 动机 曾经使用许多方法在数据库和目标代码之间传输数据。从手动编码的SQL到JDO，然后再到EJB，我从未找到一种特别喜欢的方法。自从采用测试驱动开发（TDD）作为指导原则以来，这种不满情绪变得更加强烈。 单元测试的障碍应尽可能少。在关系数据库中，障碍的范围从外部依赖（数据库在运行吗？）到保持关系模型和对象模型同步的速度。由于这些原因，保持数据库访问代码与核心对象模型分离且无需涉及真实数据库而进行尽可能多的测试是很重要的。 通常这会导致我们进入下面两种模式之一。第一种是具体化所有访问域对象的数据以及数据与单独类或接口之间的关系。这就是典型的能够检索、编辑、删除和添加域实体的数据存储对象。这在单元测试中是最容易模拟出来的，但趋向于把域模型对象作为不带有任何关系行为的纯数据对象。直接从父对象访问子记录是最理想的，而不是将父对象处理为第三方类来决定子记录。 其他方法已经使访问接口的域对象进入数据映射层（一种la Martin Fowler的数据映象模式）。这具有推动域模型中的对象关系的优点，在域模型中，对象关系型接口只需表达一次即可。使用域模型的类不支持持久性机制，因为它本身内在化到域模型中。这使代码集中在设法解决的业务问题，而很少关注对象关系型映射机制。 我的当前项目涉及到处理大量的棒球统计数据，并使用这些数据进行模拟。因为数据已经在关系数据库中，所以对于我来说，有机会开发Hibernate对象关系型映射系

统。我曾对Hibernate有很深刻的印象，但我遇到的一个问题是，在使用Hibernate进行单元测试的数据映射时，设法插入一个间接层。该附加层非常脆弱，编写起来感到非常困难。实际部署版本简单地通过了特定于Hibernate的实现。更坏的情况是，模拟版本比真正的“产品级”版本更复杂，只因为模拟版本里没有基本对象存储器和带有Hibernate的映射。 我也使用很多复杂的Hibernate查询，想要对应用程序的重要部分进行单元测试。然而，对活动的数据库进行测试不是好主意，因为这几乎总是产生维护问题。另外，由于测试最好互相独立，在测试上下文数据中使用相同的主键意味着必须在每次测试前创建代码来清理数据库，当涉及到大量关系时就成为一个实际问题。 通过使用HSQLDB和Hibernate强大的模式生成工具，能够对应用程序映射层进行单元测试，并在对象查询中找到不计其数的bug，这在以前手工测试时是做不到的。利用下面的技术概述，可以在开发过程中对整个应用程序进行测试，并且在测试有效区域内没有损害。 设置HSQLDB 以前使用HSQLDB 1.7.3.0 版。为了使用数据库的内存版本，需要激活org.hsqldb.JDBCDriver的静态加载程序。当获得JDBC连接时，就可以使用JDBC url例如jdbc:hspldb:mem:yourdb，这里’yourdb’就是想要使用的内存数据库的名称。 因为使用Hibernate (3.0 beta 4)，所以我几乎无需接触实际活动的JDBC对象。相反,我可以让Hibernate完成很多繁重的任务，包括从Hibernate映射文件中自动创建数据库模式。因为Hibernate创建自身专有的连接池，所以它会基于TestSchema类中的配置代码自动加载HSQLDB JDBC驱动程序。下面就是该类的静态的初始化程序。 public class

TestSchema { static { Configuration config = new Configuration(). setProperty("hibernate.dialect", "org.hibernate.dialect.HSQLDialect"). setProperty("hibernate.connection.driver_class", "org.hsqldb.jdbcDriver"). setProperty("hibernate.connection.url", "jdbc:hsqldb:mem:baseball"). setProperty("hibernate.connection.username", "sa"). setProperty("hibernate.connection.passWord", ""). setProperty("hibernate.connection.pool_size", "1"). setProperty("hibernate.connection.autocommit", "true"). setProperty("hibernate.cache.provider_class", "org.hibernate.cache.HashtableCacheProvider"). setProperty("hibernate.hbm2ddl.auto", "create-0drop"). setProperty("hibernate.show_sql", "true"). addClass(Player.class). addClass(BattingStint.class). addClass(FieldingStint.class). addClass(PitchingStint.class). HibernateUtil.setSessionFactory(config.buildSessionFactory()). }

Hibernate提供了许多不同的方式来配置该框架，包括程序方面的配置。上述代码设置了连接池。注意，使用HSQLDB的内存数据库需要用户名’sa’。还样要确保指定一个空格作为口令。为了启动Hibernate的自动模式生成功能，需设置hibernate.hbm2ddl.auto属性为’creat-0drop’。 实际测试 我的项目是处理将大量的棒球数据，所以我添加了四个进行映射的类（Player、PintchingStint、,BattingSint和FieldStint）。最后创建Hibernate的会话工厂，并将其插入HibernateUtil类，该类只为Hibernate会话的整个应用程序提供一个访问方法

。HibernateUtil的代码如下： import org.hibernate.*. import org.hibernate.cfg.Configuration. public class HibernateUtil { private static SessionFactory factory. public static synchronized Session getSession() { if (factory == null) { factory = new Configuration().configure().buildSessionFactory(). } return factory.openSession(). } public static void setSessionFactory(SessionFactory factory) { HibernateUtil.factory = factory. } } 因为所有代码（经过单元测试的产品级代码）都是从HibernateUtil获取Hibernate会话，所以能在同一个位置对其进行配置。为了对代码的第一位进行单元测试而访问TestSchema类将会激活静态初始化程序，该程序将安装Hibernate并且将测试SessionFactory插入到HibernateUtil中。对于产品级代码，可以使用标准hibernate.cfg.XML配置机制来初始化 SessionFactory。 那么单元测试中的外部特征是什么？下面的测试代码片段是用来检查逻辑的，决定运动员在棒球联盟比赛中是哪个位置的人选： public void testGetEligiblePositions() throws Exception { Player player = new Player("playerId"). TestSchema.addPlayer(player). FieldingStint stint1 = new FieldingStint("playerId", 2004, "SEA", Position.CATCHER). stint1.setGames(20). TestSchema.addFieldingStint(stint1). Set positions = player.getEligiblePositions(2004). assertEquals(1, positions.size()). assertTrue(positions.contains(Position.CATCHER)). } 第一次创建新Player实例并通过addPlayer()方法添加到TestSchema中。必须首先完成此步骤，因为FidldStint类和Player类之间有外键关系。如果不首先添加该实例，在设法添加FieldingStint时将

会出现外键约束违例。一旦测试上下文就位，就可以测试getEligiblePositions()方法来检索校正数据。下面是在TsetSchema中addPlayer()方法的代码。您将注意到使用Hibernate而不是bare-metal JDBC代码： public static void addPlayer(Player player) { if (player.getPlayerId() == null) { throw new IllegalArgumentException("No primary key specified"). } Session session = HibernateUtil.getSession(). Transaction transaction = session.beginTransaction(). try { session.save(player, player.getPlayerId()). transaction.commit(). } finally { session.close(). } } 在单元测试中最重要的就是要保持测试实例是独立的。因为该方法仍然涉及数据库，所以需要一种方法在每个测试实例之前清理数据库。在我的数据库架构中有四个表，所以我在TestSchemaz上编写了reset()方法，该方法从使用JDBC的表中删除所有行。注意，因为HSQLDB能识别外键，删除表的顺序是很重要的，下面是代码： public static void reset() throws SchemaException { Session session = HibernateUtil.getSession(). try { Connection connection = session.connection(). try { Statement statement = connection.createStatement(). try { statement.executeUpdate("0delete from Batting"). statement.executeUpdate("0delete from Fielding"). statement.executeUpdate("0delete from Pitching"). statement.executeUpdate("0delete from Player"). connection.commit(). } finally { statement.close(). } } catch (HibernateException e) { connection.rollback(). throw new SchemaException(e). } catch (SQLException e) {

connection.rollback(). throw new SchemaException(e). } } catch (SQLException e) { throw new SchemaException(e). } finally { session.close(). } } 当确定在Hibernate 3.0中进行大量删除操作时，应该能从应用程序中删除直接JDBC的最后一位。到此时为止，必须获取数据库连接并向数据库直接提交SQL。 在确保没有关闭连接的情况下，为了释放资源，只关闭会话就足够了。出于手工编写许多JCBC代码来进行开发的习惯，第一个版本关闭了JDBC连接。因为通过配置Hibernate创建的连接池只带有一个链接，在第一个之后就完全破坏了测试。一定要注意这种情况！ 既然在测试类运行时（设想运行所有的测试实例）不能确定数据库的状态，应该在setUp()方法中包含数据库清除，如下所示: public void setUp() throws Exception { TestSchema.reset(). } 结束语 在使用像Hibernate这种复杂的O/R映射程序时，必须能够测试实际存在（real-live）的RDBMS，而不会发生任何针对已部署数据库的争论。虽然Hibernate有内置模式生成工具，让此类测试特别简单，但是在这里展示的例子并不排除Hibernate，并且可能与JDO或TopLink一起运行。使用上面描述的设置，您不必离开舒适的IDE环境，但仍然可以对代码进行大量测试。 