Hibernate In Action

中文版

第二章 引入与集成Hibernate

理解一些Java应用对对象-关系映射的需求是对的，但你可能更急于看到活动的Hibernate。让我们从一个简单的例子开始，展示它的一些能力。

你可能知道，编程书籍通过一个“Hello World”例子开始是一项传统。在本章中，我们将遵循这个传统，使用一个相对简单的“Hello World”程序来介绍Hibernate。然而，简单地在控制台窗口上打印一条消息并不足以实际地展示Hibernate。相反，我们的程序会将新建的对象存储到数据库中，更新它们，并且执行查询从数据库中取出它们。

本章是以后各章的基础。除了这个规范的“Hello World”例子，我们还介绍了核心的Hibernate API，并说明了如何在各种不同的运行环境例如J2EE应用服务器和独立的应用中配置Hibernate。

2. 1 Hibernate中的“Hello World”

Hibernate应用定义了映射到数据库表的持续类。我们的“Hello World”例子由一个类和一个映射文件组成。让我们看一下一个简单的持续类是什么样子的，映射是如何指定的，和我们使用Hibernate持续类的实例可以做的一些其它的事情。

这个简单应用的目标是在数据库中存储消息并且取出它们进行显示。这个应用有一个简单的持续类“Message”，表示了这些用于打印的消息。我们的Message类如清单2.1所示。

```
package hello;
public class Message {
    // 标识符属性
    private Long id;
```

```
    // 消息文本
    private String text;
    // 另一个消息的引用
    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;
    }
}
```

（清单2.1）

我们的Message类有三个属性：标识符属性，消息文本和另一个消息的引用。标识符属性允许应用访问数据库识别——持续性对象的主键。如果两个Message实例具有相同的标识符值，则它们表示了数据库中相同的行。我们将标识符属性的类型定义成了Long，但这并不是必须的。实际上，像你将要看到的一样，Hibernate允许任意的标识符类型。

你可能已经注意到了Message类的所有属性都具有JavaBean风格的属性访问方法。这个类也包含一个没有参数的构造方法。在我们的例子中使用的持续类几乎总是与此有些相似。

Message类的实例可以由Hibernate进行管理（对其持续化），但这也不是必须的。因为Message对象没有实现任何Hibernate特定的类或接口，我们可以像任何其它Java类那样使用它：

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

这段代码精确地完成了我们刚提到过的对“Hello World”应用的期望：将“Hello World”打印到控制台上。这有点像我们在自作聪明；实际上，我们展示了Hibernate区别于其它一些持续性解决方案例如EJB实体Bean的一项重要特征。从根本上讲我们的持续类可以在任意的执行环境中使用——不需要专门的容器。当然，你到这里来是想看一下Hibernate，因此让我们将一个新的Message保存到数据库里：

```
Session session = getSessionFactory().openSession();
Transaction tx = session.beginTransaction();
Message message = new Message("Hello World");
session.save(message);
tx.commit();
session.close();
```

这段代码调用了Hibernate的Session和Transaction接口（很快我们就会开始介绍getSessionFactory()方法）。它与下面这条SQL语句的执行结果相似：

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

继续——Message_ID字段的初始值非常奇怪。我们没有在任何地方设置Message类的id属

性，因此我们可能认为它是null，对吗？实际上，id属性是比较特别的：它是一个标识符属性——它持有一个生成的唯一值（稍候，我们将讨论这个值是如何生成的）。当调用save方法时，这个值通过Hibernate赋给了Message类的实例。

对这个例子来说，我们假定MESSAGES表早已存在了。在第9章，我们将说明只使用映射文件里的信息，如何让Hibernate自动创建你的应用所需的表（不用你手工编写更多的SQL）。当然，我们希望我们的“Hello World”程序能将消息打印到控制台上。现在我们的数据库里已经有了一条消息，我们准备进行展示了。下一个例子按字母顺序从数据库中取出所有的消息，并打印它们：

```
Session newSession = getSessionFactory().openSession();
Transaction newTransaction = newSession.beginTransaction();
List messages =
        newSession.find("from Message as m order by m.text asc");
System.out.println( messages.size() + " message(s) found:" );
for ( Iterator iter = messages.iterator(); iter.hasNext(); ) {
    Message message = (Message) iter.next();
    System.out.println( message.getText() );
}
newTransaction.commit();
newSession.close();
```

文本串“from Message as m order by m.text asc”是一个使用Hibernate自己的查询语言（HQL）表示的查询。当调用find()方法时，这个查询在内部被转化成下面的SQL语句：

```
select m.MESSAGE_ID, m.MESSAGE_TEXT, m.NEXT_MESSAGE_ID
from MESSAGES m
order by m.MESSAGE_TEXT asc
```

下面的消息会被打印出来：

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

如果以前你从没有用过像Hibernate这样的工具，你可能期望在代码或元数据里看到SQL语句。但它们并不存在。所有的SQL都是在运行时（对所有可重用的SQL语句实际是在起动时）生成的。

为了允许这个魔法发生，Hibernate需要更多关于Message类如何被持续化的信息。这些信息通常在XML映射文件里提供。这个映射文件定义了Message类的属性如何映射到MESSAGES表的字段和其它一些信息。让我们看一下清单2.2中的映射文件：

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

（清单2.2）

这个映射文件告诉Hibernate：Message类被持续化到MESSAGES表里，标识符属性映射到名为MESSAGE_ID的字段，文本属性映射到名为MESSAGE_TEXT的字段，名为nextMessage的属性是一个多对一的关联，它映射到名为NEXT_MESSAGE_ID的字段（现在不用担心其它细节）。

像你看到的一样，XML文件并不难以理解。你可以很容易地进行手工编写和维护。第3章，我们将讨论一种根据嵌入在源代码中的注释生成XML文件的方式。无论你选择了哪种方法，Hibernate都有足够的信息完全地生成插入、更新、删除和选取Message实例所需的所有的SQL语句。你不再需要手工编写这些SQL语句。

注意 许多Java开发者都在抱怨伴随着J2EE开发的元数据的“地狱”。许多人建议我们远离XML元数据，返回到普通的Java代码。虽然对有些问题我们赞同这个建议，但ORM代表了一种确实需要基于文本的元数据的情形。Hibernate有一些明显的缺省值这减少了你的键入，并且它还有一个成熟的文档类型定义，可以用于在编辑器中进行自动地生成或验证。你甚至可以使用各种工具自动地生成元数据。

现在，让我们改变一下我们的第一个message，并且在其上创建一个新的关联message，参见清单2.3。

```
Session session = getSessionFactory().openSession();
Transaction tx = session.beginTransaction();
// 1 是第一个消息生成的ID
Message message =
        (Message) session.load( Message.class, new Long(1) );
message.setText("Greetings Earthling");
Message nextMessage = new Message("Take me to your leader (please)");
message.setNextMessage( nextMessage );
tx.commit();
session.close();
```

（清单2.3）

这段代码在同一个事务里调用了三条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如何检测到第一个消息的text和nextMessage属性的修改并且自动更新了数据库。我们已经利用了Hibernate称为自动脏检查的特征：当我们在一个事务里修改了对象的状态时，这个特征节省了我们明确地要求Hibernate更新数据库的努力。相似地，当在第一个消息对象上引用一个新消息对象时，你可以看到新消息对象被持续化了。这个特征被称为级联保存：只要对一个已持续化的实例是可达的，就能够节省我们明确地调用save方法对新对象进行持续化的努力。也要注意到SQL语句的顺序不同于我们设置属性值的顺序。Hibernate使用了一个复杂的算法来确定一个有效的顺序以避免违反数据库的外键约束，但对用户来说，这仍然是充分可预知的。这个特征被称为事务写置后。

如果我们再次运行“Hello World”，它会打印出：

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

这是目前为止我们完成的“Hello World”应用。现在我们终于有了一些代码经历，我们将后退一步介绍Hibernate主要的API预览。

2．2 理解Hibernate的体系结构

为了在你的应用中使用Hibernate，编程接口是你首先需要学习的东西。API设计的一个主要目标是保持软件组件间的接口尽可能小。然而，实际上ORM的API不会特别少。不过也不用担心，你不必马上理解Hibernate的所有接口。图2.1介绍了在业务层和持续层中最重要的一些Hibernate接口的角色。我们将业务层显示在持续层之上，因为在传统的分层应用中业务层担当持续层的一个客户。注意一些简单的应用可能不会明确区分业务层与持续层；这也没有问题——它只是将图形进行了简化。

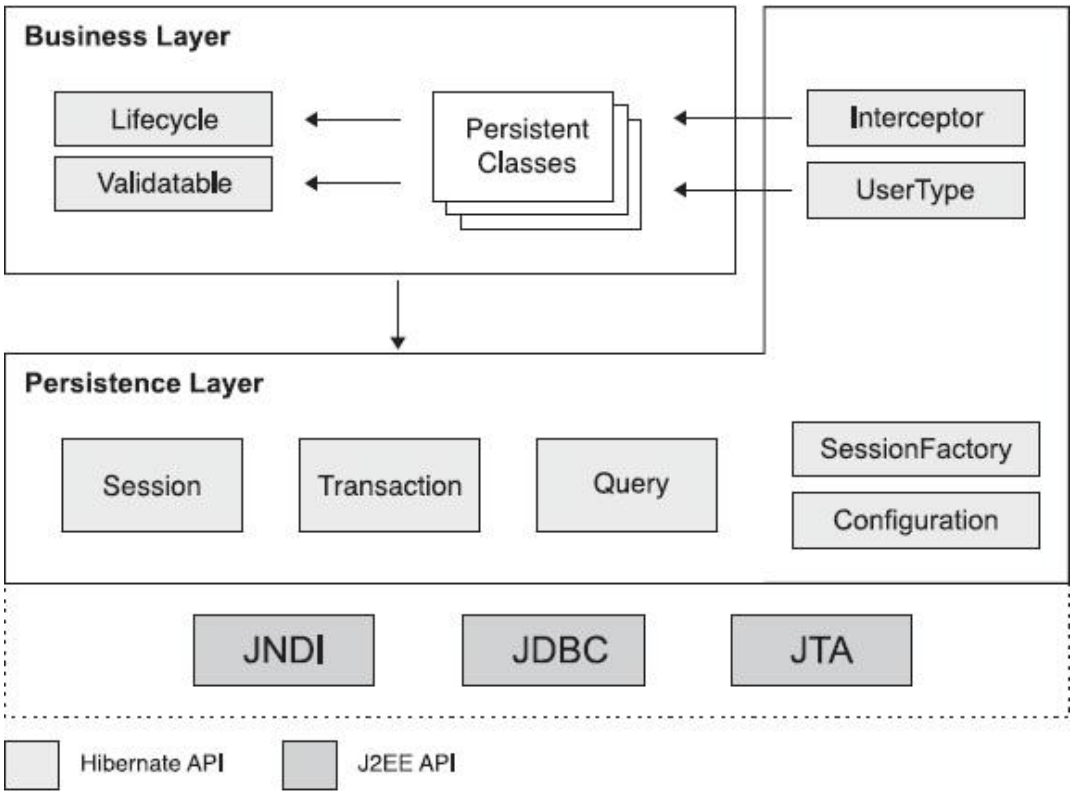

Figure 2.1 High-level overview of the Hibernate API in a layered architecture

（图2.1）

图2.1中显示的接口可以近似地分为如下几类：

■ 由应用调用以完成基本的CRUD和查询操作的接口。这些接口是应用的业务/控制逻辑对Hibernate的主要的依赖点。它们包括Session，Transaction和Query。

■ 由应用的底层代码调用以配置Hibernate的接口，最重要的是Configuration类。

■ 允许应用对Hibernate内部出现的事件进行处理的回调接口，例如Interceptor，Lifecycle和Validatable接口。

■ 允许对Hibernate强大的映射功能进行扩展的接口，例如UserType，CompositeUserType和IdentifierGenerator。这些接口由应用的底层代码实现（如果需要的话）。

Hibernate使用了许多现有的Java API，包括JDBC，Java事务API（JTA）和Java命名和目录接口（JNDI）。JDBC为关系数据库的共通功能提供了一个基本级别的抽象，允许几乎所有具有JDBC驱动的数据库被Hibernate支持。JNDI和JTA允许Hibernate与J2EE应用服务器进行集成。

在本节中，我们并不包含Hibernate API方法的详细语义，只是介绍每一个主要接口的角色。你可以在net.sf.hibernate包中找到这些接口中的大部分。让我们依次简单地看一下每一个接口。

2．2．1 核心接口

有五个核心接口几乎在每个Hibernate应用中都会用到。使用这些接口，你可以存储与取出持续对象或者对事务进行控制。

Session接口

Session（会话）接口是Hibernate应用使用的主要接口。会话接口的实例是轻量级的并且创建与销毁的代价也不昂贵。这很重要因为你的应用可能始终在创建与销毁会话，可能每

一次请求都会如此。Hibernate会话并不是线程安全的因此应该被设计为每次只能在一个线程中使用。

Hibernate会话是一个介于连接和事务之间的概念。你可以简单地认为会话是对于一个单独的工作单元已装载对象的缓存或集合。Hibernate可以检测到这个工作单元中对象的改变。我们有时也将会话称为持续性管理器，因为它也是与持续性有关的操作例如存储和取出对象的接口。注意，Hibernate会话与Web层的HttpSession没有任何关系。当我们在本书中使用会话时，我们指的是Hibernate会话。为了区别，有时我们将HttpSession对象称为用户会话。

我们将在第4章第4.2节“持续性管理器”中详细地描述会话接口。

SessionFactory接口

应用从SessionFactory（会话工厂）里获得会话实例。与会话接口相比，这个对象不够令人兴奋。

会话工厂当然不是轻量级的！它打算在多个应用线程间进行共享。典型地，整个应用只有唯一的一个会话工厂——例如在应用初始化时被创建。然而，如果你的应用使用Hibernate访问多个数据库，你需要对每一个数据库使用一个会话工厂。

会话工厂缓存了生成的SQL语句和Hibernate在运行时使用的映射元数据。它也保存了在一个工作单元中读入的数据并且可能在以后的工作单元中被重用（只有类和集合映射指定了这种二级缓存是想要的时才会如此）。

Configuration接口

Configuration（配置）对象用来配置和引导Hibernate。应用使用一个配置实例来指定映射文件的位置和Hibernate的特定属性，然后创建会话工厂。

即使配置接口只担当了整个Hibernate应用范围内一个相对较小的部分，但它却是在你开始使用Hibernate时遇到的第一个对象。第2.3节比较详细地介绍了一些配置Hibernate的问题。

Transaction接口

Transaction（事务）接口是一个可选的API。Hibernate应用可以选择不使用这个接口，而是在它们自己的底层代码中管理事务。事务将应用代码从下层的事务实现中抽象出来——这可能是一个JDBC事务，一个JTA用户事务或者甚至是一个公共对象请求代理结构（CORBA）——允许应用通过一组一致的API控制事务边界。这有助于保持Hibernate应用在不同类型的执行环境或容器中的可移植性。

我们自始至终在本书中使用Hibernate事务API。事务与事务接口在第5章进行说明。

Query与Criteria接口

Query（查询）接口允许你在数据库上执行查询并控制查询如何执行。查询使用HQL或者本地数据库的SQL方言编写。查询实例用来绑定查询参数，限定查询返回的结果数，并且最终执行查询。

Criteria（标准）接口非常小，它允许你创建和执行面向对象的标准查询。

为了帮助应用代码减少冗余，Hibernate在会话接口上提供了一些快捷方法，允许你可以在一行代码内调用一个查询。在本书中我们不使用这些快捷方法，相反，我们会一直使用查询接口。

查询实例是轻量级的并且不能在创建它的会话外使用。我们将在第7章描述查询接口的这个特征。

2．2．2 回调接口

当一个对象发生了应用感兴趣的事情——例如，当一个对象被装载、保存或删除时，回调接口允许应用可以接收到通知。Hibernate应用并不必需实现这些回调，但是在实现特定类型的功能（例如创建审计记录）时，它们非常有用。

接口Lifecycle和Validatable允许持续对象对与其有关的生命周期事件做出反应。持续性生命周期由对象的CRUD操作构成。Hibernate开发组受到了其它具有相似回调接口的ORM解决方案的很深的影响。后来，他们认识到让持续类实现Hibernate的特定接口可能不是一个好主意，因为这样做污染了我们的持续类使其成为了不可移植的代码。因此这些方法不再赞成使用，我们也不在本书中讨论它们。

引入接口Interceptor是为了允许应用处理回调而又不用强制持续类实现Hibernate特定的API。接口Interceptor的实现被作为参数传递给持续类的实例。我们将在第8章讨论一个这样的例子。

2．2．3 类型

一个基础的并且非常强大的体系结构元素是Hibernate的类型的概念。Hibernate的类型对象将一个Java类型映射到数据库字段的类型（实际上，类型可能跨越多个字段）。持续类所有的持续属性，包括关联，都有一个对应的Hibernate类型。这种设计使Hibernate变得极端灵活并易于扩展。

内建类型的范围非常广泛，覆盖了所有的Java基础类型和许多JDK类，包括java.util.Currency，java.util.Calendar，byte[]和java.io.Serializable。

甚至更好一些，Hibernate支持用户自定义类型。它提供了UserType和CompositeUserType

接口允许你增加自己的类型。使用这个特征，应用使用的共通类例如Address，Name或MonetaryAmount就可以方便优雅地进行处理了。自定义类型被认为是Hibernate的重要特征，并鼓励你对它们进行新的或创造性的使用。

我们将在第6章第6.1节“理解Hibernate的类型系统”中介绍Hibernate类型和用户自定义类型。

2．2．4 扩展接口

Hibernate提供的大多数功能都是可配置的，允许你在一些内置的策略中进行选择。当内置策略不能满足需要时，Hibernate通常会允许你通过实现一个接口插入你自己的定制实现。扩展点包括：

- 主键生成（IdentifierGenerator接口）
- SQL方言支持（Dialect抽象类）
- 缓存策略（Cache和CacheProvider接口）
- JDBC连接管理（ConnectionProvider接口）
- 事务管理（TransactionFactory，Transaction和TransactionManagerLookup接口）
- ORM策略（ClassPersister接口层次）
- 属性访问策略（PropertyAccessor接口）
- 代理创建（ProxyFactory接口）

对上面列出的每一个接口Hibernate至少都自带了一种实现，因此如果你想对内置功能进行扩展的话通常不需要你从头开始。对于自己的实现，你可以将源代码作为例子来使用。

现在你可以看到在我们开始使用Hibernate编写任何代码之前我们必须回答这个问题：我们如何取得一个需要的会话呢？

2．3 基本配置

我们已经看了一个应用的例子并且分析了Hibernate的核心接口。为了在应用中使用Hibernate，你必须知道如何配置它。Hibernate可以配置在几乎所有的Java应用和开发环境中运行。通常，Hibernate在两层或三层的客户/服务器应用中使用，而且只配置在服务器端。客户端应用通常是一个Web浏览器，Swing和SWT的客户端应用并不常见。虽然在本书中我们集中在多层的Web应用上，我们的说明也可以等同地应用到其它体系结构上，例如命令行应用。理解在管理与非管理环境中配置Hibernate的区别是非常重要的：

■ 管理环境——将资源例如数据库连接进行池化并且允许事务边界和安全通过声明进行指定（即使用元数据）。J2EE应用服务器例如JBoss，BEA WebLogic或IBM Websphere都实现了标准的管理环境（J2EE特定的）。

■ 非管理环境——通过线程池提供了基本的并发管理。像Jetty或Tomcat这样的Servlet容器为Java Web应用提供了非管理的服务器环境。独立的桌面或命令行应用也被认为是非管理的。非管理环境并不提供自动的事务、资源管理和底层的安全结构。应用自己管理数据库连接并划分事务界限。

Hibernate试图抽象它的配置环境。在非管理环境的场合，Hibernate自己处理事务和JDBC连接（或者委托应用代码处理这些问题）。在管理环境的场合，Hibernate与容器管理的事务和数据源相结合。在两种环境中都可以根据配置来设定Hibernate。

在管理与非管理两种环境中，你必须作的第一件事都是起动Hibernate。实际上，这非常简单。你只要根据配置创建一个会话工厂即可。

2．3．1 创建会话工厂

为了创建一个会话工厂，在应用初始化时你首先要创建一个单独的配置实例并使用它设置映射文件的位置。一旦配置完成，你就可以使用配置实例创建会话工厂了。在会话工厂创建以后，你可以丢弃配置类。

下面的代码起动了Hibernate：

```
Configuration cfg = new Configuration();
cfg.addResource("hello/Message.hbm.xml");
cfg.setProperties( System.getProperties() );
SessionFactory sessions = cfg.buildSessionFactory();
```

映射文件（Message.hbm.xml）的位置是相对于应用的classpath的。例如，如果classpath是当前目录，则Message.hbm.xml文件必须位于hello目录下。XML映射文件必须位于classpath里。在这个例子里，我们也使用了虚拟机的系统属性来设置其它的配置选项（这些选项可能早已被应用代码或者作为起动选项设置过了）。

方法链 方法链是许多Hibernate接口支持的编程风格。这种风格在Smalltalk中比在Java中更流行，但是许多人认为它与普通的Java风格相比更不易阅读并且更难于调试。然而，在大多数情况下，它都非常方便。

大多数Java开发者都将设置或计算方法的类型设置为void，这意味着它们没有返回值。在Smalltalk里没有void类型，设置或计算方法通常将接收的对象返回。这就允许我们如下重写前面例子的代码：

```
SessionFactory sessions = new Configuration()
    .addResource("hello/Message.hbm.xml")
    .setProperties( System.getProperties() )
    .buildSessionFactory();
```

注意我们不再需要为配置类声明局部变量了。我们在一些代码例子中使用了这种风格；

但如果你不喜欢，你自己并不需要使用。如果你确实使用了这种编码风格，最好将每个方法调用写在不同的行上。否则，在调试器里单步调试代码可能会比较困难。

依照惯例，Hibernate的XML映射文件以.hbm.xml扩展名命名。另一个惯例是每个类一个映射文件，而不是将所有的映射列在一个文件中（这是可能的但被认为是很差的风格）。我们的例子“Hello World”只有一个持续类，但假定我们有多个持续类，并且每个类一个XML映射文件，我们应该将这些映射文件放在哪里呢？

Hibernate文档推荐每个持续类的映射文件放在与映射类相同的目录中。例如，Message类的映射文件应该放在hello目录中，并且命名为Message.hbm.xml。如果我们有另一个持续类，它应该被定义在自己的映射文件中。我们建议你遵循这项实践。许多框架鼓励单一的元数据文件，例如在struts中建立的struts-config.xml，就是“元数据地狱”的一个主要贡献者。每当需要时你可以调用addResource()装载多个映射文件，或者，如果你遵循刚刚描述过的惯例，你可以使用addClass()方法，传递一个持续类作为参数：

```
SessionFactory sessions = new Configuration()
    .addClass(org.hibernate.auction.model.Item.class)
    .addClass(org.hibernate.auction.model.Category.class)
    .addClass(org.hibernate.auction.model.Bid.class)
    .setProperties( System.getProperties() )
    .buildSessionFactory();
```

AddClass()方法假定映射文件的文件名以.hbm.xml扩展名结尾并且与被映射的类文件配置在一起。

我们已经展示了单独的会话工厂的创建，它是大多数应用需要的全部内容。如果你需要另一个会话工厂——例如，有多个数据库的情形——你只需要重复这个过程即可。每个会话工厂仅用于一个数据库，并且准备生成会话，用来与特定的数据库和一组类映射一起工作。

当然，还有更多的内容需要配置而不仅仅是指明映射文件。你也需要指定如何获得数据库连接，以及其它各种在运行时影响Hibernate行为的设置。多数配置属性可能会压倒性的出现（在Hibernate文档中包含一个完全的列表），但是不用担心，大多数属性都定义了合理的缺省值，并且只有少数是普遍需要的。

为了指定配置选项，你可以使用下列任何技术：

- 传递一个java.util.Properties实例到Configuration.setProperties()。
- 使用java -Dproperty=value设置系统属性。
- 将一个名为hibernate.properties的文件放置在classpath中。
- 在classpath中的hibernate.cfg.xml文件里包含<property>元素。

第一、二个选项除了快速测试和原型外很少使用，大多数应用需要一个固定的配置文件。文件hibernate.properties和hibernate.cfg.xml提供了配置Hibernate的相同的功能。选择使用哪个文件依赖于你的语法爱好。像你将要在本章中看到的一样，也可能混合使用这两种选择并且对开发和配置使用不同的设置。

一种罕见的使用选择是允许应用从会话工厂中打开一个Hibernate会话时提供JDBC连接（例如通过调用sessions.openSession(myConnection)）。使用这种选择意味着你不必指定任何数据库连接属性。对新应用我们不推荐这种方法，因为它们可以使用环境中的数据库连接的底层构造（例如，JDBC连接池或者应用服务器数据源）进行配置。

在所有的配置选项中，数据库连接的设置是最重要的。它们在管理与非管理环境中的设置是不同的，因此我们独立地处理这两种情况。让我们首先从非管理环境开始。

2．3．2 在非管理环境中配置

在非管理环境例如Servlet容器中，应用负责取得JDBC连接。Hibernate是应用的一部分，因此，它会负责取得这些连接。但是需要你告诉Hibernate如何取得（或创建）JDBC连接。通常，每次与数据库交互都创建一个连接的做法是不可取的。相反，Java应用应该使用JDBC连接池。这有三方面的原因：

- 获得新连接的代价是很昂贵的。
- 维护许多无用的连接也是很浪费的。
- 对许多驱动程序来说创建预编译语句同样是很昂贵的。

图2.2显示了JDBC连接池在Web应用运行环境中的角色。因为非管理环境没有实现连接池，所以应用必须实现自己的池化算法或者依赖于第三方类库例如开源的C3P0连接池。不使用Hibernate时，应用代码通常直接调用连接池来获得JDBC连接并执行SQL语句。

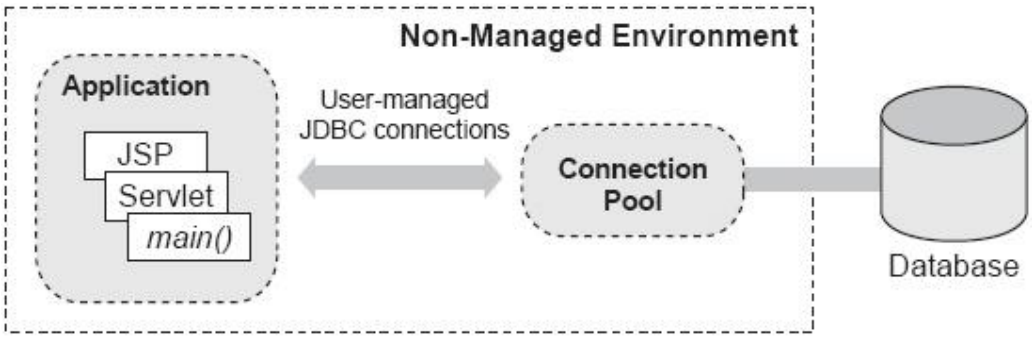

Figure 2.2 JDBC connection pooling in a non-managed environment

（图2.2）

使用Hibernate时，图像发生了改变：它被作为JDBC连接池的一个客户，如图2.3所示。应用代码使用Hibernate会话和查询API进行持续性操作并且理论上只需要使用Hibernate事务API管理数据库事务。

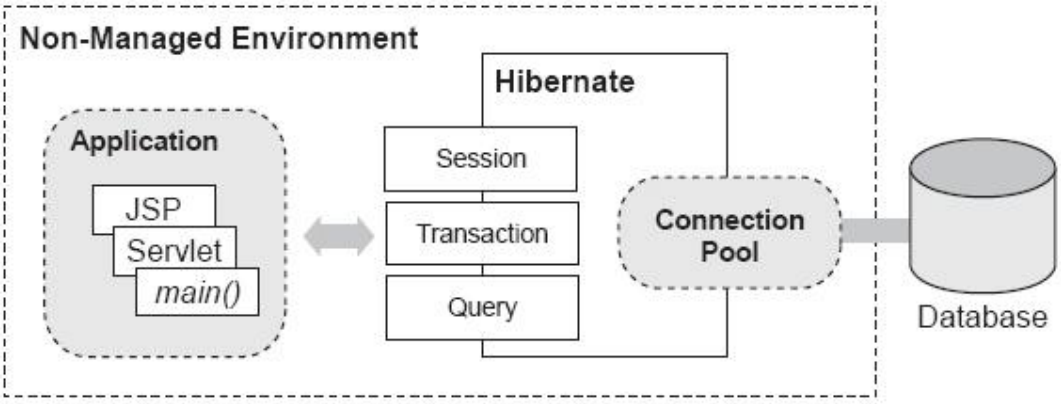

Figure 2.3 Hibernate with a connection pool in a non-managed environment

（图2.3）

使用连接池

Hibernate定义了一个插件体系结构，允许你与任何连接池集成。然而，Hibernate内置了对C3P0的支持，因此我们将使用它。Hibernate使用特定的属性为你建立了连接池。一个使用了C3P0的hibernate.properties文件的例子参见清单2.4。

```
hibernate.connection.driver_class = org.postgresql.Driver
hibernate.connection.url = jdbc:postgresql://localhost/auctiondb
hibernate.connection.username = auctionuser
hibernate.connection.password = secret
hibernate.dialect = net.sf.hibernate.dialect.PostgreSQLDialect
hibernate.c3p0.min_size = 5
hibernate.c3p0.max_size = 20
hibernate.c3p0.timeout = 300
hibernate.c3p0.max_statements = 50
hibernate.c3p0.idle_test_period = 3000
```

（清单2.4）

从第一行开始，以上代码分别指定了下面这些信息：

■ 实现JDBC驱动程序的Java类的名称（驱动程序的jar文件必须位于应用的classpath中）。

■ 指定JDBC连接的主机名和数据库名的JDBC URL。

■ 数据库用户名。

■ 指定用户的数据库密码。

■ 一种数据库方言。尽管有ANSI标准化的努力，不同的数据库厂商还是有不同的SQL实现。因此，你必须指定一种方言。Hibernate对所有流行的SQL数据库都包含内建的支持，并且也很容易定义新的方言。

■ C3P0保持的JDBC连接的最小值。

■ 池中连接的最大值。如果在运行时这个值被耗尽则会抛出一个异常。

■ 一个超时时间（在本例中，是5分钟或300秒），在这之后空闲的连接会被从池中删除。

■ 可以被缓存的预编译语句的最大数量。对预编译语句进行缓存是高性能使用Hibernate必需的。

■ 连接自动生效前的空闲时间（以秒为单位）。

以hibernate.c3p0.*格式指定的属性选择了C3P0作为Hibernate的连接池（为了能支持C3P0，你不需要再设置任何其它选项）。C3P0有比我们在前面的例子中展示的更多的特征，因此我们建议你查询Hibernate API文档。类net.sf.hibernate.cfg.Environment的Javadoc文档化了Hibernate的每一个配置属性，包括所有C3P0相关的设置和其它Hibernate直接支持的第三方连接池的设置。

其它支持的连接池是Apache DBCP和Proxool。在你决定之前，你应该首先在你的环境里试验一下每一个连接池。不过，Hibernate社团更倾向于C3P0和Proxool。

Hibernate也包含了一个缺省的连接池机制。这个连接池只适用于测试或试验Hibernate。在产品系统中，你不应该使用这个内建的池。在并发请求较多的环境中它的设计是不可扩展的，并且它缺乏一些专业的连接池才具有的容错特征。

起动Hibernate

如何使用这些属性起动**Hibernate**？你在一个名为**hibernate.properties**的文件中声明这些属性，因此你只需要将这个文件放在应用的**classpath**中即可。当你创建了一个配置对象，**Hibernate**首次进行初始化时，它会被自动地检测和读取。

让我们总结一下目前为止你已经学到的配置步骤（如果你想在非配置环境下继续的话，这是下载和安装**Hibernate**的一个很好的时机）。

1．下载并解包你的数据库的**JDBC**驱动程序，这通常可以从数据库厂商的**Web**站点得到。将**JAR**文件放到应用的**classpath**中；同时也将**hibernate2.jar**放在相同的位置。

2．将**Hibernate**的依赖包放到**classpath**中；它们在**lib/**目录中与**Hibernate**一起发布。同时也要看一下**lib/**目录下的文本文件**README.txt**，它包含一个必需与可选库的列表。

3．选择一个**Hibernate**支持的**JDBC**连接池并使用属性文件进行配置。不要忘记指定**SQL**方言。

4．通过将它们放在一个位于**classpath**中的**hibernate.properties**文件中，让配置对象知道这些属性。

5．在你的应用中创建一个配置对象的实例并使用**addResource()**或**addClass()**方法装载**XML**映射文件。通过调用配置对象的**buildSessionFactory()**方法创建一个会话工厂。

很不幸，你还没有任何映射文件。如果你喜欢，你可以运行“**Hello World**”的例子或者跳过本章剩余的部分并开始学习第**3**章持续类与映射。或者，如果你想知道更多关于在管理环境中使用**Hibernate**的知识，请继续往下读。

2．3．3 在管理环境中配置

管理环境处理特定的“**cross-cutting**”关系，例如应用安全（授权与验证），连接池和

事务管理。J2EE应用服务器是典型的管理环境。虽然应用服务器通常是为了支持EJB而设计的，但即使你不使用EJB实体Bean，你仍然可以利用它提供的其它服务。

Hibernate经常与会话或消息驱动EJB一起使用，如图2.4所示。像servlet、JSP和独立的应用一样，EJB调用相同的Hibernate API：Session（会话）、Transaction（事务）和Query（查询）。与Hibernate相关的代码在非管理与管理环境之间是完全可移植的。Hibernate透明地处理了不同的连接与事务策略。

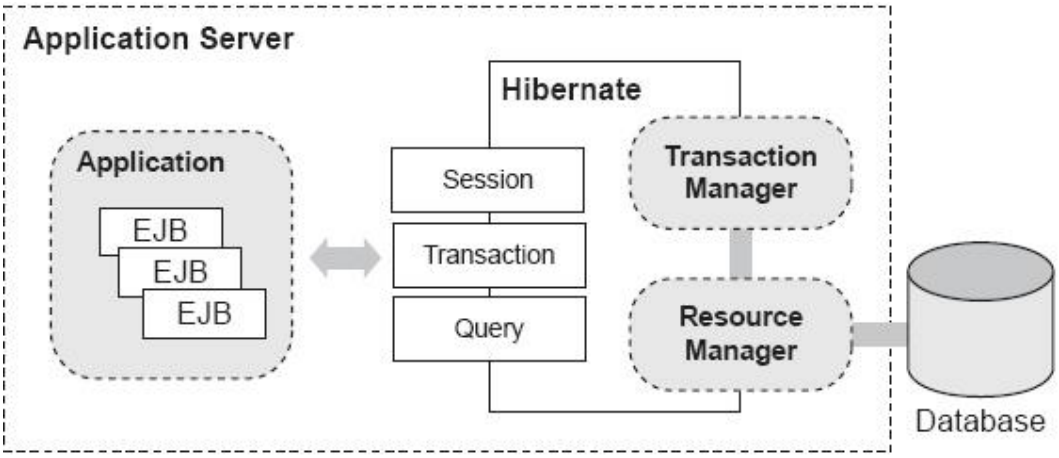

Figure 2.4 Hibernate in a managed environment with an application server

（图2.4）

应用服务器将连接池对外显示为JNDI绑定数据源，它是javax.jdbc.Datasource类的一个实例。你需要提供一个JNDI全限定名来告诉Hibernate，到哪里去查找JNDI数据源。这种情况下的一个Hibernate配置文件的例子参见清单2.5。

```
hibernate.connection.datasource = java:/comp/env/jdbc/AuctionDB
hibernate.transaction.factory_class = \
    net.sf.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class = \
    net.sf.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = net.sf.hibernate.dialect.PostgreSQLDialect
```

（清单2.5）

这个文件首先给出了数据源的JNDI名。数据源必须在J2EE企业应用的配置描述符中进行配置；这是一个厂商特定的设置。接着，你允许Hibernate与JTA集成。为了与容器事务完全集成， Hibernate需要定位应用服务器的事务管理器。J2EE规范没有定义标准的方法，但是Hibernate包含了对所有流行的应用服务器的支持。当然，最后还需要指定SQL方言。

现在你已经正确地配置了每件事情，在管理环境中使用Hibernate与在非管理环境中没有太多的区别：都只是使用映射创建一个配置对象并构造一个会话工厂。然而，许多与事物环境有关的设置值得进行额外的考虑。

Java早就有了一个标准的事务API：JTA，它用来在J2EE管理环境中控制事务。这被称作容器管理的事务（CMT）。如果存在JTA事务管理器，JDBC连接将被它支持并完全位于它的控制之下。这与非管理环境不同，在非管理环境中应用（或连接池）直接管理JDBC连接和JDBC事务。

因此，管理环境与非管理环境可以使用不同的事务方法。因为Hibernate需要在这两种环境之间保证可移植性，因此它定义了一组控制事务的API。Hibernate的事务接口抽象了下层的JTA或JDBC事务（或者，甚至潜在的CORBA事务）。这个下层的事务策略可以使用属性hibernate. connection.factory_class来设置，它可以取下列两值之一：

■ net.sf.hibernate.transaction.JDBCTransactionFactory代表了直接的JDBC事务。这种策略应该与非管理环境中的连接池一起使用，并且如果没有指定任何策略时它就是缺省值。

■ net.sf.hibernate.transaction.JTATransactionFactory代表了JTA。对于CMT这是正确的策略，此时连接由JTA支持。注意如果调用beginTransaction()时已经有一个JTA事务在

运行，随后的工作将发生在那个事务的上下文中（否则，将会开始一个新的JTA事务）。

对Hibernate事务API更详细的介绍与它对特定应用场景的影响，请参考第5章第5.1节“事务”。你只要记住使用J2EE服务器工作时必需的两个步骤：像前面描述的那样为Hibernate事务API设置工厂类以支持JTA，并且声明你的应用服务器特定的事务管理器的查找策略。只有你使用了Hibernate的二级缓存系统时查找策略才是必需的，但即使你没有使用缓存设上它也没有坏处。

Hibernate与Tomcat Tomcat并不是一个完整的应用服务器；它只是一个Servlet容器，尽管它包含一些通常只有在应用服务器中才能找到的特征。其中的一个特征可能被Hibernate使用：Tomcat的连接池。Tomcat内部使用的是DBCP连接池，但也像真正的应用服务器一样将它作为JNDI数据源对外显示。为了配置Tomcat数据源，你需要根据Tomcat JNDI/JDBC文档的指导编辑server.xml文件。你可以设置hibernate.connection.datasource来配置Hibernate使用这个数据源。注意Tomcat并不包含一个事务管理器，因此这种情形更像以前描述的非管理环境。

无论使用的是一个简单的Servlet容器还是一个应用服务器，现在你应该有了一个正在运行的Hibernate系统。创建并编译一个持续类（例如，最初的Message类），将Hibernate和它需要的类库以及文件hibernate.properties拷贝到classpath中，构造一个会话工厂。

下一节包含高级的Hibernate配置选项。其中许多是推荐使用的，例如将可执行的SQL语句写入日志以利于调试；或者使用方便的XML配置文件而不是无格式的属性。然而，你可以安全的跳过这一节而直接去读第三章，在你学到更多关于持续类的知识后再回来。

2．4 高级配置设置

当你最终有了一个可以运行的Hibernate应用时，对Hibernate所有的配置参数有一个全面的了解是非常值得的。这些参数允许你优化Hibernate运行时的行为，特别是调整与JDBC的交互（例如，使用JDBC进行批处理更新时）。

现在我们不会让你对这些细节感到厌烦；关于配置选项最好的信息来源当然是Hibernate的参考文档。在上一节里，我们已经向你说明了一些开始时需要知道的选项。

然而，有一个参数我们必须在这里强调一下。无论何时你使用Hibernate开发软件时都会频繁地用到它。将属性hibernate.show_sql设置为true就可以将所有生成的SQL记录到控制台上。你可以使用它来诊断故障，调整性能或者只是看看生成了什么。它有利于你知道ORM层正在做什么——这就是为什么ORM不对开发者隐藏SQL的原因。

到目前为止，我们一直假定你在使用hibernate.properties文件或者在程序里使用一个java.util.Properties的实例来指定配置参数。除此之外，还有你可能会喜欢的第三种选择：使用XML配置文件。

2．4．1 使用基于XML的配置

你可以使用一个XML配置文件（如清单2.6所示）来完全地配置一个会话工厂。不像只包含配置参数的文件hibernate.properties，文件hibernate.cfg.xml也可以指定映射文件的位置。许多用户更喜欢以这种方式集中Hibernate的配置，而不是在应用代码中往配置对象里增加参数。

```
<?xml version='1.0'encoding='utf-8'?>
<!DOCTYPE hibernate-configuration                                    ①
    PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">
<hibernate-configuration>
```

```
    <session-factory name="java:/hibernate/HibernateFactory">                    ②
        <property name="show_sql">true</property>
        <property name="connection.datasource">                                  ③
            java:/comp/env/jdbc/AuctionDB
        </property>
        <property name="dialect">
            net.sf.hibernate.dialect.PostgreSQLDialect
        </property>
        <property name="transaction.manager_lookup_class">
            net.sf.hibernate.transaction.JBossTransactionManagerLookup
        </property>
        <mapping resource="auction/Item.hbm.xml"/>                               ④
        <mapping resource="auction/Category.hbm.xml"/>
        <mapping resource="auction/Bid.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

（清单2.6）

① 文档类型声明，XML解析器根据它声明的Hibernate配置DTD来验证文档。

② 可选的name属性等价于属性hibernate.session_factory_name，它用于会话工厂的JNDI绑定，将在下一节进行讨论。

③ 可以不使用“hibernate”作为前缀指定的Hibernate属性。属性名与值在其它方面与程序化的配置属性是相同的。

④ 映射文件可以作为应用资源甚至是硬编码的文件名来指定。这里使用的文件来自于我们的在线拍卖应用，我们将在第3章进行介绍。

现在你可以使用下面的语句初始化Hibernate了：

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

稍等——Hibernate如何知道配置文件位于哪里？

当调用configure()方法时，Hibernate在classpath中查找名为hibernate.cfg.xml的文件。如果你想使用一个不同的文件名或者想让Hibernate在一个子目录中进行查找，你必须将路径传递给configure()方法。

```
SessionFactory sessions = new Configuration()
    .configure("/hibernate-config/auction.cfg.xml")
    .buildSessionFactory();
```

使用XML配置文件的确比属性文件甚至是程序化的属性配置让人感觉更舒适。这种方法主要的好处是可以让类映射文件从应用的源代码中（即使它仅出现在起动时的帮助类中）具体地表示出来。举例来说，你可以对不同的数据库和环境（开发或产品）使用不同的映射文件（和不同的配置选项），并通过程序切换它们。

如果在classpath中同时存在hibernate.properties和hibernate.cfg.xml两个文件，则XML配置文件的设置会重载属性文件。如果你想在属性文件中保持一些基本设置并且在每个部署中使用XML配置文件重载它们时这非常有用。

你可能已经注意到了在XML配置文件中同时给会话工厂指定了一个名字。在会话工厂创建之后，Hibernate自动地使用这个名字将它绑定到JNDI。

2．4．2 绑定到JNDI的会话工厂

在大多数Hibernate应用中，在应用初始化期间会话工厂都会被实例化一次。然后这个唯一的实例会被一个特定过程的所有代码使用，并且所有的会话都由这个唯一的会话工厂创建。一个经常询问的问题是这个工厂应该放到什么地方并且如何轻而易举地访问它。

在J2EE环境中，绑定到JNDI的会话工厂可以很容易地在不同的线程和各种支持Hibernate的组件之间进行共享。当然，JNDI并不是应用组件能够获得会话工厂的唯一方式。这种注册

模式有多种可能的实现，包括使用ServletContext或者单体中的static final变量。一种特别优雅的方法是使用一个应用范围的IoC（控制反转）框架组件。然而，JNDI是一种比较流行的方法（稍后你将看到，它作为JMX服务对外显示）。我们将在第8章第8.1节“设计分层的应用”中讨论一些其它的可选方案。

注意 Java命名与目录接口（JNDI）API允许从一个层次结构（目录树）中存储与取出对象。JNDI实现了注册模式。底层对象（事务上下文，数据源），配置设置（环境设置，用户注册）甚至是应用对象（EJB引用，对象工厂）都可以绑定到JNDI。

如果为属性hibernate.session_factory_name设置了目录节点的名称，会话工厂会自动地将它自己绑定到JNDI。如果你的运行环境没有提供一个缺省的JNDI上下文（或者缺省的JNDI实现不支持Referenceable的实例），你需要使用属性hibernate.jndi.url和hibernate.jndi.class指定JNDI的初始上下文。

这儿有一个Hibernate配置的例子，它使用Sun的（免费的）基于文件系统的JNDI实现（fscontext.jar），将会话工厂绑定到了名称hibernate/HibernateFactory上：

```
hibernate.connection.datasource = java:/comp/env/jdbc/AuctionDB
hibernate.transaction.factory_class = \
    net.sf.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class = \
    net.sf.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = net.sf.hibernate.dialect.PostgreSQLDialect
hibernate.session_factory_name = hibernate/HibernateFactory
hibernate.jndi.class = com.sun.jndi.fscontext.RefFSContextFactory
hibernate.jndi.url = file:/auction/jndi
```

当然，对这个任务你也可以使用基于XML的配置。这个例子也不太现实，因为大多数通过JNDI提供连接池的应用服务器都会有一个JNDI的可写的缺省上下文的实现。JBoss当然也有，

因此你可以跳过最后两个属性仅指定一个会话工厂名即可。现在你需要做的所有的事情是调用一次Configuration.configure().buildSessionFactory()来对绑定进行初始化。

注意 Tomcat带有一个只读的JNDI上下文，在Servlet容器起动之后对应用级别的代码来说它是不可写的。Hibernate不能绑定到这种类型的上下文上；你必须或者使用一个完全的上下文的实现（例如Sun的文件系统上下文）或者忽略配置属性session_factory_name来禁用会话工厂的JNDI绑定。

让我们看一些非常重要的用于记录Hibernate操作日志的其它的配置设定。

2．4．3 日志

Hibernate（与许多其它的ORM实现）异步地执行SQL语句。通常当应用调用Session.save()时并不会执行INSERT语句，当应用调用Item.addBid()时也不会立即执行UPDATE语句。相反，SQL语句通常都在事务的终点执行。这种行为被称作写置后，我们以前曾经提到过。

有时当跟踪与调试ORM代码很重要时，这个事实也会变得很明显。理论上，应用将Hibernate作为黑盒看待而忽略它的行为是可能的。当然Hibernate应用不会觉察到这种异步（至少不对直接的JDBC调用进行重新排序时不会）。然而，当你遇到一个难题时，你需要能够精确地看到Hibernate内部发生了什么。因为Hibernate是开源的，你可以很容易地单步调试Hibernate的代码。有时这样做很有帮助！但是，特别是面对异步行为时，调试Hibernate会很快地让你迷失。此时，你可以使用日志来得到Hibernate的内部视图。

我们早已提到过hibernate.show_sql这个配置参数，当遇到麻烦时它通常是你能想到的第一个入口。有时只有SQL是不够的，那时，你必须钻得更深一些。

Hibernate使用Apache的commons-logging将所有感兴趣的事件记入了日志。

commons-logging是很薄的一个抽象层，它直接输出到Apache的log4j里（如果你在classpath里放入了log4j.jar）或者是JDK1.4的logging里（如果你运行在JDK1.4以上并且log4j不存在时）。我们推荐log4j，因为它更成熟，更流行，并且开发也更活跃。

为了看到log4j的输出，你需要将一个名为log4j.properties的文件放到classpath里（紧挨着hibernate.properties或者hibernate.cfg.xml）。下面的例子指示将所有日志信息输出到控制台上：

```
### 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=warn, stdout
### Hibernate logging options ###
log4j.logger.net.sf.hibernate=info
### log JDBC bind parameters ###
log4j.logger.net.sf.hibernate.type=info
### log PreparedStatement cache activity ###
log4j.logger.net.sf.hibernate.ps.PreparedStatementCache=info
```

使用这个配置，你不会在运行时看到太多的日志信息。将log4j.logger.net.sf.hibernate的类型由info替换为debug将会显示出Hibernate的内部工作状况。确保你在产品环境中不会这么做——写日志比实际的数据库访问要慢很多。

最终，你有了hibernate.properties，hibernate.cfg.xml和log4j.properties三个配置文件。

如果你的应用服务器支持Java管理扩展的话，还有另外一种配置Hibernate的方式。

2．4．4 Java管理扩展（JMX）

在Java世界里充满了规范、标准当然也包括它们的实现。一个相对较新但又非常重要的标准有了它的初版：Java管理扩展（JMX）。JMX是关于系统组件或者更好一点是关于系统服务的管理。

Hibernate怎样适应这种新的情形？当配置到应用服务器中时，Hibernate使用了像受管理的事务与池化的数据库事务等一些其它的服务。但是为什么不让Hibernate本身成为一个受管理的服务，让其它服务可以依赖或使用它？使用Hibernate的JMX集成，使Hibernate成为一个受管理的JMX组件是可以做到这一点的。

JMX规范定义了下面的组件：

- JMX MBean——揭示了管理接口的可重用组件（通常是底层结构）
- JMX容器——协调对MBean的类属的访问（本地或远程的）
- JMX客户（通常是类属的）——可用于通过容器管理MBean

一个支持JMX的应用服务器（例如JBoss）担当JMX容器并允许MBean作为应用服务器起动过程的一部分来配置和初始化。可以使用应用服务器的管理控制台（作为JMX客户）来监视与管理MBean。

MBean可以打包为JMX服务，它不仅在支持JMX的应用服务器之间是可移植的而且对于运行中的系统它也是可配置的（热配置）。

Hibernate可以被打包并且被作为JMX MBean管理。Hibernate JMX服务允许Hibernate在应用服务器起动时初始化并且允许通过JMX客户进行控制（配置）。然而，JMX组件不能自动地

与容器管理的事务集成。因此，清单2.7（一个JBoss服务配置描述符）中列出的配置选项与通常Hibernate在管理环境中的设置看起来有些相似。

```
<server>
    <mbean
        code="net.sf.hibernate.jmx.HibernateService"
        name="jboss.jca:service=HibernateFactory, name=HibernateFactory">
        <depends>jboss.jca:service=RARDeployer</depends>
        <depends>jboss.jca:service=LocalTxCM,name=DataSource</depends>
        <attribute name="MapResources">
            auction/Item.hbm.xml, auction/Bid.hbm.xml
        </attribute>
        <attribute name="JndiName">
            java:/hibernate/HibernateFactory
        </attribute>
        <attribute name="Datasource">
            java:/comp/env/jdbc/AuctionDB
        </attribute>
        <attribute name="Dialect">
            net.sf.hibernate.dialect.PostgreSQLDialect
        </attribute>
        <attribute name="TransactionStrategy">
            net.sf.hibernate.transaction.JTATransactionFactory
        </attribute>
        <attribute name="TransactionManagerLookupStrategy">
            net.sf.hibernate.transaction.JBossTransactionManagerLookup
        </attribute>
        <attribute name="UserTransactionName">
            java:/UserTransaction
        </attribute>
    </mbean>
</server>
```

（清单2.7）

HibernateService依赖于其它两个JMX服务：service = RARDeployer和service =

LocalTxCM，name = DataSource，它们都位于jboss.jca服务域名里。

Hibernate MBean可以在包net.sf.hibernate.jmx里找到。很不幸，生命周期管理方法例如开始与停止JMX服务并不是JMX1.0规范的一部分。因此，HibernateService的start()与stop()方法是特定于JBoss应用服务器的。

注意 如果你对JMX的高级用法感兴趣，JBoss是一个很好的开源的起点：JBoss中所有的服务（甚至是EJB容器）都是作为MBean实现的并且都可以通过一个提供的控制台接口来管理。

我们推荐你在试图将Hibernate作为JMX服务运行之前，首先试验程序化地配置Hibernate（使用配置对象）。然而，许多特征（像Hibernate应用的热配置）一旦在Hibernate中可用了，可能也只能作为JMX使用。现在，Hibernate使用JMX的最大优点是自动起动；这意味着在你的应用代码中，你不再需要创建一个配置对象并构造一个会话工厂了，一旦HibernateService被配置并起动后你只需要通过JNDI简单地访问会话工厂就行了。

2．5 总结

本章在运行了一个简单的“Hello World”例子之后，我们在较高的层次上对Hibernate和它的体系结构进行了浏览。你也看到了如何在不同的环境中使用不同的技术配置Hibernate，甚至包括JMX。

接口Configuration（配置）和SessionFactory（会话工厂）是在管理与非管理两种环境中运行Hibernate应用的入口点。Hibernate还提供了另外的API，例如Transaction（事务）接口，来弥补这两种环境间的差异并允许你保持你的持续性代码的可移植性。

Hibernate可以被集成到几乎每一种Java环境中，可以是Servlet，Applet或者一个完全

管理的三层的客户/服务器应用。在Hibernate配置中最重要的元素是数据库资源（连接的配置），事务策略，当然还有基于XML的映射元数据。

Hibernate的配置接口被设计为包含尽可能多的使用场景同时还要易于理解。通常一个名为hibernate.cfg.xml的文件和一行代码就足够配置和运行Hibernate了。

如果没有持续类与XML映射文件的话这些东西都没有太多的使用价值。下一章我们将专注于编写和映射持续类。很快你就能够在一个包含重要的对象-关系映射的实际应用中存储与取出持续对象了。