第3章

驾驭Hibernate

好了，通过前两章的学习，我们已经打牢了基础，定义好对象/关系映射（object/relational mapping）后，又用它创建了相匹配的Java类和数据库表。但这如何应用呢？现在就介绍一下用Hibernate在Java程序代码中处理持久化（persistent）数据是多么的方便。

配置Hibernate

在我们继续学习Hibernate的使用之前，我们需要解决某些妨碍继续前进的问题。在上一章中，我们使用src目录下的hibernate.properties文件来配置Hibernate的JDBC连接。在这一章中，我们将使用Hibernate XML配置文件来配置JDBC连接、SQL方言等各种Hibernate设置。与hibernate.properties文件一样，我们也把这个XML配置文件放在src目录中。将例3-1的内容输入到一个名为hibernate.cfg.xml的文件中，并将其保存在src目录下，再删除原来的hibernate.properties文件。

例3-1：用XML配置Hibernate: hibernate.cfg.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <!-- SQL dialect -->
    <property name="dialect">org.hibernate.dialect.HSQLDialect</property> ❶

    <!-- Database connection settings --> ❷
    <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
    <property name="connection.url">jdbc:hsqldb:data/music</property>
    <property name="connection.username">sa</property>
    <property name="connection.password"></property>
    <property name="connection.shutdown">true</property>

    <!-- JDBC connection pool (use the built-in one) -->
```

```
        <property name="connection.pool_size">1</property> ❸

        <!-- Enable Hibernate's automatic session context management -->
        <property name="current_session_context_class">thread</property>

        <!-- Disable the second-level cache  --> ❹
        <property
         name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>

        <!-- disable batching so HSQLDB will propagate errors correctly. -->
        <property name="jdbc.batch_size">0</property> ❺

        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">true</property> ❻

        <!-- List all the mapping documents we're using --> ❼
        <mapping resource="com/oreilly/hh/data/Track.hbm.xml"/>
      </session-factory>
    </hibernate-configuration>
```

从示例中可以看到，hibernate.cfg.xml配置了SQL方言、JDBC参数、连接池（connection pool）以及缓存提供者（cache provider）。这个XML文档还引用了我们在上一章写的映射文档，这样就不需要我们再从Java源代码中引用这些映射文档了。稍后将详细介绍这一配置文件的细节：

❶ 就像第2章的hibernate.properties文件一样，我们在这一行是为使用HSQLDB而定义它的SQL方言。你可能已经注意到property元素的name属性是dialect，类似于.properties文件中的属性的名称（也就是hibernate.dialect）。当使用XML配置文件来配置Hibernate时，其实也是在向Hibernate传递了同样的属性。在XML配置文件中，可以省略属性名称的hibernate.前缀。这一区段（dialect）和下一区段（connection）的配置方法与我们在第2章hibernate.properties文件中的配置是一样的。

❷ 用于配置Hibernate内部的JDBC连接的属性（connection.driver_class、connection.url、connection.username、connection.password以及connection.shutdown）与例2-4的hibernate.properties中的属性集也一一对应。

❸ 将connection.pool_size属性设置为1。这意味着Hibernate将创建一个只保持一个连接的JDBC Connection（连接）池。数据库连接池对于需要达到一定规模的大型应用程序来说很重要，不过就本书的目的来说，我们可以放心地将Hibernate配置为使用只有一个JDBC连接的内建连接池。在连接池实现上，Hibernate支持很大的灵活性，可以非常容易地配置Hibernate来使用其他的连接池实现，例如Apache Commons DBCP和C3P0。

❹ 照目前情况，当Hibernate执行实际的持久化操作时，它就会警告我们还没有配置它的二级缓存系统。对于像这样的简单应用程序来说，我们根本不需要二级缓存；所以这一行配置就是要关闭二级缓存，将每个操作立即发送到数据库。

❺ 这里，我们是在关闭Hibernate的JDBC批处理功能。虽然这样做会稍微降低一点效率（对于

像HSQLDB这样的内存数据库的影响微乎其微），不过为了获取当前HSQLDB触发的错误报告，有必要这样做。当打开批处理模式时，如果批处理中有任何语句出了问题，从HSQLDB中得到的惟一异常将只是BatchUpdateException，告诉你批处理失败。这样错误报告对调试程序几乎没有什么用。HSQLDB的作者说这个问题将在下一个主要发行版本中得以修复；在此之前，当使用HSQLDB时，为了明白问题的来龙去脉，我们就先不使用批处理模式。

❻ how_sql属性是一个在开发和调试Hibernate程序时使用的属性。将show_sql设置为true，就告诉Hibernate要它打印输出对数据库操作时执行的每条语句。如果你不希望在控制台看到打印输出的SQL语句，可以将这个属性设置为false。

❼ 最后一部分列出了在我们的项目中使用的所有映射文档。注意，路径中包含的斜杠字符（“/”）是相对于src目录的。这一路径指明了.hbm.xml文件作为资源在类路径上的位置。通过在这里列出这些文件，我们就不用在处理映射类的build.xml构建目标中显式说明如何找到它们（稍后可以看到），也不需要像本书的旧版那样在每个例子源代码的main( )方法中加载这些映射文档。把所有映射文档集中在一起是个不错的做法。

除了src目录下的hibernate.cfg.xml文件，还需要修改build.xml以引用这个新的XML配置文件。修改codegen 和 schema 构建目标内的configuration元素，如例3-2中用粗体显示的几行所示。

例3-2：修改build.xml，以使用新的Hibernate XML配置文件

```
...

  <!-- Generate the java code for all mapping files in our source tree -->
  <target name="codegen" depends="prepare"
          description="Generate Java source from the O/R mapping files">
    <hibernatetool destdir="${source.root}">
      <configuration configurationfile="${source.root}/hibernate.cfg.xml"/>
      <hbm2java/>
    </hibernatetool>
  </target>

...

  <!-- Generate the schemas for all mapping files in our class tree -->
  <target name="schema" depends="prepare"
          description="Generate DB schema from the O/R mapping files">

    <hibernatetool destdir="${source.root}">
      <configuration configurationfile="${source.root}/hibernate.cfg.xml"/>
      <hbm2ddl drop="yes" />
    </hibernatetool>
  </target>

...
```

这两行告诉Hibernate Tools Ant构建任务，到哪里查找Hibernate XML配置文件，这样，Ant任务就可以在这个配置文件中找到它们需要的所有信息（包括正在处理的映射文档；回想

一下第2章的做法，我们不得不在configuration元素中显式构建了Ant的fileset元素，以匹配项目源代码树中的所有映射文件）。从现在起，本书的剩余部分将全部使用Hibernate XML配置文件，这样可以更方便地为Hibernate的相关工具传递各种需要的信息。

现在我们已经成功地配置好了Hibernate，下面我们就回到本章的主要任务：生成持久化对象。

创建持久化对象

我们先创建几个新的Track实例，再把它们持久化保存到数据库，这样就能看看对象到底是怎样转换成数据库表的行和列的。因为我们完全按照标准的Hibernate要求来组织映射文档和配置文件，所以配置Hibernate的会话工厂（session factory）也就变得相当容易了。

应该怎么做

这里的讨论假设你已经按照第2章的示例，创建好了数据库模式，并生成了Java代码。如果你还没有做这些，可以从本书的网站（注1）下载示例文件，直接跳转到ch03目录，使用命令ant prepare和ant codegen（注2），再接着执行ant schema，就可以自动取回这个示例所需要的Hibernate和HSQLDB库，并生成Java代码和数据库模式。（和其他示例一样，这些命令都应该在shell窗口中执行，而当前工作目录就是你的项目目录树的顶级目录，也就是包含Ant build.xml文件的地方。）

我们先从一个简单的示范用的CreateTest类开始，它包含了必要的导入（import）语句，以及一些辅助代码，用于设定Hibernate环境，再创建几个用XML映射文件来进行持久化存储的Track实例。源代码如例3-3所示，它位于src/com/oreilly/hh目录中。

例3-3：数据创建测试，CreateTest.java

```
package com.oreilly.hh;

import org.hibernate.*; ❶
import org.hibernate.cfg.Configuration;

import com.oreilly.hh.data.*;

import java.sql.Time;
import java.util.Date;

/**
 * Create sample data, letting Hibernate persist it for us.
 */
public class CreateTest {
```

注1：　http://www.oreilly.com/catalog/9780596517724/.

注2：　虽然codegen构建目标要依赖prepare构建目标，但第一次处理示例目录时，你需要先显式地运行prepare以创建正确的类路径结构，这样Ant以后才可以正常工作，如第2章的2.3节所述。

```
    public static void main(String args[]) throws Exception {
        // Create a configuration based on the XML file we've put
        // in the standard place.
        Configuration config = new Configuration(); ❷
        config.configure();

        // Get the session factory we can use for persistence
        SessionFactory sessionFactory = config.buildSessionFactory(); ❸

        // Ask for a session using the JDBC information we've configured
        Session session = sessionFactory.openSession(); ❹
        Transaction tx = null;
        try {
            // Create some data and persist it
            tx = session.beginTransaction(); ❺

            Track track = new Track("Russian Trance",
                                    "vol2/album610/track02.mp3",
                                    Time.valueOf("00:03:30"), new Date(),
                                    (short)0);
            session.save(track);

            track = new Track("Video Killed the Radio Star",
                              "vol2/album611/track12.mp3",
                              Time.valueOf("00:03:49"), new Date(),
                              (short)0);
            session.save(track);

             track = new Track("Gravity's Angel",
                              "vol2/album175/track03.mp3",
                               Time.valueOf("00:06:06"), new Date(),
                               (short)0);
            session.save(track);

            // We're done; make our changes permanent
            tx.commit(); ❻

        } catch (Exception e) {
            if (tx != null) {
                // Something went wrong; discard all partial changes
                tx.rollback();
            }
            throw new Exception("Transaction failed", e);
        } finally {
            // No matter what, close the session
            session.close();
        }

        // Clean up after ourselves
        sessionFactory.close(); ❼
    }
}
```

需要对CreateTest.java的第一部分做些解释：

❶ 我们导入一些有用的Hibernate类（包括Configuration），用它们来建立Hibernate运行环境。此外还需要导入Hibernate根据映射文档生成的所有数据类，这些都在data包中。数据对象中用Time和Date类来代表曲目的播放时间和创建时间戳（timestamp）。在CreateTest中实现的惟一方法是main( )方法，以支持从命令行的调用。

❷ 当运行这个类时，它首先创建一个Hibernate Configuration对象。由于我们没有告诉Hibernate什么其他信息，所以它默认在类路径的根上查找名为hibernate.cfg.xml的文件。Hibernate会找到我们前面创建的这个配置文件，通过这个配置文件来告诉Hibernate我们正在使用HSQLDB，以及如何找到数据库。这个例子的XML配置文件就包含了对Track对象的Hibernate Mapping XML文档的引用。调用config.configure( )就会自动加载Track类的映射文档。

❸ 为了创建和持久化曲目数据，这就是我们需要的所有配置，这样就为创建SessionFactory做好了准备。该对象的用途是为我们提供会话对象，它是同Hibernate进行交互的主要途径。SessionFactory是线程安全的，在整个应用程序中只需要创建它的一个实例（更准确地说，对于每一个需要提供持久化服务的数据库环境，只需要一个SessionFactory实例；因此大多数应用程序只需要一个这样的实例）。创建会话工厂是一个需要花费相当代价和耗时的操作，所以应该在整个应用程序中共享这个实例。在只包含一个类的应用程序中进行这样的操作显得有些繁琐，不过，参考文档提供了一些在更实现的场景中应该如何应用的好例子。

注意：牢固地理解这些对象的目的和生命周期，值！本书介绍的知识足以让你起步了；你应该继续花些时间阅读参考文档，深入理解各个例子。

---

❹ 这一步才到了真正执行持久化的时候，让SessionFactory打开一个会话，这会建立一个到数据库的连接，并提供一个上下文（context）环境，在这个上下文中我们可以创建、获取、处理以及删除持久化对象。只要保持会话为打开状态，就会维护一个到数据库的连接，与会话相关联的持久对象的变化都可以被跟踪；这样，当关闭会话时，这些变化就可以应用到数据库。从概念上说，你可以把会话认为是持久化对象和数据库之间的一个“大型事务”，它可以包含多个数据库级的事务。不过，和数据库事务一样，在应用程序运行期间长时间地打开Hibernate session（例如在等候用户输入时）。在应用程序中一个会话只用于一个特定的、边界有限的操作，例如生成用户界面或者根据用户提交的信息做出相应的变化。而随后的操作则使用一个新的会话。还要注意的是，会话对象本身不是线程安全的，所以不能在线程之间共享它们。每个线程需要从会话工厂类中获取它们自己的会话。

有必要深入介绍一下Hibernate中映射对象的生命周期，以及它与会话的关系，因为这一术语相当特殊，相关的概念也很重要。一个映射对象（例如我们的Track类的一个实例）会在与Hibernate相关的两个状态之间回来转换：瞬时状态（transient）和持久化状态（persistent）。处于瞬时状态的对象不与任何会话关联。当用new( )第一次创建一个Track

实例时，它就是瞬时状态的；除非告诉Hibernate持久化这个瞬时状态的对象，否则当应用程序结束时，这个对象就会永远消失。

将一个瞬时状态的映射对象传递给会话的save( )方法，就会持久化保存这个对象，这样，在Java VM结束以后，这个数据对象还能够存在，直到以后显式地删除它。如果你已经有了一个持久化对象，并对它调用会话的delete( )方法，这个对象就会再回到瞬时状态。虽然这个对象在应用程序中仍然作为一个实例而存在，但它不会持久保存起来，除非你改变了主意，要再保存它一次。另一方面，如果你还没有删除这个对象（所以它仍然处于持久化状态），当修改这个对象后，为了让对象的变化得以反映到数据库中，并不需要再显式地保存它一次。Hibernate会自动跟踪对任何持久化对象作出的修改，并在适当的时机将这些变化刷新写入到数据库中。当关闭会话时，任何延迟的变化都会被保存到数据库中。

注意：坚持一下，我们很快就回到例子的介绍！

当在已经关闭的会话中使用持久化对象，例如，当运行完一个查询，找到所有匹配某条件的实体（本章稍后的“检索持久化对象”一节将介绍如何做到这一点）以后，处理这些实体的状态就涉及一个重要但又微妙的要点。如前所述，保持会话打开的时间最好不要超过执行数据库操作的必要时间，所以在查询完成以后，就应该马上关闭会话。如果这时再处理已经加载的映射对象，会怎么样？嗯，当会话打开时，这些对象确实是持久化对象，但是它们现在不再与任何活动的会话相关联了（在这个例子中，是因为关闭了会话），所以它们就不再是持久化对象了。不过，这并不意味着它们代表的数据在数据库中也不存在了；相反，如果再次运行查询（假设这时没有人修改过数据），还是能够取回同样的一组对象。这只是意味着：数据对象的状态在虚拟机和数据库之间当前没有通信，它们处于脱管状态（detached）。完全有理由可以继续使用这种对象。如果以后需要修改这些对象，还想把这些修改持久保存，你可以打开一个新的会话，用它来保存修改过的对象。因为每个实体都有一个惟一的ID，在新的会话中，Hibernate没有问题地可以知道如何把处于瞬时状态的对象链接到正确的持久化对象。

当然，对脱管对象信息的修改都要由具体的数据库环境作为支持，所以你需要考虑应用程序级的数据完整性约束。可能需要设计某种高级的锁定或版本化协议来支持脱管对象的管理。虽然Hibernate为这一任务也提供了一定帮助，但是设计和实现细节还得由你负责。参考手册强烈推荐使用一个version字段，而且也有多种办法可供选用。

❺ 有了这些概念和术语，这个例子的其他部分就很容易理解了。我们用打开的会话建立了一个数据库事务，在这个事务内部，又创建了一些包含示例数据的Track实例，并在会话

中进行保存，这样就把它们由瞬时状态的实例转变为持久化的实体。

❻ 最后，我们提交事务，自动地（作为一个单独的、不可分割的单元）将所有数据库修改持久化。环绕着所有这些代码周围的try/catch/finally块演示了在进行事务处理时一种重要而且有用的习惯用法。如果操作期间发生了任何错误，catch块就会回滚（roll back）事务，再抛出异常。在finally部分中会关闭会话，以确保无论我们是成功提交事务后沿着“愉快的小路”正常退出，还是因为导致回滚的异常而退出，最终都可以关闭会话。

❼ 在方法最后，我们也关闭了会话工厂本身。这是应用程序中“优雅地关闭”（graceful shutdown）部分应该进行的操作。在Web应用环境中，这应该是一定的生命周期事件处理器。在这个简单的例子中，当main( )方法返回时，应用程序就正在结束。

现在一切就绪，通知Ant如何编译和运行这个测试程序就更简单了。将例3-4所示的构建目标添加到build.xml末尾的</project>关闭标签之前。

例3-4：用于编译所有Java源代码，并调用数据创建测试的Ant构建目标

```
<!-- Compile the java source of the project -->
<target name="compile" depends="prepare" ❶
        description="Compiles all Java classes">
  <javac srcdir="${source.root}"
         destdir="${class.root}"
         debug="on"
         optimize="off"
         deprecation="on">
    <classpath refid="project.class.path"/>
  </javac>
</target>

<target name="ctest" description="Creates and persists some sample data"
        depends="compile"> ❷
  <java classname="com.oreilly.hh.CreateTest" fork="true">
    <classpath refid="project.class.path"/>
  </java>
</target>
```

❶ 命名得体的编译构建目标使用内建的javac构建任务，将src目录中的所有Java源文件编译到classes目录中。幸好，这个构建任务也支持我们已经建立的项目类路径，所以编译器能够找到我们正在使用的所有依赖库。构建目标定义中的depends=prepare属性是告诉Ant在运行编译构建目标以前，必须先运行prepare。Ant负责管理依赖关系，当构建具有依赖关系的多个目标时，能够以正确的顺序执行，每个依赖只执行一次，即便多个构建目标都引用了同一个依赖。

如果你习惯使用shell脚本来编译很多Java源代码，那么可能会对这么快的编译速度感到吃惊。Ant调用的是它自己使用的虚拟机内部的Java编译器，所以没有针对每个编译的处理启动延迟。

❷ ctest构建目标使用编译来确保已经构建好了所有类文件，接着再创建一个新的Java虚拟机来运行我们的CreateTest类。

好了，我们可以创建一些数据了！例3-5显示了调用新的ctest构建目标的结果。ctest要依赖于compile构建目标，这样就能确保在使用之前CreateTest类会先编译好。ctest本身的输出结果会显示Hibernate所发出的日志信息（建立环境和映射数据以及数据库连接的关闭）。

例3-5：调用CreateTest类

```
% ant ctest
prepare:

compile:
    [javac] Compiling 2 source files to /Users/jim/svn/oreilly/hib_dev_2e/
current/examples/ch03/classes

ctest:
     [java] 00:21:45,833  INFO Environment:514 - Hibernate 3.2.5
     [java] 00:21:45,852  INFO Environment:547 - hibernate.properties not found
     [java] 00:21:45,864  INFO Environment:681 - Bytecode provider name : cglib
     [java] 00:21:45,875  INFO Environment:598 - using JDK 1.4 java.sql.Timestam
p handling
     [java] 00:21:46,032  INFO Configuration:1426 - configuring from resource: /
hibernate.cfg.xml
     [java] 00:21:46,034  INFO Configuration:1403 - Configuration resource: /hib
ernate.cfg.xml
     [java] 00:21:46,302  INFO Configuration:553 - Reading mappings from resourc
e : com/oreilly/hh/data/Track.hbm.xml
     [java] 00:21:46,605  INFO HbmBinder:300 - Mapping class: com.oreilly.hh.dat
a.Track -> TRACK
     [java] 00:21:46,678  INFO Configuration:1541 - Configured SessionFactory: n
ull
     [java] 00:21:46,860  INFO DriverManagerConnectionProvider:41 - Using Hibern
ate built-in connection pool (not for production use!)
     [java] 00:21:46,862  INFO DriverManagerConnectionProvider:42 - Hibernate co
nnection pool size: 1
     [java] 00:21:46,864  INFO DriverManagerConnectionProvider:45 - autocommit m
ode: false
     [java] 00:21:46,879  INFO DriverManagerConnectionProvider:80 - using driver
: org.hsqldb.jdbcDriver at URL: jdbc:hsqldb:data/music
     [java] 00:21:46,891  INFO DriverManagerConnectionProvider:86 - connection p
roperties: {user=sa, password=****, shutdown=true}
     [java] 00:21:47,533  INFO SettingsFactory:89 - RDBMS: HSQL Database Engine,
 version: 1.8.0
     [java] 00:21:47,538  INFO SettingsFactory:90 - JDBC driver: HSQL Database E
ngine Driver, version: 1.8.0
     [java] 00:21:47,613  INFO Dialect:152 - Using dialect: org.hibernate.dialec
t.HSQLDialect
     [java] 00:21:47,638  INFO TransactionFactoryFactory:31 - Using default tran
saction strategy (direct JDBC transactions)
     [java] 00:21:47,646  INFO TransactionManagerLookupFactory:33 - No Transacti
onManagerLookup configured (in JTA environment, use of read-write or transaction
al second-level cache is not recommended)
```

```
     [java] 00:21:47,649  INFO SettingsFactory:143 - Automatic flush during befo
reCompletion(): disabled
     [java] 00:21:47,650  INFO SettingsFactory:147 - Automatic session close at
end of transaction: disabled
     [java] 00:21:47,657  INFO SettingsFactory:154 - JDBC batch size: 15
     [java] 00:21:47,659  INFO SettingsFactory:157 - JDBC batch updates for vers
ioned data: disabled
     [java] 00:21:47,664  INFO SettingsFactory:162 - Scrollable result sets: ena
bled
     [java] 00:21:47,666  INFO SettingsFactory:170 - JDBC3 getGeneratedKeys(): d
isabled
     [java] 00:21:47,668  INFO SettingsFactory:178 - Connection release mode: au
to
     [java] 00:21:47,671  INFO SettingsFactory:205 - Default batch fetch size: 1
     [java] 00:21:47,678  INFO SettingsFactory:209 - Generate SQL with comments:
 disabled
     [java] 00:21:47,680  INFO SettingsFactory:213 - Order SQL updates by primar
y key: disabled
     [java] 00:21:47,681  INFO SettingsFactory:217 - Order SQL inserts for batch
ing: disabled
     [java] 00:21:47,684  INFO SettingsFactory:386 - Query translator: org.hiber
nate.hql.ast.ASTQueryTranslatorFactory
     [java] 00:21:47,690  INFO ASTQueryTranslatorFactory:24 - Using ASTQueryTran
slatorFactory
     [java] 00:21:47,694  INFO SettingsFactory:225 - Query language substitution
s: {}
     [java] 00:21:47,695  INFO SettingsFactory:230 - JPA-QL strict compliance: d
isabled
     [java] 00:21:47,702  INFO SettingsFactory:235 - Second-level cache: enabled
     [java] 00:21:47,704  INFO SettingsFactory:239 - Query cache: disabled
     [java] 00:21:47,706  INFO SettingsFactory:373 - Cache provider: org.hiberna
te.cache.NoCacheProvider
     [java] 00:21:47,707  INFO SettingsFactory:254 - Optimize cache for minimal
puts: disabled
     [java] 00:21:47,709  INFO SettingsFactory:263 - Structured second-level cac
he entries: disabled
     [java] 00:21:47,724  INFO SettingsFactory:283 - Echoing all SQL to stdout
     [java] 00:21:47,731  INFO SettingsFactory:290 - Statistics: disabled
     [java] 00:21:47,732  INFO SettingsFactory:294 - Deleted entity synthetic id
entifier rollback: disabled
     [java] 00:21:47,734  INFO SettingsFactory:309 - Default entity-mode: pojo
     [java] 00:21:47,735  INFO SettingsFactory:313 - Named query checking : enab
led
     [java] 00:21:47,838  INFO SessionFactoryImpl:161 - building session factory
     [java] 00:21:48,464  INFO SessionFactoryObjectFactory:82 - Not binding fact
ory to JNDI, no JNDI name configured
     [java] Hibernate: insert into TRACK (TRACK_ID, title, filePath, playTime, a
dded, volume) values (null, ?, ?, ?, ?, ?)
     [java] Hibernate: call identity()
     [java] Hibernate: insert into TRACK (TRACK_ID, title, filePath, playTime, a
dded, volume) values (null, ?, ?, ?, ?, ?)
     [java] Hibernate: call identity()
     [java] Hibernate: insert into TRACK (TRACK_ID, title, filePath, playTime, a
dded, volume) values (null, ?, ?, ?, ?, ?)
     [java] Hibernate: call identity()
```

```
     [java] 00:21:49,365  INFO SessionFactoryImpl:769 - closing
     [java] 00:21:49,369  INFO DriverManagerConnectionProvider:147 - cleaning up
 connection pool: jdbc:hsqldb:data/music

BUILD SUCCESSFUL
Total time: 2 seconds
```

发生了什么事

如果你查看Hibernate打印输出的所有消息（因为我们打开了日志的“ info ”级别的输出标志），你可以看到我们的测试类会启动Hibernate，加载Track类的映射信息，打开一个持久会话（persistence session）以连接相关的HSQLDB数据库，再创建一些实例，并用会话将它们持久化保存到TRACK数据库表中。接着，再关闭这个会话和数据库连接，以确保数据正确地完成存储。

运行完这个测试以后，你可以用ant db看一看数据库的内容。现在，你应该可以在TRACK表中看到三条记录，如图3-1所示（在窗口顶部的文本框中输入查询语句，点击“ Execute ”按钮。也可以选择菜单栏中的“ Command ”→“ Select ”，会得到一个SQL命令的框架和语法说明文档）。

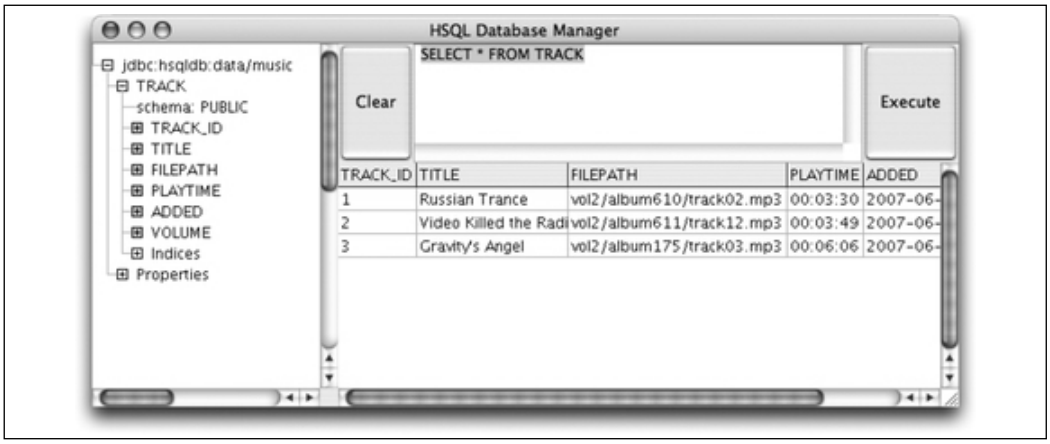


图3-1：持久保存到TRACK表中的测试数据

现在，我们停下来一会儿，反思一个事实——我们没有编写任何连接数据库或执行SQL命令的代码。再回想上一节，我们甚至也不必亲自创建数据库表和封装数据的Track对象。但是，图3-1中查询输出显示的那些整整齐齐的数据表明，我们简短的测试程序确实已经创建了Java对象，并正确地进行了持久化处理。希望你可以因此而认同，作为一种持久化服务，Hibernate真的功能强大、使用方便。作为一种免费的、轻型的工具，Hibernate确实为我们完成了很多工作，这一切又是那么的快捷而简单！

如果你以前直接用过JDBC，尤其是当你还不太熟悉它时，你可能习惯使用数据库驱动程序的“auto-commit”（自动提交）模式，而不是使用数据库事务处理（transaction）。Hibernate同样认为这是构造应用程序的一种错误方法，“自动提交”模式惟一有意义的使用场合就是供人使用的数据库控制台环境。所以，在Hibernate应用中，持久化操作总是需要使用事务处理。

如第1章所述，你可以在data目录下的music.script文件中直接查看用于创建数据的SQL语句，如例3-6所示。

例3-6：查看原始的数据库脚本文件

```
% cat data/music.script
CREATE SCHEMA PUBLIC AUTHORIZATION DBA
CREATE MEMORY TABLE TRACK(TRACK_ID INTEGER GENERATED BY DEFAULT AS IDENTITY(STAR
T WITH 1) NOT NULL PRIMARY KEY,TITLE VARCHAR(255) NOT NULL,FILEPATH VARCHAR(255)
 NOT NULL,PLAYTIME TIME,ADDED DATE,VOLUME SMALLINT NOT NULL)
ALTER TABLE TRACK ALTER COLUMN TRACK_ID RESTART WITH 4
CREATE USER SA PASSWORD ""
GRANT DBA TO SA
SET WRITE_DELAY 10
SET SCHEMA PUBLIC
INSERT INTO TRACK VALUES(1,'Russian Trance','vol2/album610/track02.mp3','00:03:3
0','2007-06-17',0)
INSERT INTO TRACK VALUES(2,'Video Killed the Radio Star','vol2/album611/track12.
mp3','00:03:49','2007-06-17',0)
INSERT INTO TRACK VALUES(3,'Gravity''s Angel','vol2/album175/track03.mp3','00:06
:06','2007-06-17',0)
```

最后三条语句是我们的TRACK表的数据行。脚本的最前面包含了当创建新的数据库时，默认使用的模式和用户名。（当然，在实际应用环境中，你可能需要修改这些身份验证信息，除非数据库只供内存访问（in-memory access）。）

注意：想多学点HSQLDB？我们支持你！

其他

对象和其他对象之间的关系呢？对象集合（collection）呢？没错，有些情况会让持久化处理更具挑战性（如果做得好的话，就相当有价值）。Hibernate能妥善处理这种关联性。事实上，我们不需要为此花费什么力气。在第4章将会讨论这一点。就目前而言，让我们先看看如何将先前会话中持久保存的对象取出来。

检索持久化对象

现在，是该来个180度大转弯，看看如何从数据库加载数据到Java对象中。

使用Hibernate查询语言（Hibernate Query Language，HQL），就能以面向对象的方法来检索

映射到数据库表的内容。这些数据可以是以前的会话中保存的持久化对象，也可以是完全来自于应用程序代码以外的数据。

应该怎么做

例3-7演示了一个程序，对我们刚才创建的测试数据进行简单的查询。整体的结构看起来应该非常熟悉，因为所有的Hibernate设置都和上一个程序相同。

例3-7：数据检索测试QueryTest.java

```
package com.oreilly.hh;

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

import com.oreilly.hh.data.*;

import java.sql.Time;
import java.util.*;

/**
 * Retrieve data as objects
 */
public class QueryTest {

    /**
     * Retrieve any tracks that fit in the specified amount of time.
     *
     * @param length the maximum playing time for tracks to be returned.
     * @param session the Hibernate session that can retrieve data.
     * @return a list of {@link Track}s meeting the length restriction.
     */
    public static List tracksNoLongerThan(Time length, Session session) { ❶
        Query query = session.createQuery("from Track as track " +
                                          "where track.playTime <= ?");
        query.setParameter(0, length, Hibernate.TIME);
        return query.list();
    }

    /**
     * Look up and print some tracks when invoked from the command line.
     */
    public static void main(String args[]) throws Exception {
        // Create a configuration based on the properties file we've put
        // in the standard place.
        Configuration config = new Configuration();
        config.configure();

        // Get the session factory we can use for persistence
        SessionFactory sessionFactory = config.buildSessionFactory();

        // Ask for a session using the JDBC information we've configured
        Session session = sessionFactory.openSession();
        try { ❷
            // Print the tracks that will fit in five minutes
            List tracks = tracksNoLongerThan(Time.valueOf("00:05:00"),
                                             session);
```

```
            for (ListIterator iter = tracks.listIterator() ;
                 iter.hasNext() ; ) {
                Track aTrack = (Track)iter.next();
                System.out.println("Track: \"" + aTrack.getTitle() +
                                   "\", " + aTrack.getPlayTime());
            }
        } finally {
            // No matter what, close the session
            session.close();
        }

        // Clean up after ourselves
        sessionFactory.close();
    }
}
```

同样，如例3-8所示，在build.xml的末尾（在project的关闭标签以前）添加一个构建目标，以运行这个测试。

例3-8：调用查询测试的Ant构建目标

```
<target name="qtest" description="Run a simple Hibernate query"
         depends="compile">
  <java classname="com.oreilly.hh.QueryTest" fork="true">
    <classpath refid="project.class.path"/>
  </java>
</target>
```

准备好以后，只要输入ant qtest，就可以检索出数据并显示出来，其结果如例3-9所示。为了节省输出结果占据的空间，我们编辑log4j.properties文件，将所有“info”级别的信息输出都关掉，因为这些信息和前一个例子没有差别。你可以修改一下这一行：

```
log4j.logger.org.hibernate=info
```

将info替换成warn：

```
log4j.logger.org.hibernate=warn
```

例3-9：运行查询测试

```
% ant qtest
Buildfile: build.xml

prepare:
     [copy] Copying 1 file to /Users/jim/svn/oreilly/hib_dev_2e/current
/examples/ch03/classes

compile:
    [javac] Compiling 1 source file to /Users/jim/svn/oreilly/hib_dev_2e
/current/examples/ch03/classes

qtest:
     [java] Hibernate: select track0_.TRACK_ID as TRACK1_0_, track0_.title as ti
tle0_, track0_.filePath as filePath0_, track0_.playTime as playTime0_, track0_.a
dded as added0_, track0_.volume as volume0_ from TRACK track0_ where track0_.pla
yTime<=?
```

```
     [java] Track: "Russian Trance", 00:03:30
     [java] Track: "Video Killed the Radio Star", 00:03:49

BUILD SUCCESSFUL
Total time: 2 seconds
```

发生了什么事

❶ 我们先是定义了一个工具方法：tracksNoLongerThan( )，由它执行真正的Hibernate查询，取回播放时间小于或等于参数指定的值的任何曲目。注意HQL（Hibernate根据SQL独创的查询语言）支持参数占位符，非常像JDBC中的PreparedStatement。而且，与之类似的是，使用参数占位符更适合于通过字符串处理将所有查询集中在一起（尤其是这样可以免受SQL注入的攻击）。不过，你将看到，Hibernate提供了在Java中更好的使用查询的方法。

查询本身看起来有些古怪。它以from作为开始，而不是你可能期待的像select something之类的语句。虽然你确实可以使用与标准SQL更加类似的格式，而且当需要在查询中从一个对象提取出单独的属性时，也只能这么做；如果你想获取整个对象，就可以使用这种更简洁的语法。

还要注意的是，查询是按照映射的Java对象和属性来表达的，而不是数据表和列。在这个例子中这一区别还不明显，因为对象和数据表的名称都相同，属性和列的名称也都相同，但查询确实是按照对象和属性来表达的。保持二者的名称一致是一种相当自然的选择，如果使用Hibernate来生成数据库模式和数据对象，结果也总是这样，除非你明确地告诉Hibernate使用不同的列名称。

当你使用的是以前就有的数据库和对象时，就应该特别留意HQL查询引用的是对象属性，而不是数据库表的列。

此外，就像在SQL中一样，可以为数据库表和列起其他的别名。在HQL中，也可以为类起别名，以方便选择它们的属性或增加约束条件。当然，在这个简单的例子中不会看到这种用法，但是如果你深入研究附录E中的内容，肯定会遇到这种用法。

❷ 这个程序的其他部分看起来可能和上一个例子类似。这里我们简化了try块的处理，因为没有改变任何数据，所以就不需要显式地访问事务。我们仍旧使用try块，这样就能够有一个finally子句，以清晰地关闭会话。代码体本身很简单，调用我们的查询方法以请求播放时间小于或等于5分钟的所有曲目，接着再遍历结果中的Track对象，分别打印它们的标题和播放时间。

既然我们已经关掉了Hibernate内部“ info ”级别的日志输出，那么我们在hibernate.cfg.xml中配置的SQL调试输出就更容易定位了。输出中“ qtest: ”部分的第1行并不是我们自己在QueryTest.java中编写的，我们看到的SQL语句是Hibernate生成的，以实现我们请求的HQL

查询。这些内幕信息很有趣吧！如果你对这些信息也不感兴趣，则可以将show_sql属性设置为false，就不会看到它们了。

其他

如果你已经对数据创建脚本生成的数据进行了修改，现在又想清空数据库，以一种“干净的状态”（clean slate）来进行测试，那么你需要做的就是再次运行ant schema命令。这样会将原有的TRACK表完全删除，并重新创建新的TRACK表，使其处于最原始的空状态。除非你真的需要这样，否则不要轻易这样做！

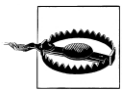
如果你想选择性地删除一些数据，则或者通过HSQLDB UI (ant db)里的SQL命令；或者先进行一定的查询，检索回你想删除的对象。在取得持久化对象的引用以后，可以将该对象的引用传递给Session的delete()方法，这样就可以将它从数据库中删除了：

```
session.delete(aTrack);
```

直到aTrack变量超出其作用域范围（scope）或者重新赋值以前，你的程序还至少有一个指向这个被删除对象的引用。因此，就概念上而言，理解delete()方法最简单的方式就是将其视为把一个持久化对象（persistent object）转变为一个瞬时对象（transient object）。

删除对象的另外一种方法就编写一条可以匹配多个对象的HQL删除查询语句。这样可以一次性删除多个持久化对象，而不管这些对象是否在内存中，还不用自己写循环。除了使用ant schema命令，改用Java方法，以较“温和”的手法来清除掉所有曲目时，就可以像这样写：

```
Query query = session.createQuery("delete from Track");
query.executeUpdate();
```



不要忘了，无论采用哪种方法，都得将数据处理的代码放在Hibernate事务处理以内，如果想让修改的内容“固定”下来，就得提交（commit）该事务。

建立查询的更好方法

如本章前面所述，HQL让你不必使用JDBC风格的查询占位符，就能方便地将参数整合到查询中。可以使用命名参数（named parameter）和命名查询（named query）来让程序更易于阅读和维护。

为何在意

命名参数之所以可以让代码更容易理解，是因为不管在查询本身内部，还是在建立查询的Java代码内部，都清晰地表达了参数的意图。这种自描述性（self-documenting）的本质就是命名参数的价值所在，同时也减少了引发错误的潜在可能，因为使用命名参数时，你不用计算SQL语句中逗号和问号的个数。可以在一个单独的查询中多次使用相同的参数，这样

在一定程度上也可以提高程序的性能。

命名查询可以将查询完全从Java代码中分离出来。将查询语句和Java源代码分离开，会使其更易于阅读和编辑，因为查询语句不再是原来一大堆跨越多行的Java字符串序列，且交织着大量的问号、反斜线以及其他Java标点符号。第一次输入这样的语句是相当麻烦的，但是如果你需要对这种嵌入在程序中的查询做重要的修改时，就得四处移动问号和加号，尽可能让语句行再次在适当的位置断开。

应该怎么做

对于Hibernate而言，这些能力的关键就是Query接口。我们在例3-7中就已经使用过这个接口，因为从Hibernate 3开始，Query接口是惟一Hibernate不反对使用（nondeprecated ）的执行查询的接口。所以，和本节介绍的其他功能相比，现在更容易了。

我们先修改查询语句，使用命名参数，如例3-10所示。（对于这种只有单一参数的查询语句而言，这并不是什么难事，但宝贵的是从现在起就养成正确的习惯。当你开始为实际的项目打造一大堆难以看清的查询语句时，你会很感激这个习惯的！）

例3-10：修改查询语句，改用命名参数

```
public static List tracksNoLongerThan(Time length, Session session) {
    Query query = session.createQuery("from Track as track " +
                                      "where track.playTime <= :length");
    query.setTime("length", length);
    return query.list();
}
```

在查询语句内部，命名参数通过在它们的名称前面加一个冒号“ : ”作为标识。此处，我们将“ ? ”号替换为“ :length ”。会话对象提供了一个createQuery()方法，可以返回一个Query接口的实现，以供我们使用。为了设置命名参数的值，Query提供了一整套类型安全的方法。这里，我们传递的是一个Time类型的值，所以我们使用了setTime( )方法。即使在如此简单的情况下，和原来的查询语句相比，使用命名参数以后的语法更为自然和方便阅读。如果我们原来传递参数使用的是值和类型的匿名数组（anonymous array）（必需传递多个参数），这样的改进就显得更重要了。此外，我们又多加了一层编译时（compile-time）的类型检查，这总是很受开发人员欢迎的调整。

运行这一版本的程序产生的结果和原来的程序完全相同。

那么，我们怎么将查询语句文本放到Java源代码以外呢？同样，这个查询太简单了，以至于这样做的需要不像在实际项目中那样令人印象深刻。但是，这是处理查询语句的最佳方法，所以开始练习吧！就像你预料的那样，我们存放查询语句的地方就是映射文档内部。例3-11显示了映射文档中的查询语句。我们必须使用有点笨重的CDATA结构，因为查询语句中可

能会包含一些破坏XML解析器处理的字符（例如，像“ < ”这样的字符）。

例3-11：映射文档中的查询语句

```
<query name="com.oreilly.hh.tracksNoLongerThan">
  <![CDATA[
      from Track as track
      where track.playTime <= :length
    ]]>
</query>
```

将这些内容放在Track.hbm.xml中类定义的闭合标签（</class>）之后（就在</hibernate-mapping>那一行的前面）。然后，对QueryTest.java做最后一次修改，如例3-12所示。同样地，程序产生的结果和最初版本完全相同，只是现在组织得更好。如果我们需要使用更复杂的查询，有这些基础已经相当好了。

例3-12：查询方法的最终版本

```
public static List tracksNoLongerThan(Time length, Session session) {
    Query query = session.getNamedQuery(
                        "com.oreilly.hh.tracksNoLongerThan");
    query.setTime("length", length);
    return query.list();
}
```

除了这里探讨的内容以外，Query接口还有很多其他有用的功能。可以使用接口控制要取回多少条记录（并可以指定特定的记录行）。如果JDBC驱动程序支持可滚动的（scrollable）ResultSet，通过Query接口也可以使用这一功能。相关更多的细节，可以查阅JavaDoc和Hibernate的参考手册。

其他

完全不使用类SQL语言？或者深入HQL，探索更为复杂的查询？这两种方法都是本书后面将要介绍的内容。

第8章会讨论条件查询（criteria query）。条件查询是一种很有趣的机制，可以让你用普通的Java API表达出想要对实体施加的限制条件。这样就可以构建Java对象来表示你想要找到的数据，对于非数据库专家来说，这种方法更容易理解。这种方法还可以让你利用IDE的代码完成（code completion）功能作为一种辅助编辑方法，甚至可以提供编译时的语法检查。此外，Hibernate还支持一种“按照例子来查询”（query by example）的方法，就是要找什么对象，先提供与之类似的对象例子。这些对实现应用程序的查询搜索接口都很有帮助。

想多看点HQL的SQL老手可以跳到第9章，那一章将讨论HQL的更多其他能力和独特功能。

就目前而言，接下来我们要继续探索对象/关系映射中如何处理对象之间的相互关系，这在任何稍复杂的实际应用中都会遇到。