得心烦意乱。首先建立项目目录。

### 1. 创建工作目录

在系统上创建一个新目录，可以在你喜欢的任何位置，如果用的是Microsoft Windows，C:\helloworld是个好选择。后面的示例将把这个目录当作WORKDIR。创建lib和src子目录，并复制所有需要的库：

```
WORKDIR
+lib
  antlr.jar
  asm.jar
  asm-attrs.jar
  c3p0.jar
  cglib.jar
  commons-collections.jar
  commons-logging.jar
  dom4j.jar
  hibernate3.jar
  hsqldb.jar
  jta.jar
+src
```

你在lib目录中见到的这些库来自Hibernate发行包，一个典型的Hibernate项目需要这些库中的大部分。hsqldb.jar文件来自HSQLDB发行包；如果要使用一个不同的数据库管理系统，就用一个不同的驱动程序JAR代替。记住，你在此处见到的这些库，其中有些对于你正在使用的Hibernate的特定版本来说可能不需要，它可能比我们撰写本书时用过的版本更新。为了确定你具备了正确的程序库，要始终检查Hibernate发行包中的lib/README.txt文件。这个文件包含所有必需的和可选的第三方程序库的一个最新列表——你只需要列出运行时所需要的库。

在这个“Hello World”应用程序中，你想要把消息保存在数据库中，并从数据库中加载消息。要给这个业务案例创建领域模型。

### 2. 创建领域模型

Hibernate应用程序定义了被映射到数据库表的持久化类。你以业务领域的分析为基础定义这些

类；因此，它们是一个领域模型。“Hello World”示例由一个类和它的映射组成。我们来看一下简单的持久化类是什么样子，如何创建映射，以及可以用Hibernate中的持久化类的实例完成哪些事情。

这个例子的目标是在一个数据库中存储消息，并获取消息用来显示。你的应用程序有一个简单的持久化类Message，它表示这些可打印的消息。代码清单2-1显示了Message类。

### 代码清单2-1 Message.java: 一个简单的持久化类

```
package hello;

public class Message {
    private Long id;           标识符属性
    private String text;        消息文本
    private Message nextMessage; 另一个Message
                                对象的引用

    Message() {}

    public Message(String text) {
        this.text = text;
    }

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

    public String getText() {
        return text;
    }
    public void setText(String text) {
        this.text = text;
    }

    public Message getNextMessage() {
        return nextMessage;
    }
    public void setNextMessage(Message nextMessage) {
        this.nextMessage = nextMessage;
    }
}
```

Message类有3个属性：标识符属性、消息文本和另一个Message对象的引用。标识符属性允许应用程序访问持久化对象的数据库同一性——主键值。如果Message的两个实例具有相同的标识符值，它们在数据库中就表示相同的行。

这个例子对标识符属性的类型使用Long，但这不是必要条件。事实上Hibernate允许给标识符类型使用任何东西，稍后你会见到。

你可能已经注意到Message类的所有属性都有JavaBeans风格的属性访问方法。类也有一个无参构造函数。本例中介绍的持久化类看起来通常有点像这样。无参构造函数是个必要条件（像Hibernate这样的工具在这个构造函数上使用反射来实例化对象）。

Message类的实例可以由Hibernate管理（变成持久化），但是并非一定。因为Message对象不实现任何特定于Hibernate的类或者接口，可以像使用任何其他的Java类一样使用它：

```
Message message = new Message("Hello World");
System.out.println( message.getText() );
```

这个代码片段正确地完成了你对“Hello World”应用程序的期待：把Hello World打印到控制台。这看起来可能像是我们在这里自作聪明；事实上，我们正在示范一项重要的特性，它把Hibernate和一些其他的持久化方案区别开来。无论如何，持久化类都可以被用在任何执行环境中——不需要专门的容器。注意这也是新的JPA实体的好处之一，这些实体也是简单的Java对象。

把Message类的代码保存到你的源代码文件夹，该文件夹位于一个以hello命名的目录和包(package)中。

### 3. 映射类到数据库Schema

为了让对象/关系映射发挥神奇的魔力，Hibernate需要更多关于Message类究竟应该如何被持久化的信息。换句话说，Hibernate需要知道该类的实例要如何存储和加载。这个元数据可被写进一个XML映射文档，除了其他东西之外，这个文档还定义了Message类的属性如何映射到MESSAGES表的列。来看看代码清单2-2中的映射文档。

#### 代码清单2-2 简单的Hibernate XML映射

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
  "-//Hibernate/Hibernate Mapping DTD//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class
    name="hello.Message"
    table="MESSAGES">

    <id
      name="id"
      column="MESSAGE_ID">
      <generator class="increment"/>
    </id>

    <property
      name="text"
      column="MESSAGE_TEXT"/>

    <many-to-one
      name="nextMessage"
      cascade="all"
      column="NEXT_MESSAGE_ID"
      foreign-key="FK_NEXT_MESSAGE"/>

  </class>
</hibernate-mapping>
```

这个映射文档告诉Hibernate，Message类将被持久化到MESSAGES表，标识符属性映射到具名MESSAGE\_ID的列，text属性映射到具名MESSAGE\_TEXT的列，并且具名nextMessage的属性是一个关联，带有映射到具名NEXT\_MESSAGE\_ID的外键列的多对一多样性。Hibernate也为你生成数据库Schema，并添加名为FK\_NEXT\_MESSAGE的外键约束到数据库目录。（现在不用担心其他细节。）

这个XML文档不难理解，你可以很容易地手工编写和维护它。稍后，我们讨论一种直接在源代码中使用注解来定义映射信息的方法；但是无论你选择哪种方法，Hibernate都有足够的信息生成所有需要插入、更新、删除和获取Message类实例的SQL语句。你不再需要用手工编写这些SQL语句。

创建名为Message.hbm.xml的文件，包含代码清单2-2中所示的内容，并把它与源代码hello包中的Message.java文件放在一起。hbm的后缀是被Hibernate社区接受的一个命名约定，且大部分开发人员更喜欢把映射文件与它们领域类的源代码放在一起。

让我们在“Hello World”应用程序的主代码中加载和存储一些对象。

#### 4. 存储和加载对象

你阅读本书的真正目的是要学习Hibernate，因此让我们把一个新的Message保存到数据库（请见代码清单2-3）。

#### 代码清单2-3 “Hello World”的主应用程序代码

```
package hello;

import java.util.*;
import org.hibernate.*;
import persistence.*;

public class HelloWorld {

    public static void main(String[] args) {
        // First unit of work
        Session session =
            HibernateUtil.getSessionFactory().openSession();
        Transaction tx = session.beginTransaction();

        Message message = new Message("Hello World");
        Long msgId = (Long) session.save(message);

        tx.commit();
        session.close();

        // Second unit of work
        Session newSession =
            HibernateUtil.getSessionFactory().openSession();
        Transaction newTransaction = newSession.beginTransaction();

        List messages =
            newSession.createQuery("from Message m order by
                m.text asc").list();

        System.out.println( messages.size() +
            " message(s) found: " );
    }
}
```

```

        for ( Iterator iter = messages.iterator();
              iter.hasNext(); ) {
            Message loadedMsg = (Message) iter.next();
            System.out.println( loadedMsg.getText() );
        }

        newTransaction.commit();
        newSession.close();

        // Shutting down the application
        HibernateUtil.shutdown();
    }

}

```

把这段代码放在你项目的源代码文件夹中的HelloWorld.java文件中，在hello包中。我们来浏览一下这段代码。

类有一个标准的Java `main()`方法，可以直接从命令行调用它。在主应用程序代码内部，用Hibernate执行两个独立的工作单元。第一个单元存储一个新的Message对象，第二个单元加载所有对象并把它们的文本打印到控制台。

调用Hibernate Session、Transaction和Query接口访问数据库：

- Session（会话）——Hibernate Session集多种功能于一身。它是个单线程、非共享的对象，表示使用数据库的一个特定工作单元。它有持久化管理器API，调用它来加载和存储对象。（Session的内部由一列SQL语句组成，这些语句要与数据库在某个时点上进行同步，且托管持久化实例的一个映象由Session监控。）
- Transaction（事务）——这个Hibernate API可以用来编程式地设置事务范围，但它是可选的（事务范围不是可选的）。其他的选择还有JDBC事务划分、JTA接口，或者带有EJB的容器托管事务。
- Query（查询）——数据库查询可以写进Hibernate自己的面向对象的查询语言（HQL）或者简单的SQL中。这个接口允许你创建查询、在查询中绑定参数给占位符，并以各种方式执行查询。

先忽略调用`HibernateUtil.getSessionFactory()`的代码行——我们很快就会谈到它。

第一个工作单元，如果运行起来，会导致一些类似下列SQL的执行：

```
insert into MESSAGES (MESSAGE_ID, MESSAGE_TEXT, NEXT_MESSAGE_ID)
  values (1, 'Hello World', null)
```

等等，MESSAGE\_ID列正被初始化为一个奇怪的值。你没有在任何地方设置Message的id属性，因为希望它的值为NULL，对吗？事实上，id属性很特殊。它是一个标识符属性：它保存一个生成的唯一值。当调用`save()`时，这个值被Hibernate分配到Message实例。（我们稍后将讨论这个值如何生成。）

看看第二个工作单元。这个文字型字符串"from Message m order by m.text asc"是个Hibernate查询，用HQL表达。调用`list()`时，这个查询在内部被翻译成下列SQL：

```
select m.MESSAGE_ID, m.MESSAGE_TEXT, m.NEXT_MESSAGE_ID
  from MESSAGES m
```

```
order by m.MESSAGE_TEXT asc
```

如果运行这个main()方法（现在不要尝试——你仍需要配置Hibernate），控制台输出如下：

```
1 message(s) found:  
Hello World
```

如果以前从未使用过像Hibernate这样的ORM工具，你可能希望在代码或者映射元数据中的某个地方看到SQL语句，但是没有。所有SQL都在运行时生成（事实上，所有可重用的SQL语句都是在启动时生成）。

下一步一般是配置Hibernate。然而，如果你有信心，可以另外添加两个Hibernate特性——自动脏检查和级联——在第三个单元，通过给主应用程序添加下列代码：

```
// Third unit of work
Session thirdSession =
    HibernateUtil.getSessionFactory().openSession();
Transaction thirdTransaction = thirdSession.beginTransaction();

// msgId holds the identifier value of the first message
message = (Message) thirdSession.get( Message.class, msgId );

message.setText( "Greetings Earthling" );
message.setNextMessage(
    new Message( "Take me to your leader (please)" )
);

thirdTransaction.commit();
thirdSession.close();
```

这段代码在同一个数据库事务内部调用3个SQL语句：

```
select m.MESSAGE_ID, m.MESSAGE_TEXT, m.NEXT_MESSAGE_ID
from MESSAGES m
where m.MESSAGE_ID = 1

insert into MESSAGES (MESSAGE_ID, MESSAGE_TEXT, NEXT_MESSAGE_ID)
values (2, 'Take me to your leader (please)', null)

update MESSAGES
set MESSAGE_TEXT = 'Greetings Earthling', NEXT_MESSAGE_ID = 2
where MESSAGE_ID = 1
```

注意Hibernate如何侦测到text和第一条消息的nextMessage属性的修改，并自动更新数据库——Hibernate的确进行了自动脏检查。当你在一个工作单元内部修改对象状态时，这个特性为你免除了显式要求Hibernate去更新数据库的动作。类似地，当从第一条消息创建引用时，新消息被变成了持久化。这个特性称作级联保存（cascading save）。它为你免除了通过调用save()显式地使新对象变成持久化的动作，只要它可以通过一个已经持久化的实例获得。

还注意到SQL语句的排列与你设置属性值的顺序不同。Hibernate使用一个完善的运算法则来确定一种有效的顺序，避免违背数据库外键约束，但对于用户仍然足以预测。这个特性称作事务迟写（transactional write-behind）。

如果现在运行应用程序，会得到下列输出（你必须把第二个工作单元复制到第三个之后，再次执行查询-显示步骤）：

```
2 message(s) found:
Greetings Earthling
Take me to your leader (please)
```

现在你有了领域类、一个XML映射文件和一个加载和存储对象的“Hello World”应用程序代码。在可以编译和运行这段代码之前，需要创建Hibernate的配置（并揭开HibernateUtil类的神秘面纱）。

### 2.1.3 Hibernate 配置和启动

初始化Hibernate的常规方法是从一个Configuration对象中创建一个SessionFactory对象。如果你喜欢的话，可以把Configuration当作Hibernate配置文件（或者属性文件）的一个对象表示法。

在将其包装在HibernateUtil类中之前，先来看看这些变量。

#### 1. 构建SessionFactory

这是一个典型的Hibernate启动过程的例子，在一行代码中，使用自动的配置文件侦测：

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

等等，Hibernate怎么知道配置文件放置的位置，以及要加载哪个配置文件呢？

调用new Configuration()时，Hibernate在classpath的根目录下搜索名为hibernate.properties的文件。如果找到了，所有hibernate.\*属性都会被加载并添加到Configuration对象。

调用configure()时，Hibernate在classpath的根目录下搜索名为hibernate.cfg.xml的文件，如果无法找到就会抛出一个异常。当然，如果你没有这个配置文件，就不一定要调用这个方法。如果XML配置文件中的设置与较早设置的属性完全相同，XML设置就覆盖前面的设置。

hibernate.properties配置文件的位置始终在classpath的根目录中，处在任何包之外。如果要使用一个不同的文件，或者要Hibernate在classpath的子目录中查找XML配置文件，就必须把路径当作configure()方法的一个实参进行传递：

```
SessionFactory sessionFactory = new Configuration()
    .configure("/persistence/auction.cfg.xml")
    .buildSessionFactory();
```

最后，在创建SessionFactory之前，你始终可以在Configuration对象中编程式地设置额外的配置选项或者映射文件位置：

```
SessionFactory sessionFactory = new Configuration()
    .configure("/persistence/auction.cfg.xml")
    .setProperty(Environment.DEFAULT_SCHEMA, "CAVEATEMPTOR")
    .addResource("auction/CreditCard.hbm.xml")
    .buildSessionFactory();
```

许多配置源在这里得到应用：首先读取classpath中的hibernate.properties文件（如果有的话）。接下来，添加来自/persistence/auction.cfg.xml的所有设置，并覆盖以前应用的任何设置。最后，通过编程设置另一个配置属性（默认的数据库Schema名称），并且添加另一个Hibernate XML映射

元数据文件到配置中。

当然，可以通过编程设置所有选项，或者在不同的XML配置文件之间切换，得到不同的部署数据库。对于如何配置和部署Hibernate，实际上并没有限制；最后，只需要从一个准备好的配置中创建SessionFactory。

**说明** 方法链 (method chaining) ——方法链是一种编程风格，为许多Hibernate接口所支持。这种风格在Smalltalk中比在Java中更普及，有些人认为，比起更受认可的Java风格，它更不易阅读且更难以调试。然而，在许多情况下它很方便，例如用于本节中已经见过的配置片段。它是这样工作的：大部分Java开发人员声明设置方法 (setter method) 或者添加方法 (adder method) 为void类型，意味着它们不返回值；但是在Smalltalk中没有void类型，设置方法或者添加方法通常返回接收对象。我们在一些代码实例中使用这种Smalltalk风格，但是如果你不喜欢，就不必用它。如果你真的使用了这种编码风格，最好在一个不同的行中编写每一种方法调用。否则，你的调试器在这些代码中可能举步维艰。

既然了解Hibernate如何启动，以及如何创建SessionFactory，接下来要做什么？必须给Hibernate创建一个配置文件。

## 2. 创建XML配置文件

假设你要让事情保持简单，并且像大部分用户一样，你决定给包含所有配置细节的Hibernate使用单个XML配置文件。

建议你给新的配置文件使用默认名hibernate.cfg.xml，并把它直接放在项目的源代码目录中，处在任何包之外。这样，编译之后它就会在classpath的根目录中结束，并且Hibernate会自动找到它。看看代码清单2-4中的文件。

### 代码清单2-4 简单的Hibernate XML配置文件

```
<!DOCTYPE hibernate-configuration SYSTEM
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">
            org.hsqldb.jdbcDriver
        </property>
        <property name="hibernate.connection.url">
            jdbc:hsqldb:hsq://localhost
        </property>
        <property name="hibernate.connection.username">
            sa
        </property>
        <property name="hibernate.dialect">
            org.hibernate.dialect.HSQLDialect
        </property>
        <!-- Use the C3P0 connection pool provider -->
```

```

<property name="hibernate.c3p0.min_size">5</property>
<property name="hibernate.c3p0.max_size">20</property>
<property name="hibernate.c3p0.timeout">300</property>
<property name="hibernate.c3p0.max_statements">50</property>
<property name="hibernate.c3p0.idle_test_period">3000</property>

<!-- Show and print nice SQL on stdout -->
<property name="show_sql">true</property>
<property name="format_sql">true</property>

<!-- List of XML mapping files -->
<mapping resource="hello/Message.hbm.xml" />

</session-factory>
</hibernate-configuration>

```

文档类型声明 (Document Type Declaration, DTD) 被XML解析器用来针对Hibernate 配置DTD验证这个文件。注意这与用于Hibernate XML映射文件的那个DTD不一样。还要注意我们在属性值里面增加了一些换行，使它更易于阅读——在真正的配置文件中你不应该这么做（除非你的数据库用户名包含了换行）。

在配置文件中首当其冲的是数据库连接设置。你要告诉Hibernate用了哪种数据库JDBC驱动，以及如何通过URL、用户名和密码连接到数据库(此处省略了密码, 因为默认时HSQLDB不需要)。你设置了一个Dialect，因此Hibernate知道它必须生成哪个SQL变种来与数据库进行对话；Hibernate配有许多方言——查看一下Hibernate API文档中的清单。

在XML配置文件中，可以指定Hibernate属性而不用hibernate作为前缀，因此你可以写成hibernate.show\_sql或者只是show\_sql。除此之外，属性名称和值与编程式的配置属性是一致的——也就是说，与org.hibernate.cfg.Environment中定义的常量一致。例如，hibernate.connection.driver\_class属性具有常量Enviroment.DRIVER。

在探讨一些重要的配置选项之前，考虑配置中命名Hibernate XML映射文件的最后一行。在创建SessionFactory之前，Configuration对象需要知道你所有的XML映射文件。SessionFactory是给一组特定的映射元数据表示特定Hibernate配置的一个对象。可以在Hibernate XML映射文件中列出所有的XML映射文件，也可以在Configuration对象中编程式地设置它们的名称和路径。不管怎样，如果把它们当作一个资源 (resource) 列出，映射文件的路径在classpath中就是个相对位置，在这个实例中，hello是classpath根目录中的一个包。

你还启用了把所有Hibernate执行的SQL打印到控制台，并告诉Hibernate给它设置好格式，以便可以检查幕后到底发生了些什么。本章稍后会回到日志 (logging) 的话题。

另一个偶尔会派上用场的技巧，是使系统属性的配置选项更动态些：

```

...
<property name="show_sql">${displaysql}</property>
...

```

现在你可以在启动应用程序时，在命令行中指定一个系统属性，例如用java-displaysql=true，并且这将被自动应用到Hibernate配置属性上。

数据库连接池设置值得特别关注。

### 3. 数据库连接池

一般来说，不需要在每次要与数据库交互时都创建连接。反之，Java应用程序应该使用连接池（pool）。需要在数据库上工作的每个应用程序线程都从这个池中请求连接，然后当执行完所有SQL操作后把它返回到池中。这个池维护着连接，并使打开和关闭连接的成本减到最少。

使用连接池有3个原因：

- 获得新的连接很昂贵。有些数据库管理系统甚至给每个连接启动一个全新的服务器进程。
- 为数据库管理系统维护许多闲置的连接很昂贵，并且连接池可以最优化闲置连接的使用（或者没有请求时就断开连接）。
- 给某些驱动程序创建预编译的语句也很昂贵，且连接池可以对跨请求的连接高速缓存语句。

图2-2显示了连接池在非托管的应用程序运行时环境中的角色（即不要任何应用程序服务器）。

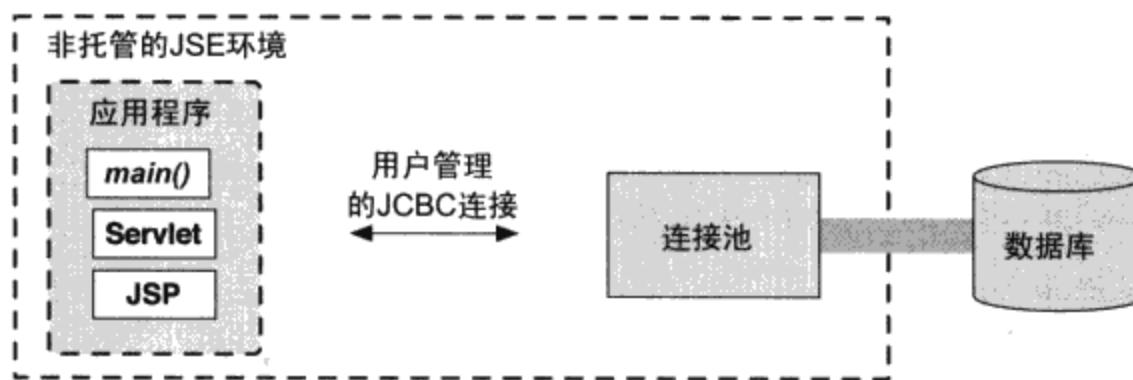


图2-2 非托管环境中的JDBC连接池

没有应用程序服务器提供连接池，应用程序要么实现它自己的池化算法，要么依赖于第三方的库（如开源C3P0连接池软件）。没有Hibernate，应用程序代码就调用连接池来获取JDBC连接，然后用JDBC编程接口执行SQL语句。当应用程序关闭SQL语句，且最后关闭连接时，预编译的语句和连接没有被释放，而是返回到池中。

使用Hibernate，这个图就变了：它充当JDBC连接池的一个客户端程序，如图2-3所示。应用程序代码给持久化操作使用Hibernate的Session和Query API，它（可能）用Hibernate的Transaction API管理数据库事务。

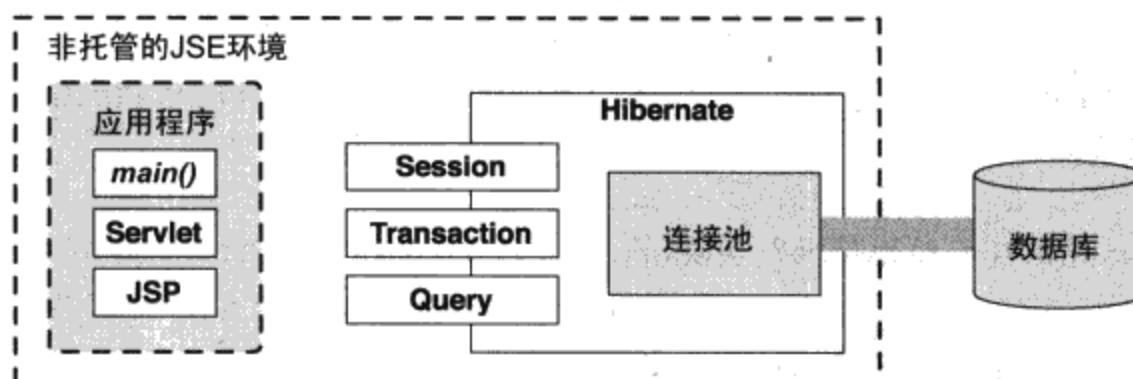


图2-3 非托管环境中带有连接池的Hibernate

Hibernate定义了一个插件架构，允许与任何连接池软件整合。然而，对C3P0的支持是内建的，并且这个软件与Hibernate捆绑在一起，因此你将会用到它（你已经复制了c3p0.jar文件到库目录下，对吗？）。Hibernate替你维护连接池，并且传递配置属性。如何通过Hibernate配置C3P0呢？

配置连接池的一种方法是把设置放进hibernate.cfg.xml配置文件，就像前一节中所做的那样。

另一种方法是，可以在应用的classpath根目录中创建一个hibernate.properties文件。C3P0的hibernate.properties文件的一个实例如代码清单2-5所示。注意这个文件，除了映射资源清单之外，相当于代码清单2-4中所示的配置。

#### 代码清单2-5 给C3P0连接池设置使用hibernate.properties

```
hibernate.connection.driver_class = org.hsqldb.jdbcDriver
hibernate.connection.url = jdbc:hsqldb:hsq1://localhost
hibernate.connection.username = sa
hibernate.dialect = org.hibernate.dialect.HSQLDialect
① hibernate.c3p0.min_size = 5 ← ②
hibernate.c3p0.max_size = 20 ← ③
hibernate.c3p0.timeout = 300 ← ④
hibernate.c3p0.max_statements = 50 ←
hibernate.c3p0.idle_test_period = 3000 ← ⑤
hibernate.show_sql = true
hibernate.format_sql = true
```

- ① 这是C3P0随时做好准备的最少的JDBC连接数量。
- ② 这是池中最多的连接数量。如果超出这个数量，就会在运行时抛出异常。
- ③ 指定超时周期（在本例中为300秒），在它之后闲置连接将从池中移除。
- ④ 最多高速缓存50个预编译的语句。高速缓存预编译的语句是利用Hibernate获得最好性能的要素。
- ⑤ 这是在连接被自动验证之前，一个以秒为单位的闲置时间。

指定表单hibernate.c3p0.\*的属性选择C3P0作为连接池（c3p0.max\_size选项是必需的——你无需任何其他的切换来启用C3P0支持）。C3P0有着比前一个例子所展示的更多特性；参考Hibernate发行包的etc/子目录中的属性文件，找到一个可以从中复制的综合示例。

类org.hibernate.cfg.Environment的Javadoc也为每一个Hibernate配置属性提供文档。甚至，可以在Hibernate参考文档中找到一个带有全部Hibernate配置选项的最新表格。然而，本书通篇都将阐述最重要的设置。现在你已经知道了在开始之前需要的一切。

**常见问题** 可以提供自己的连接吗？实现org.hibernate.connection.ConnectionProvider接口，用hibernate.connection.provider\_class配置选项命名你的实现。现在，如果Hibernate需要数据库连接，它将依赖于你定制的提供程序（provider）。

既然已经完成了Hibernate配置文件，你就可以继续前进，在应用程序中创建SessionFactory了。

#### 4. 处理SessionFactory

在大部分Hibernate应用程序中，SessionFactory应该在应用程序初始化期间被实例化一次。然后单独的实例应该为特定程序中的所有代码所用，任何Session都应该用这个单独的SessionFactory来创建。SessionFactory是线程安全的，且能够被共享；Session是个单线程的对象。

一个经常被问及的问题是，这个工厂创建后应该被存储在什么地方，以及它如何被轻松地访问。有更高级而舒适的选项如JNDI和JMX，但是它们通常只能在完全的Java EE应用程序服务器中才能使用。反之，我们将引入一个实用、快速的解决方案，解决Hibernate启动的问题（那一行代码）和SessionFactory的存储与访问：你将使用一个静态的全局变量和静态的初始化。

变量和初始化这两者都可以在单个类中实现，这个类将称作HibernateUtil。这个辅助类在Hibernate社区中大名鼎鼎——在没有Java EE服务的普通Java应用程序中，它是Hibernate启动的一种常用模式。代码清单2-6中显示了一个基本的实现。

**代码清单2-6** HibernateUtil类用于启动而SessionFactory用于处理

```
package persistence;

import org.hibernate.*;
import org.hibernate.cfg.*;

public class HibernateUtil {

    private static SessionFactory sessionFactory;

    static {
        try {
            sessionFactory=new Configuration()
                .configure()
                .buildSessionFactory();
        } catch (Throwable ex) {
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        // Alternatively, you could look up in JNDI here
        return sessionFactory;
    }

    public static void shutdown() {
        // Close caches and connection pools
        getSessionFactory().close();
    }
}
```

创建一个静态初始化器代码块来启动Hibernate：这个块正好被这个类的加载器(loader)在初始化中执行一次，在该类被加载的时候。应用程序中的HibernateUtil的第一次调用加载类，创建SessionFactory，同时设置静态变量。如果出现问题，任何Exception或者Error就都被

包起来并抛到静态的块之外（这就是你为什么要捕捉Throwable的原因）。ExceptionInInitializerError中的包装对于静态的初始化器来说是强制的。

你已经在一个名为persistence的新包中创建了这个新的类。在一个全特性的Hibernate应用程序中，经常需要这样一个包——例如，把你定制的持久层拦截器和数据类型转换器包成基础结构的一部分。

现在，每当你需要在应用程序中访问一个Hibernate Session时，就可以轻松地用HibernateUtil.getSessionFactory().openSession()获得，就像稍早你在HelloWorld主应用代码中所做的那样。

你差不多准备运行和测试应用程序了。但是因为你必定想要知道幕后发生了什么，因此首先启用日志。

## 5. 启用日志和统计

你已经见过了hibernate.show\_sql配置属性。当你用Hibernate开发软件时，将会频繁地需要用到它；它把所有生成的SQL的日志启用到控制台。你将用它解决问题、性能调优，及查看正在进行的工作。如果还启用了hibernate.format\_sql，输出就会更易阅读，但会占用更多的屏幕空间。目前为止你还没有设置的第三个选项是hibernate.use\_sql\_comments——它导致Hibernate把注释放在所有生成的SQL语句内部，来提示它们的出处。例如，你可以轻松地看出某个特定的SQL语句是从显式的查询还是所需的集合初始化中生成的。

启用SQL输出为stdout只是你的第一个日志选项。Hibernate（和许多其他的ORM实现）异步地执行SQL语句。当应用程序调用session.save()时，通常不执行INSERT语句；或者当应用程序调用item.setPrice()时，也不会立即发布UPDATE。反之，SQL语句通常在事务结束时发布。

这意味着追踪和调试ORM代码有时候不容易。理论上，应用程序可能把Hibernate当作黑盒处理，并忽视这个行为。然而，当正在解决一道难题时，需要能够准确地看到Hibernate内部正在做什么。由于Hibernate是开源的，你可以轻松地进入Hibernate代码，并且这偶尔还能帮上很大的忙！经验丰富的Hibernate专家仅仅通过查看Hibernate日志和映射文件来调试问题；我们鼓励你在由Hibernate生成的日志输出上花些时间，并熟悉一下它的内部情况。

Hibernate通过Apache通用日志工具（commons-logging）这个指示输出到Apache Log4j（如果把log4j.jar放在classpath中）或者JDK 1.4 logging（如果你正在JDK 1.4或者其更高的版本上运行，且没有Log4j的话）很薄的抽象层，对所有值得关注的事件都作了日志记录。推荐Log4j，因为它更成熟、更普及，并且在更积极的开发之中。

为了从Log4j中看到输出，你需要classpath中名为log4j.properties的文件（就在hibernate.properties或者hibernate.cfg.xml旁边）。而且，别忘记复制log4j.jar库到lib目录下。代码清单2-7中的Log4j配置示例指示所有的日志消息到控制台。

### 代码清单2-7 log4j.properties配置文件的一个示例

```
# Direct log messages to stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
```

```

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE}
    %5p %c{1}:%L - %m%n

# Root logger option
log4j.rootLogger=INFO, stdout

# Hibernate logging options (INFO only shows startup messages)
log4j.logger.org.hibernate=INFO
# Log JDBC bind parameter runtime arguments
log4j.logger.org.hibernate.type=INFO

```

这个配置文件中的最后一个类特别值得关注：如果把它设置成DEBUG等级，就会启用JDBC绑定参数的日志，提供通常你在特别的SQL控制台日志中看不到的信息。更全面的示例，请查阅捆绑在Hibernate发行包的etc/目录中的log4j.properties文件，也看一下Log4j文档了解更多信息。注意，在产品运行时永远不要给DEBUG级别下的任何东西记录日志，因为这么做会严重影响应用程序的性能。

也可以通过启用实时统计来监控Hibernate。没有应用程序服务器（也就是说，如果你没有JMX部署环境），运行时从Hibernate引擎之外获取统计的最容易的方法就是SessionFactory：

```

Statistics stats =
    HibernateUtil.getSessionFactory().getStatistics();

stats.setStatisticsEnabled(true);
...

stats.getSessionOpenCount();
stats.logSummary();

EntityStatistics itemStats =
    stats.getEntityStatistics("auction.model.Item");
itemStats.getFetchCount();

```

统计接口中，Statistics用于全局的信息，EntityStatistics用于有关一个特定实体的信息，CollectionStatistics用于一个特定的集合任务，QueryStatistics用于SQL和HQL查询，SecondLevelCacheStatistics用于详细的关于可选的二级数据高速缓存中的一个特定区域的运行时信息。一种方便的方法是logSummary()，它通过单个调用把一个完整的摘要打印到控制台。如果想要通过配置来启用统计集合，并且不用编程，就设置hibernate.generate\_statistics配置属性为true。有关各种统计获取方法的更多信息请见API文档。

在运行“Hello World”应用程序之前，检查一个工作目录是否具备所有必需的文件：

```

WORKDIR
build.xml
+lib
<all required libraries>
+src
+hello
    HelloWorld.java
    Message.java
    Message.hbm.xml

```

```
+persistence
    HibernateUtil.java
    hibernate.cfg.xml (or hibernate.properties)
    log4j.properties
```

第一个文件build.xml是一个Ant构建定义。它包含构建和运行应用程序的Ant目标，这个我们接下来会讨论。你还将添加一个可以自动生成数据库Schema的任务。

### 2.1.4 运行和测试应用程序

要运行应用程序，首先要对它进行编译，并通过正确的数据库Schema启动数据库管理系统。Ant是Java的一个强大的构建系统。一般要给项目编写build.xml文件，并用Ant命令行工具调用你在这个文件中定义的构建任务。如果支持的话，也可以从Java IDE中调用Ant任务。

#### 1. 用Ant编译项目

现在要给“Hello World”项目添加build.xml文件和一些任务。构建文件的初始内容如代码清单2-8所示——直接在WORKDIR中创建这个文件。

**代码清单2-8 “Hello World”的一个基础的Ant构建文件**

```
<project name="HelloWorld" default="compile" basedir=".">
    <!-- Name of project and version -->
    <property name="proj.name"      value="HelloWorld"/>
    <property name="proj.version"   value="1.0"/>

    <!-- Global properties for this build -->
    <property name="src.java.dir"   value="src"/>
    <property name="lib.dir"        value="lib"/>
    <property name="build.dir"     value="bin"/>

    <!-- Classpath declaration -->
    <path id="project.classpath">
        <fileset dir="${lib.dir}">
            <include name="**/*.jar"/>
            <include name="**/*.zip"/>
        </fileset>
    </path>

    <!-- Useful shortcuts -->
    <patternset id="meta.files">
        <include name="**/*.xml"/>
        <include name="**/*.properties"/>
    </patternset>

    <!-- Clean up -->
    <target name="clean">
        <delete dir="${build.dir}"/>
        <mkdir dir="${build.dir}"/>
    </target>

    <!-- Compile Java source -->
    <target name="compile" depends="clean">
        <mkdir dir="${build.dir}"/>
```

```

<javac
    srcdir="${src.java.dir}"
    destdir="${build.dir}"
    nowarn="on">
    <classpath refid="project.classpath"/>
</javac>
</target>

<!-- Copy metadata to build classpath -->
<target name="copymetafiles">
    <copy todir="${build.dir}">
        <fileset dir="${src.java.dir}">
            <patternset refid="meta.files"/>
        </fileset>
    </copy>
</target>

<!-- Run HelloWorld -->
<target name="run" depends="compile, copymetafiles"
    description="Build and run HelloWorld">
    <java fork="true"
        classname="hello.HelloWorld"
        classpathref="project.classpath">
        <classpath path="${build.dir}" />
    </java>
</target>

</project>

```

这个Ant构建文件的前半部分包含属性设置，例如项目名称以及文件和目录的全局位置。你可能已经发现这个构建是以现有的目录结构(即你的WORKDIR)为基础的(对于Ant, 这与basedir是相同目录)。当用没有具名的任务调用这个构建文件时, default的目标就是compile。

接下来, 定义一个以后能够很容易引用的名称project.classpath, 作为项目的库目录中所有库的一个快捷操作。另一个很方便的的快捷样式定义为meta.files。你在构建过程中, 需要使用这个过滤器分开处理配置和元数据文件。

clean任务移除了所有创建的和编译的文件, 并清理了项目。最后3个任务compile、copymetafiles和run, 应该不言自明的。根据所有Java源文件的编译运行应用程序, 并复制所有映射和属性配置文件到构建目录下。

现在, 在WORKDIR中执行ant compile, 来编译“Hello World”应用。编译期间你应该看不到错误(也没有任何警告), 而在bin目录中可以找到编译好的类文件。也调用ant copymetafiles一次, 检查是否所有的配置和映射文件都正确地复制到了bin目录。

在运行应用之前, 启动数据库管理系统, 并导出一个新的数据库Schema。

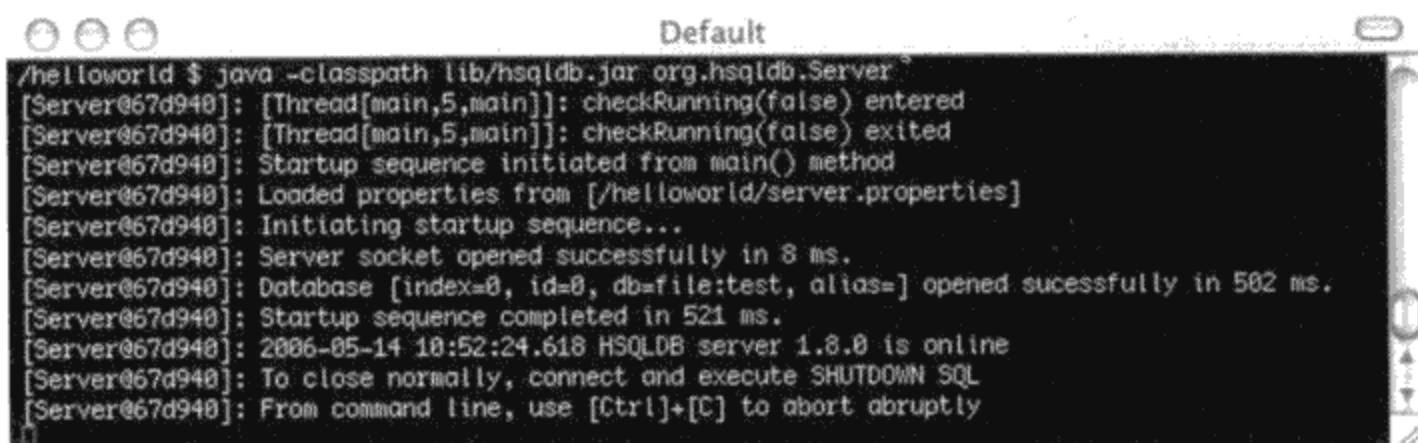
## 2. 启动HSQL数据库系统

Hibernate支持25个以上预设好的SQL数据库管理系统, 并支持任何可被轻松添加的未知方言。如果你有一个现成的数据库, 或者如果你了解基础的数据库管理, 也可以用喜欢的系统设置取代以前创建的配置选项(主要是连接和方言设置)。

要向世界说hello, 你需要一个易于安装和配置的轻量级、实用的数据库系统。HSQLDB是个

不错的选择，它是用Java编写的一个开源的SQL数据库管理系统。它可以与主应用程序同步运行，但是依我们的经验，用一个TCP端口监听给连接独立运行HSQLDB通常更方便。你已经把hsqldb.jar文件复制到WORKDIR的库目录中——这个库包括了连接到运行示例所需的数据库引擎和JDBC驱动程序。

要启动HSQLDB服务器，得打开一个命令行，切换到你的WORKDIR，并运行图2-4所示的命令。你应该看到启动消息，最后是一个教你如何关闭数据库系统的帮助消息（用Ctrl+C也可以）。你还将在WORKDIR中看到一些新文件，从test开始——这些是HSQLDB用来给你存储数据的文件。如果想要从一个新的数据库开始，就在服务器重启之间删除这些文件。



```

Default
/helloworld $ java -classpath lib/hsqldb.jar org.hsqldb.Server
[Server@67d940]: [Thread[main,5,main]]: checkRunning(false) entered
[Server@67d940]: [Thread[main,5,main]]: checkRunning(false) exited
[Server@67d940]: Startup sequence initiated from main() method
[Server@67d940]: Loaded properties from [/helloworld/server.properties]
[Server@67d940]: Initiating startup sequence...
[Server@67d940]: Server socket opened successfully in 8 ms.
[Server@67d940]: Database [index=0, id=0, db=file:test, alias=] opened sucessfully in 502 ms.
[Server@67d940]: Startup sequence completed in 521 ms.
[Server@67d940]: 2006-05-14 10:52:24.618 HSQLDB server 1.8.0 is online
[Server@67d940]: To close normally, connect and execute SHUTDOWN SQL
[Server@67d940]: From command line, use [Ctrl]+[C] to abort abruptly

```

图2-4 从命令行启动HSQLDB服务器

现在有了一个没有内容甚至也没有Schema的空数据库。接下来要创建Schema。

### 3. 导出数据库Schema

可以用CREATE语句手工编写SQL DDL，并在数据库中执行这个DDL，来创建数据库Schema。或者（且这是更方便的做法）可以让Hibernate完成这项工作，并给你的应用程序创建一个默认的Schema。在Hibernate中自动生成SQL DDL的先决条件始终是Hibernate映射元数据定义，或者在XML映射文件中，或者在Java源代码的注解中。假设你已经随着前几节设计和实现了领域模型类，并用XML编写了映射元数据。

用来生成模式的工具是hbm2ddl；它的类是org.hibernate.tool.hbm2ddl.SchemaExport，因此它有时也称作SchemaExport。

运行这个工具并创建一个Schema有3种方法：

- 可以在常规的构建过程中，在一个Ant任务中运行<hbm2ddl>。
- 可以在应用程序代码中用程序运行SchemaExport，可能在HibernateUtil启动类中。然而，这不常用，因为Schema的生成很少需要编程式的控制。
- 当通过设置Hibernate.hbm2ddl.auto配置属性为create或者create-drop来创建SessionFactory时，可以启用Schema的自动导出。第一个设置在创建SessionFactory时产生DROP语句，之后产生CREATE语句。第二个设置在应用关闭时添加另外的DROP语句，同时关闭SessionFactory——实际上在每一次运行之后留下了一个干净的数据库。

编程式的生成模式很简单：

```
Configuration cfg = new Configuration().configure();
SchemaExport schemaExport = new SchemaExport(cfg);
schemaExport.create(false, true);
```

从一个Configuration中创建一个新的SchemaExport对象；所有设置（例如数据库驱动程序、连接URL等）都传递到SchemaExport构造函数。create (false, true) 调用触发DDL生成程序，没有任何SQL输出到stdout（因为false设置），但DDL立即在数据库中执行（true）。更多信息和其他的设置请见SchemaExport API。

你的开发过程决定了是否应该用hibernate.hbm2ddl.auto配置设置启用自动的Schema导出。许多Hibernate新用户发现SessionFactory中的自动删除和重新创建有点费解。一旦你进一步熟悉Hibernate之后，我们鼓励你去探索这个选项，以便在集成测试中加快周转时间。

这个配置属性的另一个选项是update，开发期间会很有用：它启用内建的SchemaUpdate工具，使得Schema演变更加容易。如果启用，Hibernate就在启动时读取JDBC数据库元数据，并通过把旧的Schema和当前的映射元数据相比较，来创建新的表和约束。注意这项功能取决于JDBC驱动程序提供的元数据的特性，一个缺乏许多驱动程序的领域。因此，这个特性在实际应用中并没有它听起来的那么令人激动和有用。

---

**警告** 我们见过Hibernate用户尝试用SchemaUpdate自动更新一个产品数据库的Schema。这可能很快带来灾难，并且你的数据库管理员（DBA）也不会允许。

---

也可以编程式地运行SchemaUpdate：

```
Configuration cfg = new Configuration().configure();
SchemaUpdate schemaUpdate = new SchemaUpdate(cfg);
schemaUpdate.execute(false);
```

最后再次设置false使SQL DDL不能输出到控制台，并且仅仅在数据库中直接执行语句。如果导出DDL到控制台或者一个文本文件，你的DBA可能会用它作为产生一个优质Schema演变脚本的起点。

开发中有用的另一个hibernate.hbm2ddl.auto设置是validate。它使SchemaValidator能够在启动时运行。这个工具可以把映射和JDBC数据库进行比较，并告诉你这个Schema和映射是否匹配。也可以编程式地运行SchemaValidator：

```
Configuration cfg = new Configuration().configure();
new SchemaValidator(cfg).validate();
```

如果发现映射和数据库Schema之间不匹配，就会抛出异常。

因为正把系统构建在Ant之上，理想的情况是，每当你需要Schema时，就给Ant构建添加一个schemaexport目标，它从数据库中生成并导出一个新的Schema（请见代码清单2-9）。

### 代码清单2-9 Schema导出的Ant目标

```
<taskdef name="hibernatetool"
        classname="org.hibernate.tool.ant.HibernateToolTask"
```

```

        classpathref="project.classpath"/>

<target name="schemaexport" depends="compile, copymetafiles"
description="Exports a generated schema to DB and file">
    <hibernatetool destdir="${basedir}">
        <classpath path="${build.dir}" />

        <configuration
            configurationfile="${build.dir}/hibernate.cfg.xml"/>

        <hbm2ddl
            drop="true"
            create="true"
            export="true"
            outputfilename="helloworld-ddl.sql"
            delimiter=";"
            format="true" />

    </hibernatetool>
</target>

```

在这个目标中，首先定义了一个你想要使用的新Ant任务HibernateToolTask。这个一般的任务能够完成很多工作——从Hibernate映射元数据中导出一个SQL DDL Schema只是其中的一项工作。你将在本章的所有Ant构建中自始至终使用它。确保在任务定义的classpath中包括了所有Hibernate库、必要的第三方库和JDBC驱动程序。还需要添加hibernate-tools.jar文件，可以在Hibernate Tools下载包中找到它。

Ant目标schemaexport使用这个任务，它还取决于构建目录中编译好的类和复制好的配置文件。`<hibernatetool>`任务的基本用法始终是一样的：配置是所有代码工件（artifact）生成的起点。此处显示的变种`<configuration>`，它理解Hibernate XML配置文件，并读取给定配置中列出的所有Hibernate XML映射元数据文件。从这些信息中，生成了一个内部的Hibernate元数据模型（即hbm在各处所代表的含义），导出器随后处理这个模型数据。本章稍后讨论能够读取注解或者反向工程数据库的工具配置。

目标中的另一个元素是所谓的导出器（exporter）。工具配置传递它的元数据信息给你选中的导出器；在前面的例子中，指`<hbm2ddl>`导出器。就像你可能已经猜到的，这个导出器理解Hibernate元数据模型，并生成SQL DDL。你可以用几种方法控制DDL生成：

- 导出器生成SQL，因此在Hibernate配置文件中设置一种SQL方言是强制的。
- 如果drop设置为true，将首先生成SQL DROP语句，并且所有的表和约束都被移除，如果有的话。如果create设置为true，接下来就生成SQL CREATE语句，来创建所有的表和约束。如果这两个选项你都启用了，实际上就在Ant目标的每一次运行中删除和重新创建数据库Schema。
- 如果export设置为true，所有的DDL语句就直接在数据库中被执行。导出器使用在配置文件中找到的连接设置打开一个和数据库的连接。
- 如果有outputfilename，所有的DDL语句就被写到这个文件，并且文件保存在你配置的

destdir中。附加delimiter字符给写到这个文件的所有SQL语句，并且如果format是启用的，所有SQL语句都会被很好地缩进。

现在通过在WORKDIR中运行ant schemaexport，可以生成、输出并直接把Schema导出到一个文本文件和数据库。所有的表和约束都被删除，然后被再次创建，你就有了一个新的数据库。（忽略任何报告所称的因为表不存在而不能被删除的错误消息。）

检查你的数据库是否在运行，以及它是否有正确的数据库Schema。HSQLDB包括的一个有用工具是一个简单的数据库浏览器。可以用下列Ant目标调用它：

```
<target name="dbmanager" description="Start HSQLDB manager">
    <java
        classname="org.hsqldb.util.DatabaseManagerSwing"
        fork="yes"
        classpathref="project.classpath"
        failonerror="true">
        <arg value="-url"/>
        <arg value="jdbc:hsqldb:hsq://localhost://" />
        <arg value="-driver"/>
        <arg value="org.hsqldb.jdbcDriver"/>
    </java>
</target>
```

登录以后应该看到图2-5所示的Schema。

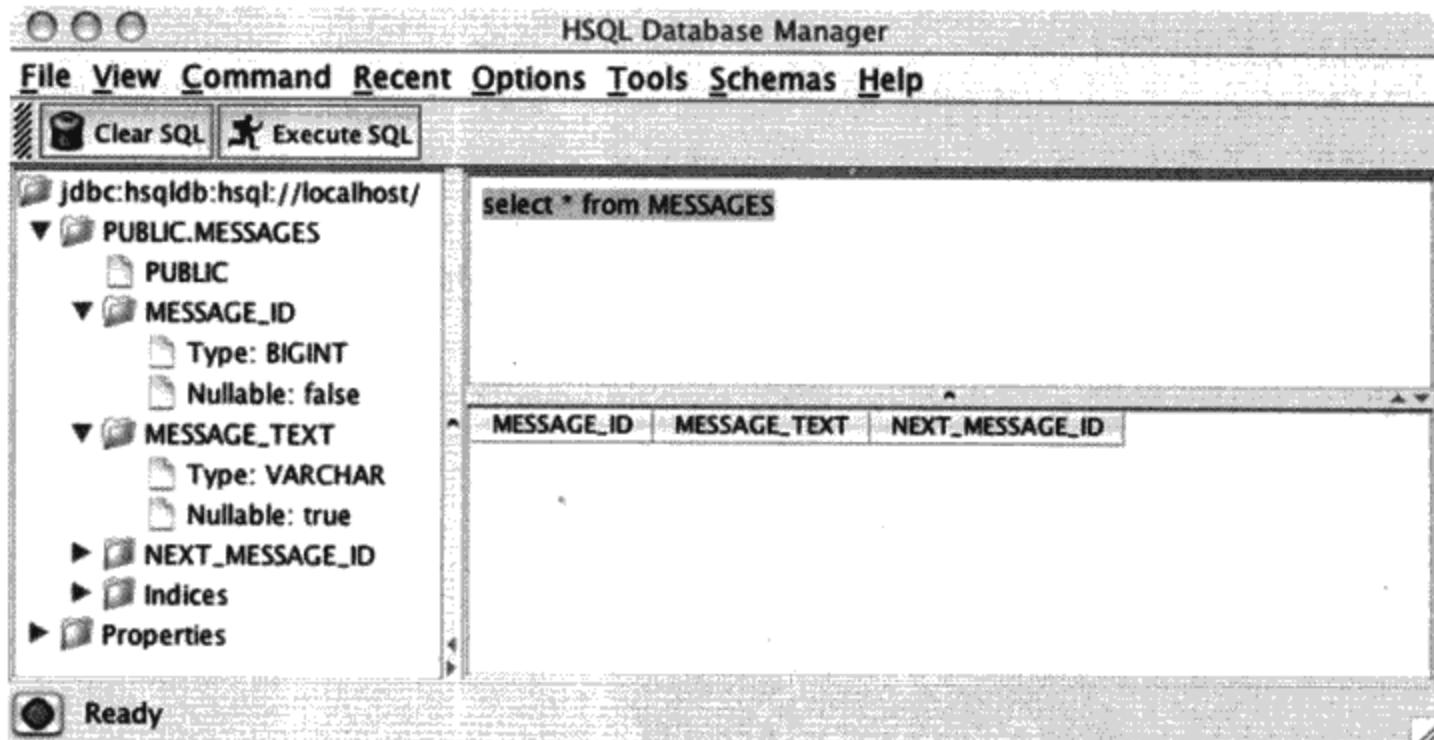


图2-5 HSQLDB浏览器和SQL控制台

用ant run运行应用程序，并观察控制台的Hibernate日志输出。应该看到你的消息被存储、加载和打印了。在HSQLDB浏览器中启动一个SQL查询，直接检查你的数据库内容。

现在有了一个有效的Hibernate基础结构和Ant项目构建。可以跳到下一章，继续编写和映射更复杂的业务类。然而，建议你在“Hello World”应用上花些时间，并且用更多的功能对它进行

扩展。例如，可以尝试不同的HQL查询或者日志选项。别忘了数据库系统仍然在后台运行着，你必须导出一个新的Schema或者停止并删除数据库文件，以便重新得到一个干净的空数据库。

在下一节中，我们用Java Persistence接口和EJB 3.0重新创建一遍这个“Hello World”示例。

## 2.2 启动 Java Persistence 项目

接下来的几节介绍JPA和新EJB 3.0标准的一些优势，以及注解和标准的编程接口如何简化应用程序开发，甚至与Hibernate比较。显然，如果你不时需要在不同的运行时环境下移植或者部署应用程序，设计和链接到标准的接口就是一个优势。尽管除了可移植性之外，还有许多很好的理由来深入探讨JPA。

现在将引导你进行另一个“Hello World”示例，这次用Hibernate Annotations和Hibernate EntityManager。你将重用在前一节中引入的基础项目架构，以便可以看到JPA与Hibernate的区别之处。介绍完注解和JPA接口之后，将介绍应用如何与其他的托管组件（EJB）整合和交互。本书稍后将讨论更多的应用设计示例；然而，这第一眼就将让你尽快决定一种特定的方法。

### 2.2.1 使用 Hibernate Annotations

首先利用Hibernate Annotations，用内嵌元数据替代Hibernate XML映射文件。你可能想要在进行下列变化之前复制现有的“Hello World”项目目录——从原生的Hibernate迁移到标准的JPA映射（且稍后迁移到程序代码）。

复制Hibernate Annotations库到WORKDIR/lib目录——查看Hibernate Annotations文档中的必需的库的清单。（在编写本书之时，hibernate-annotations.jar和ejb3-persistence.jar中的API存根是必需的。）

现在删除src/hello/Message.hbm.xml文件。下面用src/hello/Message.java类源代码中的注解来替代这个文件，如代码清单2-10所示。

**代码清单2-10 用注解映射Message类**

```
package hello;

import javax.persistence.*;

@Entity
@Table(name = "MESSAGES")
public class Message {

    @Id @GeneratedValue
    @Column(name = "MESSAGE_ID")
    private Long id;

    @Column(name = "MESSAGE_TEXT")
    private String text;

    @ManyToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "NEXT_MESSAGE_ID")
```

```

private Message nextMessage;
private Message() {}
public Message(String text) {
    this.text = text;
}
public Long getId() {
    return id;
}
private void setId(Long id) {
    this.id = id;
}
public String getText() {
    return text;
}
public void setText(String text) {
    this.text = text;
}
public Message getNextMessage() {
    return nextMessage;
}
public void setNextMessage(Message nextMessage) {
    this.nextMessage = nextMessage;
}
}

```

你在这个更新过的业务类中可能注意到的第一点是javax.persistence接口的引入。这个包内部是映射@Entity类到数据库@Table所需要的所有标准的JPA注解。你把注解放在类的私有字段上，数据库标识符映射从@Id和@GeneratedValue开始。这个JPA持久化的提供程序发现@Id注解在一个字段中，并假设它应该在运行时直接通过字段在对象中访问属性。如果把@Id注解放在getId()方法中，默认时将通过获取方法(getter method)和设置方法启用属性访问。因此，所有其他的注解遵循选中的策略，也被放在字段或者获取方法中。

注意不需要@Table、@Column和@JoinColumn注解。一个实体的所有属性自动被认为是持久化的，使用默认的策略和表/列名称。为了清楚起见，你在此处添加它们，并得到与使用XML映射文件相同的结果。现在比较这两种映射元数据的策略，你会发现注解更加方便，并且明显减少了元数据的行数。注解也是类型安全(type-safe)的，当你输入时它们在IDE中支持自动完成(就像任何其他Java接口一样)，并且它们使类和属性的重构更加容易。

如果担心JPA接口的导入将把你的代码绑定到这个包，就应该了解仅当注解被Hibernate在运行时使用的时候，才需要它处在你的classpath中。可以加载和执行这个类，而不用classpath上的JPA接口，只要你不想用Hibernate加载和存储实例。

刚接触注解的开发人员的第二个关注点，有时候与Java源代码中包含配置元数据的内容有关。按定义，配置元数据是可以随应用的每一次部署而改变的元数据，例如表名。JPA有一个简单的解决方案：你可以覆盖或者替代所有用XML元数据文件注解的元数据。本书稍后将介绍如

何实现这个解决方案。

假设这就是你想从JPA中得到的所有东西——用注解取代XML。你不想使用JPA编程接口或者查询语言，你将使用**Hibernate Session**和**HQL**。唯一需要对项目所做的其他改变，除了删除目前没用的XML映射文件之外，就是**hibernate.cfg.xml**中**Hibernate**配置的一个变化：

```
<!DOCTYPE hibernate-configuration SYSTEM
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
<session-factory>
    <!-- ... Many property settings ... -->
    <!-- List of annotated classes-->
    <mapping class="hello.Message"/>
</session-factory>
</hibernate-configuration>
```

**Hibernate**配置文件之前已经有一个包含所有XML映射文件的清单。这已经被一个包含所有被注解的类清单所替代。如果使用**SessionFactory**编程式的配置，**addAnnotatedClass()**方法就会替代**addResource()**方法：

```
// Load settings from hibernate.properties
AnnotationConfiguration cfg = new AnnotationConfiguration();
// ... set other configuration options programmatically
cfg.addAnnotatedClass(hello.Message.class);

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

注意现在你已经用**AnnotationConfiguration**代替了基本的**Hibernate**的**Configuration**接口——这个扩展识别被注解的类。至少，你还需要改变**HibernateUtil**中的初始化器以便使用这个接口。如果用一个**Ant**目标导出数据库Schema，那么就在**build.xml**文件中用**<annotationconfiguration>**取代**<configuration>**。

这就是用注解运行“Hello World”应用程序需要改变的所有内容。试着再次运行它，可能通过一个全新的数据库。

注解元数据也可以是全局的，虽然“Hello World”应用程序中并不需要。全局的注解元数据被放在名为**package-info.java**的文件中，它处在一个特定的包目录中。除了列出被注解的类之外，还需要添加包含全局元数据的包到你的配置中。例如，在一个**Hibernate** XML配置文件中，要添加以下代码：

```
<!DOCTYPE hibernate-configuration SYSTEM
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
<session-factory>
    <!-- ... Many property settings ... -->
    <!-- List of annotated classes-->
    <mapping class="hello.Message"/>

    <!-- List of packages with package-info.java -->
```

```

<mapping package="hello"/>

</session-factory>
</hibernate-configuration>

```

或者可以用编程式的配置实现相同的结果：

```

// Load settings from hibernate.properties
AnnotationConfiguration cfg = new AnnotationConfiguration();
// ... set other configuration options programmatically

cfg.addClass(hello.Message.class);
cfg.addPackage("hello");
SessionFactory sessionFactory = cfg.buildSessionFactory();

```

把这一步再深入，用使用JPA的代码代替加载和存储消息的原生的Hibernate代码。利用Hibernate Annotations和Hibernate EntityManager，可以创建可移植的和标准兼容的映射和数据访问代码。

## 2.2.2 使用 Hibernate EntityManager

Hibernate EntityManager是围绕提供JPA编程接口的Hibernate Core的一个包装，支持JPA实体实例的生命周期，并允许你用标准的Java Persistence查询语言编写查询。由于JPA功能是Hibernate原生能力的一个子集，你可能想知道为什么应当在Hibernate的顶层使用EntityManager包。本节稍后将介绍各种优势，但是一旦你给Hibernate EntityManager配置了项目，马上就会看到一种特殊的简化：你不再非得在配置文件中列出所有被注解的类（或者XML映射文件）了。

我们来修改“Hello World”项目并给它准备好全面的JPA兼容性。

### 1. 基本的JPA配置

SessionFactory在Hibernate应用程序中表示一个特定的逻辑数据存储配置。EntityManager Factory在JPA应用程序中扮演着同样的角色，你用配置文件或者在应用程序代码中配置EntityManager Factory (EMF)，就像配置SessionFactory一样。EMF的配置，与一组映射元数据（通常是被注解的类）一起，被称作持久化单元（persistence unit）。

持久化单元的概念也包括应用程序的包，但是我们想让“Hello World”尽可能地保持简单；假设你要从一个标准化的JPA配置开始，没有特殊的包。不仅内容，就连持久化单元的JPA配置文件的名称和位置也是标准的。

创建名为WORKDIR/etc/META-INF的目录，并把名为persistence.xml的基础配置文件放在那个目录下，如代码清单2-11所示：

#### 代码清单2-11 持久化单元配置文件

```

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

  <persistence-unit name="helloworld">
    <properties>

```

```

<property name="hibernate.ejb.cfgfile"
    value="/hibernate.cfg.xml"/>
</properties>
</persistence-unit>
</persistence>
```

每一个持久化单元都需要一个名称，在这个例子中为helloworld。

**说明** 前面的持久化单元配置文件中的XML首部，声明了应该使用什么模式，并且它始终相同。以后的例子中我们将把它省略，并假设你会添加它。

使用任意数量的属性对持久化单元进行进一步配置，这些属性全是特定于供应商的。前一个例子中的属性hibernate.ejb.cfgfile，就像是一个杂物箱。它引用包含着这个持久化单元所有设置的hibernate.cfg.xml文件（在类路径的根目录中）——你正在重用现有的Hibernate配置。随后，将把所有的配置细节移进persistence.xml文件，但是现在你更关注使用JPA运行“Hello World”。

JPA标准指出，persistence.xml文件要出现在被部署的持久化单元的META-INF目录中。因为你并没有真的打包和部署持久化单元，这意味着必须把persistence.xml复制到构建输出目录的META-INF目录下。修改build.xml，并把下列代码添加到copymetafiles目标：

```

<property name="src.etc.dir" value="etc"/>

<target name="copymetafiles">
    <!-- Copy metadata to build -->
    <copy todir="${build.dir}">
        <fileset dir="${src.java.dir}">
            <patternset refid="meta.files"/>
        </fileset>
    </copy>
    <!-- Copy configuration files from etc/ -->
    <copy todir="${build.dir}">
        <fileset dir="${src.etc.dir}">
            <patternset refid="meta.files"/>
        </fileset>
    </copy>
</target>
```

WORKDIR/etc目录下与meta.files模式相匹配的所有内容都被复制到构建输出目录，这是运行时classpath的一部分。

让我们用JPA重写主应用程序代码。

## 2. 使用JPA的“Hello World”

这些是Java Persistence中主要的编程接口：

- javax.persistence.Persistence——给EntityManagerFactory的创建提供一种静态方法的一个启动类。

- ❑ javax.persistence.EntityManagerFactory——等同于**Hibernate SessionFactory**。这个运行时对象表示一个特定的持久化单元。它是线程安全的，通常被当作一个单例(singleton)处理，并给EntityManager实例的创建提供方法。
  - ❑ javax.persistence.EntityManager——等同于**Hibernate Session**。这个单线程、非共享的对象表示数据访问的一个特定工作单元。它提供方法去管理实体实例的生命周期并创建Query实例。
  - ❑ javax.persistence.Query——等同于**Hibernate Query**。一个对象是一种特定的JPA查询语言或者原生的SQL查询表示法，它允许参数的安全绑定，并给查询的执行提供各种各样的方法。
  - ❑ javax.persistence.EntityTransaction——等同于**Hibernate Transaction**，在Java SE环境中用于RESOURCE\_LOCAL事务的划分。在Java EE中，依赖JTA标准的javax.transaction.UserTransaction接口进行编程式的事务划分。

要使用JPA接口，需要复制必要的库到WORKDIR/lib目录下；在Hibernate EntityManager绑定的文件中查看最新的列表。然后可以在WORKDIR/src/hello/HelloWorld.java中重写代码，并从Hibernate接口切换到JPA接口（请见代码清单2-12）。

**代码清单2-12 使用JPA的“Hello World”主应用程序代码**

```
package hello;

import java.util.*;
import javax.persistence.*;

public class HelloWorld {

    public static void main(String[] args) {
        // Start EntityManagerFactory
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("helloworld");

        // First unit of work
        EntityManager em = emf.createEntityManager();
        EntityTransaction tx = em.getTransaction();
        tx.begin();

        Message message = new Message("Hello World");
        em.persist(message);

        tx.commit();
        em.close();

        // Second unit of work
        EntityManager newEm = emf.createEntityManager();
        EntityTransaction newTx = newEm.getTransaction();
        newTx.begin();

        List messages = newEm
            .createQuery("select m from Message m
              order by m.text asc")
            .getResultList();
    }
}
```

```

        .getResultSet();

        System.out.println( messages.size() + " message(s) found" );

        for (Object m : messages) {
            Message loadedMsg = (Message) m;
            System.out.println(loadedMsg.getText());
        }

        newTx.commit();
        newEm.close();

        // Shutting down the application
        emf.close();
    }
}

```

在这段代码中你可能注意到的第一点是不再有Hibernate导入，只有javax.persistence.\*。用一个对Persistence的静态调用和持久化单元的名称来创建EntityManagerFactory。其余代码应该是不言自明的——就像使用Hibernate一样使用JPA，虽然在API中有些小差别，并且方法的名称也略有不同。此外，没有对基础结构的静态初始化使用HibernateUtil类；你可以编写一个JPUtil类，并且如果愿意，可以把EntityManagerFactory的创建移到那去，或者可以移除目前没用的WORKDIR/src/persistence包。

JPA也支持编程式的配置，通过选项的映射（map）：

```

Map myProperties = new HashMap();
myProperties.put("hibernate.hbm2ddl.auto", "create-drop");
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("helloworld", myProperties);

```

定制的程序属性覆盖了已经在persistence.xml配置文件中设置的任何属性。

试着用一个全新的数据库运行这个移植的HelloWorld代码。你在屏幕上应该看到与使用原生的Hibernate完全相同的日子输出——JPA持久化提供程序引擎是Hibernate。

### 3. 元数据的自动侦测

之前我们承诺过你将不一定要在配置中列出所有被注解的类或者XML映射文件，但是它们仍然在hibernate.cfg.xml中。我们来启用JPA的自动侦测特性。

给org.hibernate包切换到DEBUG日志后再次运行“Hello World”应用程序。日志中应该出现一些额外的行：

```

...
Ejb3Configuration:141
    - Trying to find persistence unit: helloworld
Ejb3Configuration:150
    - Analyse of persistence.xml:
        file:/helloworld/build/META-INF/persistence.xml
PersistenceXmlLoader:115
    - Persistent Unit name from persistence.xml: helloworld
Ejb3Configuration:359
    - Detect class: true; detect hbm: true
JarVisitor:178

```

```

- Searching mapped entities in jar/par: file:/helloworld/build
JarVisitor:217
- Filtering: hello.HelloWorld
JarVisitor:217
- Filtering: hello.Message
JarVisitor:255
- Java element filter matched for hello.Message
Ejb3Configuration:101
- Creating Factory: helloworld
...

```

启动时，`Persistence.createEntityManagerFactory()`方法试图查找具名的持久化单元的位置。它在classpath中搜索所有META-INF/persistence.xml文件，然后如果发现匹配就配置EMF。日志的第二部分显示了一些你可能意想不到的东西。JPA持久化提供程序试图在构建输出目录下找到所有被注解的类和所有Hibernate XML映射文件。不需要hibernate.cfg.xml中被注解类的列表（或者XML映射文件的列表），因为hello.Message这个被注解的实体类已经找到了。

不是只从hibernate.cfg.xml中移除这个单独的不必要选项，而是要移除整个文件，并把所有的配置细节移到persistence.xml里面（请见代码清单2-13）。

### 代码清单2-13 完整的持久化单元配置文件

```

<persistence-unit name="helloworld">

    <provider>org.hibernate.ejb.HibernatePersistence</provider>

    <!-- Not needed, Hibernate supports auto-detection in JSE
        <class>hello.Message</class>
    -->

    <properties>
        <property name="hibernate.archive.autodetection"
            value="class, hbm"/>

        <property name="hibernate.show_sql" value="true"/>
        <property name="hibernate.format_sql" value="true"/>

        <property name="hibernate.connection.driver_class"
            value="org.hsqldb.jdbcDriver"/>
        <property name="hibernate.connection.url"
            value="jdbc:hsqldb:hsq://localhost"/>
        <property name="hibernate.connection.username"
            value="sa"/>

        <property name="hibernate.c3p0.min_size"
            value="5"/>
        <property name="hibernate.c3p0.max_size"
            value="20"/>
        <property name="hibernate.c3p0.timeout"
            value="300"/>
        <property name="hibernate.c3p0.max_statements"
            value="50"/>
    </properties>
</persistence-unit>

```

```

<property name="hibernate.c3p0.idle_test_period"
          value="3000"/>

<property name="hibernate.dialect"
          value="org.hibernate.dialect.HSQLDialect"/>

<property name="hibernate.hbm2ddl.auto" value="create"/>

</properties>
</persistence-unit>

```

在这个配置文件中有3个值得关注的新元素。首先，你设置了应该为这个持久化单元所用的一个显式的<provider>。这通常只在你同时使用几个JPA实现时才需要，但是，我们当然希望Hibernate是唯一的实现。接下来，如果你在非Java EE的环境中部署，规范要求列出包含<class>元素的所有被注解的类——Hibernate处处支持映射元数据的自动侦测，使它变成可选。最后，Hibernate配置设置archive.autodetection告知Hibernate要自动扫描哪些元数据：被注解的类(class)和(或)Hibernate XML映射文件(hbm)。默认情况下，Hibernate EntityManager对两者都进行扫描。配置文件的其他内容包含本章稍早我们在一般的hibernate.cfg.xml文件中解释过和用过的所有选项。

被注解的类和XML映射文件的自动侦测是JPA一个很棒的特性。通常只有在Java EE应用程序服务器中才可用；至少，这是EJB 3.0规范保证的东西。但是，Hibernate作为一个JPA提供程序，也在简单的Java SE中实现它，虽然你也许不能够使用与任何其他JPA提供程序完全相同的配置。

现在，你已经构建了一个完全遵循JPA规范的应用程序。项目目录看起来应该像这样（注意我们也把log4j.properties移到了etc/目录下）：

```

WORKDIR
+etc
  log4j.properties
  +META-INF
    persistence.xml
+lib
  <all required libraries>
+src
  +hello
    HelloWorld.java
    Message.java

```

所有的JPA配置设置都捆绑在persistence.xml中，所有的映射元数据都包括在Message类的Java源代码中，Hibernate在启动时自动扫描并找到元数据。相较于纯粹的Hibernate，现在你拥有这些好处：

- 被部署元数据的自动扫描，这在大项目中是个重要的特性。如果几百个实体由一个大团队开发时，维护一系列被注解的类或者映射文件就变得很困难。
- 标准的和简化的配置，配置文件有标准的位置，以及一个部署概念——持久化单元——在一个应用程序文档(EAR)中包有几个单元(JAR)的更大项目中，它有着更多的优势。
- 标准的数据访问代码、实体实例的生命周期和完全可移植的查询。应用程序中没有私有的导入。

这些仅仅是JPA的其中一些优势。如果你把它与完全的EJB 3.0编程模型和其他的托管组件结合起来，就会看到它真正的威力。

### 2.2.3 引入 EJB 组件

当你还使用EJB 3.0会话bean和消息驱动的bean（和其他Java EE 5.0标准）时，Java Persistence就开始初露锋芒。EJB 3.0规范已经被设计为允许持久化的整合，因此你可以在Bean方法范围获得自动的事务划分，或者跨越状态会话EJB生命周期的一个持久化上下文（如Session）。

本节将从一个托管的Java EE环境中的EJB 3.0和JPA开始，你将再次通过修改“Hello World”应用程序学习基础的知识。首先需要Java EE环境——提供Java EE服务的运行时容器。有两种方法可以获得它：

- 可以安装一个支持EJB 3.0和JPA的完全Java EE 5.0应用程序服务器。几个开源（Sun GlassFish、JBoss AS、Object Web EasyBeans）和其他私有许可的选择在编写本书时市面上已有出售，当你读到本书时可能有更多的选择了。
- 可以安装一个模块化的服务器，从完全Java EE 5.0包中选择，仅仅提供你需要的服务。至少，你可能想要一个EJB 3.0容器、JTA事务服务和一个JNDI注册。编写本书之时，只有JBoss AS在一个易于定制的包中提供了模块化的Java EE 5.0服务。

为了使事情保持简单，并展示从EJB 3.0开始有多么容易，你将安装和配置模块化的JBoss Application Server，并只启用所需要的Java EE 5.0服务。

#### 1. 安装EJB容器

到<http://jboss.com/products/ejb3>下载这个模块化的可嵌入服务器，并解压下载到的文档。把服务器带来的所有库复制到项目的WORKDIR/lib目录下，并将所有被包括的配置文件复制到WORKDIR/src目录。现在你应该具备下列目录布局：

```

WORKDIR
+etc
  default.persistence.properties
  ejb3-interceptors-aop.xml
  embedded-jboss-beans.xml
  jndi.properties
  log4j.properties
+META-INF
  helloworld-beans.xml
  persistence.xml
+lib
  <all required libraries>
+src
  +hello
    HelloWorld.java
    Message.java

```

JBoss可嵌入的服务器依赖于Hibernate提供Java Persistence，因此default.persistence.properties文件包含全部部署所需的Hibernate默认设置（例如JTA整合设置）。ejb3-interceptors-aop.xml和embedded-jboss-beans.xml配置文件包含服务器的服务配置——你可以看一下这些文件，但是现在

不需要修改它们。在编写本书之时，默认启用的服务是JNDI、JCA、JTA和EJB 3.0容器——正是你所需要的。

为了迁移“Hello World”应用程序，需要一个托管的数据源，这是一个由可嵌入的服务器处理的数据库连接。配置一个托管数据源最容易的方法是，添加一个把数据源当作托管服务部署的配置文件。创建代码清单2-14中的文件为WORKDIR/ect/META-INF/helloworld-beans.xml。

#### 代码清单2-14 JBoss服务器的数据源配置文件

```

<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:jboss:bean-deployer bean-deployer_1_0.xsd"
  xmlns="urn:jboss:bean-deployer:2.0">

  <!-- Enable a JCA datasource available through JNDI -->
  <bean name="helloWorldDatasourceFactory"
    class="org.jboss.resource.adapter.jdbc.local.LocalTxDataSource">

    <property name="jndiName">java:/HelloWorldDS</property>

    <!-- HSQLDB -->
    <property name="driverClass">
      org.hsqldb.jdbcDriver
    </property>
    <property name="connectionURL">
      jdbc:hsqldb:hsq://localhost
    </property>
    <property name="userName">sa</property>

    <property name="minSize">0</property>
    <property name="maxSize">10</property>
    <property name="blockingTimeout">1000</property>
    <property name="idleTimeout">100000</property>

    <property name="transactionManager">
      <inject bean="TransactionManager"/>
    </property>
    <property name="cachedConnectionManager">
      <inject bean="CachedConnectionManager"/>
    </property>
    <property name="initialContextProperties">
      <inject bean="InitialContextProperties"/>
    </property>
  </bean>

  <bean name="HelloWorldDS" class="java.lang.Object">
    <constructor factoryMethod="getDatasource">
      <factory bean="helloWorldDatasourceFactory"/>
    </constructor>
  </bean>
</deployment>
```

同样，XML首部和Schema声明对于这个例子也不重要。你创建了两个bean：第一个是可以产生第二个bean类型的工厂。LocalTxDataSource现在实际上是数据库连接池，所有的连接池设

置都可在这个工厂使用。该工厂把一个托管数据源绑定在JNDI名称java:/HelloWorldDS下面。

第二个bean配置声明，如果另一个服务在JNDI注册中查找名为HelloWorldDS的注册对象，它应该如何被实例化。你的“Hello World”应用程序需要这个名称下的数据源，服务器在LocalTxDataSource工厂中调用getDatasource()来获得它。

还要注意我们在属性值中添加了一些换行，使它更易于阅读——在实际配置文件中你不应该这么做（除非你的数据库用户名包含换行）。

## 2. 配置持久化单元

接下来，需要改变“Hello World”应用程序的持久化单元配置，来访问一个托管的JTA数据源，而不是本地资源的连接池。把WORKDIR/etc/META-INF/persistence.xml文件变成如下：

```
<persistence ...>
    <persistence-unit name="helloworld">
        <jta-data-source>java:/HelloWorldDS</jta-data-source>
        <properties>
            <property name="hibernate.show_sql" value="true"/>
            <property name="hibernate.format_sql" value="true"/>
            <property name="hibernate.dialect"
                value="org.hibernate.dialect.HSQLDialect"/>
            <property name="hibernate.hbm2ddl.auto" value="create"/>
        </properties>
    </persistence-unit>
</persistence>
```

你去除了许多不再有关系的Hibernate配置选项，例如连接池和数据库连接设置。反之，你用JNDI中绑定的数据源名称设置了一个<jta-data-source>属性。别忘了你仍然需要配置正确的SQL方言，以及没有出现在default.persistence.properties中的任何其他的Hibernate选项。

现在，环境的安装和配置完成了，（我们很快就要介绍jndi.properties文件的用途）并且能够用EJB重写应用程序代码。

## 3. 编写EJB

设计和创建含有多个托管组件的应用程序有许多种方法。这个“Hello World”应用程序太简单，不足以介绍详细的例子，因此我们只介绍最基础的EJB类型：一个无状态会话bean(stateless session bean)。（你已经见过实体类——被注解的简单Java类，能够有持久化实例。注意术语实体bean仅仅指旧的EJB 2.1实体bean；EJB 3.0和Java Persistence给简单的实体类标准化了一个轻量级的编程模型。）

每个EJB会话bean都需要一个业务接口(business interface)。这不是一个需要实现预设方法或者扩展现有方法的特殊接口，它是简单的Java。在WORKDIR/src/hello包中创建下列接口：

```
package hello;

public interface MessageHandler {
    public void saveMessages();
    public void showMessages();
}
```

MessageHandler能够保存和显示消息；它很容易理解。实际的EJB实现这个业务接口，这个接口被默认当作一个本地接口（也就是说，远程的EJB客户端程序无法调用它），请见代码清单2-15。

#### 代码清单2-15 “Hello World” EJB会话bean应用程序代码

```
package hello;

import javax.ejb.Stateless;
import javax.persistence.*;
import java.util.List;

@Stateless
public class MessageHandlerBean implements MessageHandler {

    @PersistenceContext
    EntityManager em;

    public void saveMessages() {
        Message message = new Message("Hello World");
        em.persist(message);
    }

    public void showMessages() {
        List messages =
            em.createQuery("select m from Message m
                           order by m.text asc")
                .getResultList();

        System.out.println(messages.size() + " message(s) found:");

        for (Object m : messages) {
            Message loadedMsg = (Message) m;
            System.out.println(loadedMsg.getText());
        }
    }
}
```

在这个实现中要观察几件值得关注的事。首先，这是个简单的Java类，没有强烈地依赖于任何其他的包。它变成了一个只有单个元数据注解@Stateless的EJB。EJB支持容器托管的服务，因此可以应用@PersistenceContext注解，并且每当调用这个无状态bean中的方法时，服务器就注入一个全新的EntityManager实例。每一种方法也都通过容器自动分配一个事务。事务在调用方法时启动，并在方法返回时提交。（当方法内部抛出异常时它则回滚。）

你现在可以修改HelloWorld主类并把所有存储和显示消息的工作委托给MessageHandler。

#### 4. 运行应用程序

“Hello World”应用程序的主类在JNDI注册中找到MessageHandler的无状态会话bean之后调用它。显然，托管环境和整个应用程序服务器（包括JNDI注册）都必须先被启动。你在HelloWorld.java的main()方法中完成所有这些工作（请见代码清单2-16）。

#### 代码清单2-16 “Hello World” 主应用程序代码调用EJB

```
package hello;
```

```

import org.jboss.ejb3.embedded.EJB3StandaloneBootstrap;
import javax.naming.InitialContext;

public class HelloWorld {

    public static void main(String[] args) throws Exception {
        // Boot the JBoss Microcontainer with EJB3 settings, automatically
        // loads ejb3-interceptors-aop.xml and embedded-jboss-beans.xml
        EJB3StandaloneBootstrap.boot(null);

        // Deploy custom stateless beans (datasource, mostly)
        EJB3StandaloneBootstrap
            .deployXmlResource("META-INF/helloworld-beans.xml");

        // Deploy all EJBs found on classpath (slow, scans all)
        // EJB3StandaloneBootstrap.scanClasspath();

        // Deploy all EJBs found on classpath (fast, scans build directory)
        // This is a relative location, matching the substring end of one
        // of java.class.path locations. Print out the value of
        // System.getProperty("java.class.path") to see all paths.
        EJB3StandaloneBootstrap.scanClasspath("helloworld-ejb3/bin");

        // Create InitialContext from jndi.properties
        InitialContext initialContext = new InitialContext();

        // Look up the stateless MessageHandler EJB
        MessageHandler msgHandler = (MessageHandler) initialContext
            .lookup("MessageHandlerBean/local");

        // Call the stateless EJB
        msgHandler.saveMessages();
        msgHandler.showMessages();

        // Shut down EJB container
        EJB3StandaloneBootstrap.shutdown();
    }
}

```

main()中的第一个命令启动了服务器的内核，并部署了在服务配置文件中找到的基础服务。接下来，你稍早在helloworld-beans.xml中创建的数据源工厂配置得到了部署，且数据源被容器绑定到JNDI。从那时起，容器就准备好部署EJB了。部署所有EJB最容易的（但经常不是最快的）方法是让容器在整个classpath中搜索任何具有EJB注解的类。要学习许多其他可用的部署选项，请查阅下载文件中捆绑的JBoss AS文档。

要查找EJB，需要一个InitialContext，这是JNDI注册的入口点。如果实例化InitialContext，Java就会自动在classpath中查找文件jndi.properties。要在WORKDIR/etc中，用与JBoss服务器的JNDI注册配置相匹配的设置创建这个文件：

```

java.naming.factory.initial
  => org.jnp.interfaces.LocalOnlyContextFactory
java.naming.factory.url.pkgs org.jboss.naming:org.jnp.interfaces

```

你不需要准确了解这个配置的含义，但是它主要把InitialContext指向在本地虚拟机上运行的一个JNDI注册（远程的EJB客户端程序调用需要一个支持远程通信的JNDI服务）。

默认情况下，按一个实现类的名称查找MessageHandler bean，本地的接口带有/local后缀。EJB如何命名，它们如何被绑定到JNDI，以及你如何不同地查找它们，如何定制它们，这些都是JBoss服务器的默认值。

最后，调用MessageHandler EJB，并让它在两个单元内自动完成所有工作——每个方法调用将产生各自的事务。

这样，就用托管的EJB组件和集成的JPA完成了我们的第一个例子。你可能已经知道自动的事务划分和EntityManager注入如何改善代码的可读性。接下来，将介绍有状态会话bean如何用事务的语义帮助你实现用户和应用程序之间的复杂会话。此外，EJB组件不包含任何不必要的胶合代码（glue code）或者基础结构方法，并且它们在任何EJB 3.0容器中都是完全可重用、可移植和可执行的。

---

**说明** 持久化单元的包——我们之前没有过多谈论持久化单元的包——你不需要为了任何部署而打包“Hello World”例子。然而，如果要使用特性例如一个完全应用程序服务器上的热重新部署（hot redeployment），需要正确地打包应用。这包括JAR、WAR、EJB-JAR和EAR的一般组合。部署和打包经常也是特定于供应商的，因此应该参考你应用程序服务器的文档了解更多信息。JPA持久化单元的作用范围可以到JAR、WAR和EJB-JAR，这意味着这些文档中的一个或者几个包含了所有被注解的类，以及带有这个特定单元的所有设置的一个META-INF/persistence.xml配置文件。可以把一个或者几个JAR、WAR和EJB-JAR包在单个企业应用文档EAR中。你的应用程序服务器应该正确地发现所有持久化单元，并自动地创建必要的工厂。用@PersistenceContext注解中的一个单元名称属性，指示容器从一个特定的单元注入EntityManager。

---

应用程序的完全可移植性通常不是使用JPA或者EJB 3.0的主要原因。毕竟，你决定了要用Hibernate作为JPA持久化的提供程序。来看看你可以如何退回并不时使用Hibernate原生的特性。

#### 2.2.4 切换到 Hibernate 接口

决定用Hibernate作为JPA持久化的提供程序有几个原因：首先，Hibernate是一个很好的JPA实现，它提供许多不影响代码的选项。例如，可以在JPA配置中启用Hibernate的二级数据高速缓存，并透明地提升应用的性能和可量测性，而不需要动任何代码。

其次，必要时可以使用原生的Hibernate映射或者API。3.3节讨论映射的混合（特别是注解），但是这里我们要说明，需要的时候，你如何能够可以在JPA应用中使用Hibernate API。显然，把Hibernate API导入到代码中使得把代码移植到一个不同的JPA提供程序变得更困难了。因此，正确地隔离这部分代码变得尤其重要，或者至少用文档说明你为什么和何时使用过一个原生的Hibernate特性。

你可以从它们对应的JPA接口退回到Hibernate API，获得Configuration、SessionFactory，必要时甚至是Session。

例如，不必使用Persistence静态的类创建EntityManagerFactory，相反使用Hibernate的Ejb3Configuration：

```
Ejb3Configuration cfg = new Ejb3Configuration();
EntityManagerFactory emf =
    cfg.configure("/custom/hibernate.cfg.xml")
        .setProperty("hibernate.show_sql", "false")
        .setInterceptor( new MyInterceptor() )
        .addAnnotatedClass( hello.Message.class )
        .addResource( "/Foo.hbm.xml" )
        .buildEntityManagerFactory();

AnnotationConfiguration
    hibCfg = cfg.getHibernateConfiguration();
```

Ejb3Configuration是一个新的接口，它与常规的Hibernate Configuration一样，并没有进行扩展（这是一个实现细节）。这意味着，例如，可以从Ejb3Configuration中获得简单的Annotation Configuration对象，并通过编程把它传递到SchemaExport实例。

如果需要编程式地控制二级高速缓存区，SessionFactory接口就很有用。可以先通过转换EntityManagerFactory获得SessionFactory：

```
HibernateEntityManagerFactory hibEMF =
    (HibernateEntityManagerFactory) emf;
SessionFactory sf = hibEMF.getSessionFactory();
```

使用同样的方法可以从EntityManager中获得Session：

```
HibernateEntityManager hibEM =
    (HibernateEntityManager) em;
Session session = hibEM.getSession();
```

这不是从标准的EntityManager中获得原生API的唯一方法。JPA规范支持返回底层实现的getDelegate()方法：

```
Session session = (Session) entityManager.getDelegate();
```

或者可以获得一个注入到EJB组件的Session（虽然这仅适用于JBoss应用程序服务器）：

```
@Stateless
public class MessageHandlerBean implements MessageHandler {
    @PersistenceContext
    Session session;

    ...
}
```

极少数情况下，能够从Hibernate Session退回到简单的JDBC接口：

```
Connection jdbcConnection = session.connection();
```

这最后一个选项有一些警告：不允许你关闭从Hibernate中获得的JDBC Connection——它会自动进行。这个规则的例外情况是，在一个依赖于积极的连接释放的环境中，这意味着在JTA或者CMT环境中，你必须关闭在应用程序代码中返回的连接。

直接访问JDBC连接的一种更好更安全的方法是通过Java EE 5.0中的资源注入。在一个EJB、EJB监听器、servlet、servlet过滤器甚至JavaServer Faces backing bean中，注解一个字段或者设置方法，像下面这样：

```
@Resource(mappedName="java:/HelloWorldDS") DataSource ds;
```

目前为止，假设你是在一个没有涉及遗留应用程序代码（或者现有的数据库Schema）的新Hibernate或者新JPA项目上进行。现在转换一下视角，考虑一个自底而上的开发过程。在这样一个场景中，你或许想要自动地从现有的数据库Schema中反向工程所需要创建的东西。

## 2.3 反向工程遗留数据库

当映射遗留数据库时，第一步可能涉及一个自动的反向工程过程。毕竟，实体Schema已经存在于你的数据库系统中。为了使这项工作更容易些，Hibernate配有一组工具，可以从这个元数据（包括XML映射文件和Java源代码）中读取Schema，并生成各种需要创建的东西。所有这些都是基于模板的，因此许多定制工作成了可能。

可以用工具和Ant构建中的任务控制反向工程的进行。你之前用来从Hibernate映射元数据中导出SQL DDL的HibernateToolTask，具有更多的选项，大部分选项都与反向工程有关，关于XML映射文件、Java代码，甚至整个应用程序骨架如何可以从现有的数据库Schema中自动生成。

我们将首先介绍如何编写一个可以把现有的数据库加载到Hibernate元数据模型中的Ant目标。接下来，应用各种导出组件，并从数据库表和列中生成XML文件、Java代码和其他有用的构件。

### 2.3.1 创建数据库配置

假设你有一个新的WORKDIR目录，里面除了lib目录（以及它的一般内容）和空的src目录之外什么也没有。要从现有的数据库中生成映射和代码，首先需要创建一个包含数据库连接设置的配置文件：

```
hibernate.dialect = org.hibernate.dialect.HSQLDialect
hibernate.connection.driver_class = org.hsqldb.jdbcDriver
hibernate.connection.url = jdbc:hsqldb:hsql://localhost
hibernate.connection.username = sa
```

直接把这个文件保存在WORKDIR中，并命名为helloworld.db.properties。这里显示的四行是连接到数据库并读取所有表和列所需的最少代码。你可能已经创建了一个Hibernate XML配置文件，而不是helloworld.db.properties，但是没有必要把这弄得比需要的更复杂。

接下来编写Ant目标。在项目的一个build.xml文件中，添加下列代码：

```
<taskdef name="hibernatetool"
        classname="org.hibernate.tool.ant.HibernateToolTask"
        classpathref="project.classpath"/>

<target name="reveng.hbmxml"
        description="Produces XML mapping files in src directory">

    <hibernatetool destdir="${basedir}/src">

        <jdbcconfiguration
            propertyfile="${basedir}/helloworld.db.properties"
            revengfile="${basedir}/helloworld.reveng.xml"/>
    
```

```

<hbm2hbmxml/> <!-- Export Hibernate XML files -->
<hbm2cfgxml/> <!-- Export a hibernate.cfg.xml file -->
</hibernatetool>
</target>

```

Ant的这个HibernateToolTask定义和以前的一样。假设你将重用前面章节中介绍过的大部分构建文件，并且如project.classpath这样的引用也是一样的。<hibernatetool>任务用WORKDIR/src设置为所有生成工件的默认目标目录。

<jdbcconfiguration>是一个Hibernate工具配置，可以通过JDBC连接到数据库，以及从数据库目录中读取JDBC元数据。通常用两个选项对它进行配置：数据库连接设置（属性文件）和一个可选的反向工程定制文件。

然后，由工具配置生成的元数据被导入到导出器。这个范例的Ant目标命名了这样两个导出器：hbm2hbmxml导出器，就像你能从它的命名中猜到的那样，它从配置中取出Hibernate元数据(hbm)，并生成Hibernate XML映射文件；第二个导出器可以准备一个hibernate.cfg.xml文件，该文件列出所有生成的XML映射文件。

在谈论这些以及各种其他的导出器之前，要花点时间讨论反向工程定制文件以及如何使用它。

### 2.3.2 定制反向工程

JDBC元数据——也就是说，你可以通过JDBC从一个数据库中读取有关它自身的信息——通常不足以创建一个完美的XML映射文件，至于Java应用代码更不必说了。反过来也可能是对的：你的数据库可能包含你想要忽略（如特定的表或者列）的信息，或者你希望用非默认的策略转换的信息。可以用一个使用XML语法的反向工程配置文件定制反向工程的过程。

假设你正在使用单独的MESSAGE表和仅有的几个列，对本章前面创建的“Hello World”数据库反向工程。使用helloworld.reveng.xml文件，如代码清单2-17所示，可以定制这个反向工程。

#### 代码清单2-17 被定制反向工程的配置

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-reverse-engineering SYSTEM
  "http://.hibernate.sourceforge.net/
   hibernate-reverse-engineering-3.0.dtd">

<hibernate-reverse-engineering>
  <table-filter match-name=".*" package="hello"/> ①
  <table name="MESSAGES" schema="PUBLIC" class="Message"> ②
    <primary-key> ③
      <generator class="increment"/>
      <key-column name="MESSAGE_ID" property="id" type="long"/>
    </primary-key> ④
    <column name="MESSAGE_TEXT" property="text"/> ⑤
  </table>
</hibernate-reverse-engineering>

```

```

<foreign-key constraint-name="FK_NEXT_MESSAGE"> ◀
    <many-to-one property="nextMessage" />
    <set exclude="true" />
</foreign-key>

</table>

</hibernate-reverse-engineering>

```

- ① 这个XML文件有它自己的DTD用于验证和自动完成。
- ② 表过滤器可以用一个正则表达式按名称把表排除。然而，在这个例子中，给所有匹配这个正则表达式的表生成的类定义了一个默认的包。
- ③ 可以按名称定制单独的表。模式名称通常是可选的，但是默认时HSQLDB把PUBLIC模式分配给所有表，因此获取JDBC元数据时，识别表就需要这项设置。这里也可以给生成的实体设置一个定制的类名。
- ④ 主键列生成一个具名id的属性，默认时是messageId。也显式地声明应该使用哪个Hibernate标识符生成器。
- ⑤ 可以排除单独的列，或者，在这个例子中，可以指定生成属性的名称——默认时是messageText。
- ⑥ 如果外键约束FK\_NEXT\_MESSAGE从JDBC元数据中获取，默认时就把一个多对一的关联创建到该类的目标实体。通过按名称匹配外键约束，可以指定是否应该生成一个反向集合（一对多）（本例排除了这一点），以及这个多对一的属性应该如何命名。

如果现在使用这个定制运行Ant目标，它就会在源目录的hello包中生成Message.hbm.xml文件。（首先要把Freemarker和jTidy JAR文件复制到你的库目录里面。）你所做的定制造成了Hibernate映射文件与你稍早手写的相同，如代码清单2-2所示。

除了XML映射文件之外，Ant目标还在源目录中生成一个Hibernate XML配置文件：

```

<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">
            org.hsqldb.jdbcDriver
        </property>
        <property name="hibernate.connection.url">
            jdbc:hsqldb:hsq://localhost
        </property>
        <property name="hibernate.connection.username">
            sa
        </property>
        <property name="hibernate.dialect">
            org.hibernate.dialect.HSQLDialect
        </property>
        <mapping resource="hello/Message.hbm.xml" />
    </session-factory>
</hibernate-configuration>

```

这个导出器把你用于反向工程的所有数据库连接设置都写进了这个文件，假设这就是当你运

行应用程序时要连接到的数据库。它也把所有生成的XML映射文件添加到了配置中。

下一步做什么呢？可以开始给Message的Java类编写源代码。或者让Hibernate Tools为你生成领域模型的类。

### 2.3.3 生成 Java 源代码

假设Message类有一个现成的Hibernate XML映射文件，并且想要给这个类生成源代码。如第3章所述，一个简单的Java实体类实现Serializable是很理想的，它有一个无参的构造函数（constructor），带有所有属性的获取方法和设置方法，并且有一个封装的实现。

可以使用在Ant构建中的Hibernate Tools和hbm2java 导出器生成用于实体类的源代码。这个源工件可以是任何能够被读进Hibernate元数据模型的东西——如果你想要定制Java代码生成，Hibernate XML映射文件是最好的。

把下列目标添加到Ant构建：

```
<target name="reveng.pojos"
    description="Produces Java classes from XML mappings">

    <hibernatetool destdir="${basedir}/src">

        <configuration>
            <fileset dir="${basedir}/src">
                <include name="**/*.hbm.xml"/>
            </fileset>
        </configuration>

        <hbm2java/> <!-- Generate entity class source -->
    </hibernatetool>
</target>
```

<configuration>读取所有的Hibernate XML映射文件，<hbm2java> 导出器通过默认的策略生成Java源代码。

#### 1. 定制实体类生成

默认情况下，hbm2java给每个被映射的实体生成一个简单的实体类。这个类实现Serializable标识接口（marker interface），并具备所有属性的访问方法和必要的构造函数。类的所有属性对字段都有私有（private）可见性，尽管你可以在XML映射文件中用<meta>元素和属性改变这个行为。

对默认的反射工程行为所做的第一个改变是限制Message属性的可见性范围。默认时，所有访问方法生成时都包含公有（public）可见性。假设Message对象是不可变的；你不想在公共接口上公开设置方法，而只公开获取方法。不用<meta>元素增强每个属性的映射，而是在类级中声明一个元属性（meta-attribute），如此来把设置应用给那个类中的所有属性：

```
<class name="Message"
    table="MESSAGES">

    <meta attribute="scope-set">private</meta>
```

```
...
```

其中scope-set属性定义了属性设置方法的可见性。

hbm2java 导出器也在下一个更高的级别接受元属性，即在根<hibernate-mapping>元素中，然后它被应用到所有在XML文件中被映射的类。也可以把细粒度的元属性添加到单个属性、集合或者组件映射。

生成实体类的一个（虽然小）改进，是在生成的toString()方法的输出中包含Message的text。文本是应用程序的日志输出中一个很好的可视控件元素。可以改变Message的映射，来把它包括在生成的代码中：

```
<property name="text" type="string">
    <meta attribute="use-in-tostring">true</meta>
    <column name="MESSAGE_TEXT" />
</property>
```

Message.java中toString()方法的生成代码看起来像下面这样：

```
public String toString() {
    StringBuffer buffer = new StringBuffer();
    buffer.append(getClass().getName())
        .append("@")
        .append( Integer.toHexString(hashCode()) )
        .append(" [");
    append("text").append("='").append(getText()).append("'");
    .append("']");
    return buffer.toString();
}
```

元属性可以被继承；也就是说，如果你在一个<class>元素的级别中声明一个use-in-tostring，这个类的所有属性都被包括在toString()方法中。这种继承机制适用于所有hbm2java的元属性，但是你可以选择把它关闭：

```
<meta attribute="scope-class" inherit="false">public abstract</meta>
```

在scope-class的元属性中设置inherit（继承）为false，仅仅创建了这个<meta>元素的父类为public abstract，但没有创建任何（可能）嵌套的子类。

在编写本书之时，hbm2java 导出器支持17种元属性用于调优代码的生成。大部分与可见性、接口实现、类扩展和预定义的Javadoc注释有关。完整的列表请参阅Hibernate Tools文档。

如果你用的是JDK 5.0，可以通过在<hbm2java>任务中设置jdk5="true"，切换到自动生成的静态导入（static import）和泛型（generic）。或者可以生成包含注解的EJB 3.0实体类。

## 2. 生成Java Persistence实体类

一般地，在实体类源代码中使用Hibernate XML映射文件或者JPA注解来定义映射元数据，因此从XML映射文件中生成包含注解的Java Persistence实体类不太合理。然而，可以直接从JDBC元数据中创建包含注解的实体类源代码，并跳过XML映射步骤。看看下列Ant目标：

```
<target name="reveng.entities"
    description="Produces Java entity classes in src directory">
```

```

<hibernatetool destdir="${basedir}/src">
    <jdbcconfiguration
        propertyfile ="${basedir}/helloworld.db.properties"
        revengfile ="${basedir}/helloworld.reveng.xml"/>
    <hbm2java jdk5="true" ejb3="true"/>
    <hbm2cfgxml ejb3="true"/>
</hibernatetool>
</target>

```

这个目标生成包含映射注解的实体类源代码和列有这些被映射类的文件。可以直接编辑Java源代码来定制映射，如果在中定制太受限制的话。

还要注意所有的导出器都依赖于以FreeMarker模板语言编写的模板。可以用任何喜欢的方式定制模板，甚至可以编写自己的模板。甚至编程定制代码的生成也是可能的。Hibernate Tools参考文档说明了如何使用这些选项。

Hibernate Tools提供的其他导出器和配置如下：

- <annotationconfiguration>取代一般的<configuration>，如果你想要从被注解的Java类中读取映射元数据，而不是从XML映射文件中读取的话。它唯一的实参是包含一系列已注解类的文件的位置和名称。使用这种方法从被注解的类中导出一个数据库Schema。
- <ejb3configuration>相当于<annotationconfiguration>，除了它可以自动在classpath中扫描被注解的Java类之外；它不需要文件。
- <hbm2dao>导出器可以基于数据访问对象（Data Access Object，DAO）模式，给持久层创建额外的Java源代码。在编写本书之时，这个导出器的模板已经很旧了，需要更新。我们希望完成后的新模板将类似于16.2节中所示的DAO代码。
- <hbm2doc>导出器生成给表和Java实体提供文档的HTML文件。
- <hbmtemplate>导出器可以用一组定制的FreeMarker模板参数化，通过这种方法可以生成你想要的任何东西。Hibernate Tools捆绑了利用JBoss Seam框架来生成完整的、可运行的骨架应用程序的模板。

可以利用这些工具的导入和导出功能来实现创造性。例如，可以用<annotationconfiguration>读取被注解的Java类，并用<hbm2hbmxml>将它们导出。这允许你用JDK 5.0和更方便的注解进行开发，而在产品中（在JDK 1.4中）则部署Hibernate XML映射文件。

来用一些更高级的配置选项来结束本章的讨论，并把Hibernate和Java EE服务整合起来。

## 2.4 与 Java EE 服务整合

假设你已经试过了本章前面介绍过的“Hello World”示例，并且熟悉基础的Hibernate配置，以及Hibernate如何与简单的Java应用程序整合。现在，我们要讨论更高级的原生的Hibernate配置选项，以及一般的Hibernate应用程序如何使用Java EE应用程序服务器提供的Java EE服务。

如果你用Hibernate Annotations和Hibernate EntityManager创建过第一个JPA项目，接下来的配置建议跟你就没有太大关系了——如果你使用JPA，就已经深入到Java EE，不需要另外的整合步骤了。因此，如果使用Hibernate EntityManager的话，可以跳过本节的内容。

Java EE应用程序服务器，例如JBoss AS、BEA WebLogic和IBM WebSphere，为Java实现标准的（特定于Java EE）的托管环境。Hibernate能够与之整合的三大最值得关注的Java EE服务是JTA、JNDI和JMX。

JTA允许Hibernate参与托管资源中的事务。Hibernate可以通过JNDI查找托管资源（数据库连接），并且能够把自身当作一项服务绑定到JNDI上。最后，Hibernate可以通过JMX被部署，然后通过JMX容器被当作一项服务来管理，并且使用标准的JMX客户端程序在运行时被监控。

来看看每一项服务，以及如何把Hibernate和它整合起来。

#### 2.4.1 与 JTA 整合

JTA是Java企业应用中事务控制的标准服务接口。它公开了几个接口，例如用于事务划分的UserTransaction API和用于参与列事务生命周期中的TransactionManager API。事务管理器能够协调跨资源的单个事务——想象在单个事务的两个数据库中的两个Hibernate Session中进行。

JTA事务服务由所有Java EE应用程序服务器提供。然而，许多Java EE服务可以被独立使用，并且可以给应用部署JTA提供程序，例如JBoss Transaction或者ObjecWeb JOTM。我们对于配置的这部分不多说，而是关注Hibernate与JTA服务的整合，在完全的应用程序服务器中或者使用独立的JTA提供程序时也一样。

看看图2-6。使用Hibernate的Session接口访问（多个）数据库，Hibernate的责任是与托管环境的Java EE服务整合。

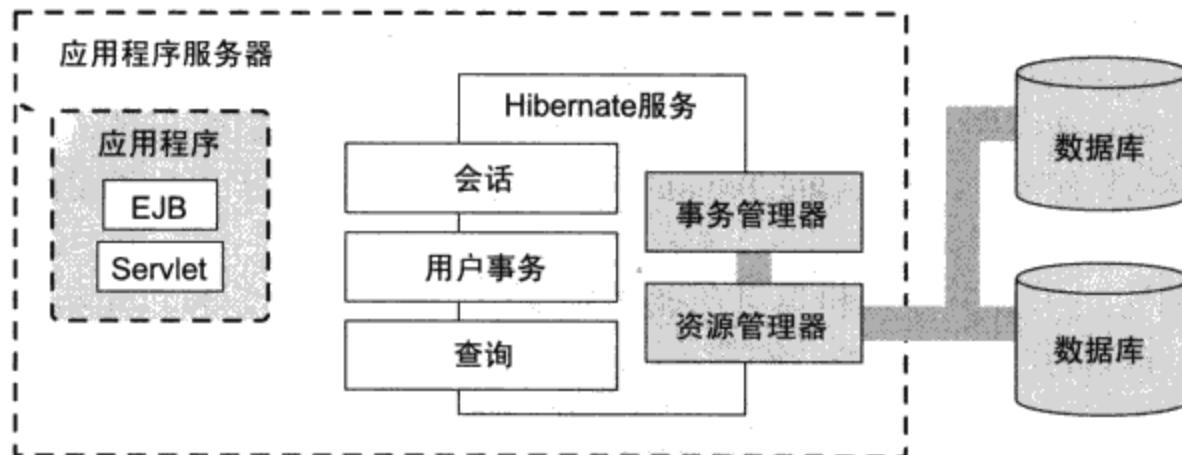


图2-6 包含托管资源的环境中的Hibernate

在这样一个托管环境中，Hibernate不再创建和维护JDBC连接池——Hibernate通过在JNDI注册中查找Datasource对象获得数据库连接。因此，你的Hibernate配置需要一个对JNDI名称的引用，在这个引用中能够获得托管的连接。

```
<hibernate-configuration>
<session-factory>

    <property name="hibernate.connection.datasource">
        java:/MyDatasource
    </property>

    <property name="hibernate.dialect">
        org.hibernate.dialect.HSQLDialect
    </property>
    ...

</session-factory>
</hibernate-configuration>
```

利用这个配置文件，Hibernate 使用名称 `java:/MyDatasource` 在 JNDI 中查找数据库连接。当你配置应用程序服务器和部署应用程序时，或者当你配置独立的 JTA 提供程序时，这就是你应该把托管的数据源绑定到上面的名称。注意 Hibernate 仍然需要一个方言设置来生成正确的 SQL。

---

**说明** **Hibernate 和 Tomcat**——Tomcat 不是一个 Java EE 应用程序服务器；它只是一个 servlet 容器，虽然 servlet 容器具有一些通常只有应用程序服务器才有的特性。这些特性中有一种可以与 Hibernate 共用：Tomcat 连接池。Tomcat 内部使用 DBCP 连接池，但是把它当作一个 JNDI 数据源公开，就像真实的应用程序服务器一样。要配置 Tomcat 数据源，需要根据 Tomcat JNDI/JDBC 文档中的说明编辑 `server.xml`。Hibernate 可以被配置为通过设置 `hibernate.connection.datasource` 来使用这个数据源。记住，Tomcat 不提供事务管理器，因此你仍然有个简单的 JDBC 事务语义，Hibernate 可以用可选的 Transaction API 把它隐藏起来。另一种方法是，可以与你的 Web 应用程序一起部署一个可与 JTA 兼容的、独立的事务管理器，考虑用它获得标准的 UserTransaction API。另一方面，一般的应用程序服务器（尤其当它像 JBoss AS 一样模块化时）可能比在 Tomcat 加 DBCP 加 JTA 更易于配置，并且提供更好的服务。

---

要完全把 JTA 与 Hibernate 整合，需要告诉 Hibernate 更多有关事务管理器的信息。Hibernate 必须钩入事务生命周期，例如管理它的高速缓存。首先，要告诉 Hibernate 你正在使用哪种事务管理器：

```
<hibernate-configuration>
<session-factory>

    <property name="hibernate.connection.datasource">
        java:/MyDatasource
    </property>

    <property name="hibernate.dialect">
        org.hibernate.dialect.HSQLDialect
    </property>

    <property name="hibernate.transaction.manager_lookup_class">
        org.hibernate.transaction.JBossTransactionManagerLookup
    </property>

    <property name="hibernate.transaction.factory_class">
        org.hibernate.transaction.JTATransactionFactory
    </property>
```

```
</property>  
  
...  
  
</session-factory>  
</hibernate-configuration>
```

你要给应用程序服务器选择适当的查找类，就如在前面的代码中所做的那样——Hibernate给最普及的JTA提供程序和应用程序服务器绑定了类。最后，告诉Hibernate要在应用程序中使用JTA事务接口设置事务范围。JTATransactionFactory完成几件事：

- 它为JTA启用了正确的Session范围和传播，如果你决定用SessionFactory.getCurrentSession()方法代替手工打开和关闭每个Session的话。我们在11.1节中将更深入地讨论这一特性。
- 它告诉Hibernate你正准备在应用程序中调用JTA UserTransaction接口，来启动、提交或者回滚系统事务。
- 万一你不想使用标准的UserTransaction，它还把Hibernate Transaction API切换到JTA。如果你现在使用Hibernate API启动事务，它会检查是否有JTA事务正在进行中，如有可能，参与这个事务。如果没有JTA事务正在进行，就会启动一个新事务。如果你用Hibernate API提交或者回滚，它要么忽略这个调用（如果Hibernate参与了一个现有的事务），要么设置系统事务为提交或者回滚。如果你在一个支持JTA的环境中部署的话，不建议使用Hibernate Transaction API。然而，这个设置使现有的代码可以在托管和非托管环境之间移植，虽然它可能使用不同的事务行为。

还有其他内建的TransactionFactory选项，并且你可以通过实现这个接口自己编写。JDBCTransactionFactory在非托管环境中是默认的，并且你在本章没有JTA的简单的“Hello World”示例中自始至终都在使用着它。如果使用JTA和EJB，并且计划在托管的EJB组件上声明设置事务范围——换句话说，如果要在一个Java EE应用程序服务器中部署EJB应用，但不在应用程序代码中使用UserTransaction接口编程式地设置事务范围——就应该启用CMTTransactionFactory。

按偏好顺序排列，我们建议的配置选项如下：

- 如果应用程序必须在托管和非托管环境中运行，应该把事务整合和资源管理的任务转给部署程序。在应用程序代码中调用JTA的UserTransaction API，让应用程序的部署程序相应地配置应用程序服务器或者独立的JTA提供程序。在Hibernate配置中启用JTATransactionFactory，与JTA服务整合，并设置正确的查找类。
- 考虑使用EJB组件通过声明设置事务范围。数据访问代码并没有被绑定到任何事务API，CMTTransactionFactory在后台为你整合和处理Hibernate Session。这是最容易的解决方案——当然，现在部署程序有责任提供一个支持JTA和EJB组件和环境。
- 用Hibernate Transaction API编写代码，并通过设置JDBCTransactionFactory或者JTATransactionFactory，让Hibernate在不同的部署环境中切换。注意事务语义可能改变，并且事务的启动或者提交可能导致一个你无法预料的空操作（no-op）。当需要事务划分的可移植性时，这永远是最后一种选择。

**常见问题** 如何使用带有Hibernate的几个数据库？如果要使用几个数据库，就创建几个配置文件。每个数据库都分配它自己的SessionFactory，并且从不同的Configuration对象中构建几个SessionFactory实例。每个Session都从任何SessionFactory中被打开，在JNDI中查找托管数据源。现在协调这些资源就是事务和资源管理器的责任了——Hibernate只在这些数据库连接上执行SQL语句。事务范围用JTA进行编程式地设置，或者由包含EJB的容器和一个声明的集合来处理。

Hibernate不仅可以在JNDI中查找托管资源，还可以把自身绑定到JNDI。接下来我们看看这个特性。

### 2.4.2 JNDI 绑定的 SessionFactory

我们已经遇到了每个Hibernate新用户都必须处理的一个问题：SessionFactory应该如何存储，以及它在应用程序代码中应该如何访问？本章稍早提出过这个问题，通过编写一个在静态字段中存放SessionFactory的HibernateUtil类，并提供静态的getSessionFactory()方法。然而，如果在一个支持JNDI的环境中部署应用程序，Hibernate可以把SessionFactory绑定到JNDI，你需要时可以到那里查找。

**说明** Java命名和目录接口API (JNDI) 允许对象向（从）一个层次结构（目录树）中存储（获取）。JNDI实现注册（registry）模式。基础的对象（事务上下文、数据源等）、配置设置（环境设置、用户注册等），甚至应用对象（EJB引用、对象工厂等）都可以被绑定到JNDI。

如果hibernate.session\_factory\_name属性设置为JNDI节点的名称，Hibernate的SessionFactory就会自动把自身绑定到JNDI。如果运行时环境不提供默认的JNDI上下文（或者如果默认的JNDI实现不支持Referenceable的实例），就需要使用hibernate.jndi.url和hibernate.jndi.class属性指定一个JNDI的初始上下文。

这里有一个例子，说明利用Sun公司（免费的）基于文件系统（file-system-based）的JNDI实现fscontext.jar，把SessionFactory绑定到java:/hibernate/MySessionFactory的Hibernate配置：

```
hibernate.connection.datasource = java:/MyDatasource
hibernate.transaction.factory_class = \
    org.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class = \
    org.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
hibernate.session_factory_name = java:/hibernate/MySessionFactory
hibernate.jndi.class = com.sun.jndi.fscontext.RefFSContextFactory
hibernate.jndi.url = file:/auction/jndi
```

当然，也可以使用基于XML的配置文件。这个例子是不现实的，因为通过JNDI提供连接池

的大部分应用程序服务器，还包含可编写的默认上下文的一个JNDI实现。JBoss AS必然也有，因此可以跳过最后两个属性，只要给SessionFactory指定一个名称即可。

**说明 JNDI和Tomcat**——Tomcat提供一个只读的JNDI上下文，servlet容器启动后，它不可以从应用程序级代码中编写。Hibernate无法绑定到这个上下文：你必须使用一个完全的上下文实现（如Sun FS上下文），或者通过在配置中省略session\_factory\_name属性，禁用SessionFactory的JNDI绑定。

SessionFactory在构建时被绑定到JNDI，也就是调用Configuration.buildSessionFactory()的时候。为了保证应用程序代码的可移植，你可能想要实现这个构建以及HibernateUtil中的查找，并在你的数据访问代码中继续使用辅助类，如代码清单2-18所示。

#### 代码清单2-18 用于SessionFactory的JNDI查找的HibernateUtil

```
public class HibernateUtil { lookup of SessionFactory

    private static Context jndiContext;

    static {
        try {
            // Build it and bind it to JNDI
            new Configuration().buildSessionFactory();

            // Get a handle to the registry (reads jndi.properties)
            jndiContext = new InitialContext();
        } catch (Throwable ex) {
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory(String sfName) {
        SessionFactory sf;
        try {
            sf = (SessionFactory) jndiContext.lookup(sfName);
        } catch (NamingException ex) {
            throw new RuntimeException(ex);
        }
        return sf;
    }
}
```

另一种方法是，可以用一个JNDI调用直接在应用程序代码中查找SessionFactory。然而，在应用程序的某个地方，仍然至少需要启动代码的new Configuration().buildSessionFactory()行。去除Hibernate启动代码的这最后一行，并完全消除HibernateUtil类的一种方法是，把Hibernate部署为一项JMX服务（或者通过使用JPA和Java EE）。

### 2.4.3 JMX 服务部署

Java的世界充满了这些规范、标准和实现。一个相对较新但是很重要的标准是它的第一版：Java管理扩展（Java Management Extension，JMX）。JMX是关于系统组件管理或者系统服务管理的。

Hibernate哪些东西符合这个新标准呢？当Hibernate被部署在一个应用程序服务器中时，它利用其他的服务，就像使用托管的事务和池化数据源（pooled datasource）一样。而且，利用Hibernate JMX整合，Hibernate可以作为一项托管的JMX服务，被其他的服务依赖和使用。

JMX规范定义了下列组件：

- JMX MBean——可以重用（通常是构成基础）的一个组件，给管理公开接口。
- JMX容器（container）——调解对MBean的一般访问（本地的或者远程的）。
- JMX客户端程序（client）——可能通过JMX容器用来管理任何Mbean。

支持JMX的应用程序服务器（例如JBoss AS）表现得就像JMX容器一样，并允许MBean被配置和初始化成应用程序服务器启动过程的一部分。你的Hibernate服务可能被打包和部署为JMX MBean；给这个绑定的接口是org.hibernate.jmx.HibernateService。可以用任何标准的JMX客户端程序通过这个接口启动、停止和监测Hibernate核心。另一个可以选择进行部署的MBean接口是org.hibernate.jmx.StatisticsService，它让你用一个JMX客户端程序启用并监测Hibernate的运行时行为。

JMX服务和MBean如何部署是特定于供应商的。例如，在JBoss应用程序服务器中，你只需要添加jboss-service.xml文件到应用的EAR，把Hibernate部署为一个托管的JMX服务。

这里不对每一个选项进行解释，JBoss应用程序服务器的信息请见参考文档。它包含一个逐步介绍Hibernate整合和部署的小节（<http://docs.jboss.org/jbossas>）。在支持JMX的其他服务器上配置和部署应该与此类似，你可以改写和移植JBoss配置文件。

## 2.5 小结

在本章中，你已经完成了第一个Hibernate项目。我们讨论了如何编写Hibernate XML映射文件，以及可以在Hibernate中调用什么API来与数据库整合。

然后介绍了Java Persistence和EJB 3.0，并阐述了它如何简化最基础的Hibernate应用程序：通过自动的元数据扫描、标准的配置和打包，以及在托管EJB组件中的依赖注入。

如果必须从一个遗留数据库开始，可以使用Hibernate工具集从一个现有的模式中反向工程XML映射文件。或者，如果使用JDK 5.0和（或）EJB 3.0，可以直接从一个SQL数据库中生成Java应用代码。

最后，我们讨论了Java EE环境中更高级的Hibernate整合和配置选项——如果你用的是JPA或者EJB 3.0，它就已经替你完成了整合。

Hibernate功能和Java Persistence之间的高级概述和对比如表2-1所示。（在每一章的最后都可以找到一张类似的对照表。）

表2-1 Hibernate和JPA对照表

Hibernate Core	Java Persistence和EJB 3.0
与所有东西在任何地方整合。灵活，但有时配置复杂	适用于Java EE和Java SE。简单和标准的配置；在Java EE环境中无需额外的整合或者特殊的配置
配置需要一系列XML映射文件或者被注解的类	JPA提供程序自动扫描XML映射文件和被注解的类
私有但是强大。持续改善原生的编程接口和查询语言	标准和稳定的接口，包含一个充分的Hibernate功能的子集。有可能轻松地退回到Hibernate API

第3章将介绍一个更复杂的示例应用程序，本书后续部分将始终使用这个应用程序。你将学习如何设计和实现一个领域模型，以及哪些映射元数据选项在大项目中是最好的选择。