数据库系统设计与开发

讲义

浙江大学城市学院

目录

第 1 章 数据库系统实例剖析

本章以基于 Java 语言和 MySql 数据库的图书管理系统为例，分析其功能实现形式，帮助读者建立数据库应用系统基本形式的概念、组成部分、以及 DBMS 和应用程序在应用系统中的地位的概念；通过分析图书管理系统的开发步骤，引出后续章节的安排：数据库逻辑结构设计（以数据建模为目的）→数据库实施和管理（数据库的设计在 MySql 数据库中的实施）→数据库应用程序设计开发（Java 应用程序如何存取数据库中的数据） →数据库物理结构优化（以提高系统性能为目的进行数据库优化设计）。

1.1 图书管理系统功能剖析

作为教材的第一个例子，旨在向读者介绍数据库系统的一般组成以及开发数据库系统的一般过程；为此，我们选择了读者耳熟能详的图书管理系统。该系统能完成图书的上架、借阅、归还、下架等基本功能；并实现读者信息的管理功能。系统的用户包括图书馆工作人员、系统管理员和读者，其中，工作人员和系统管理统称为系统用户，其功能如下图所示。

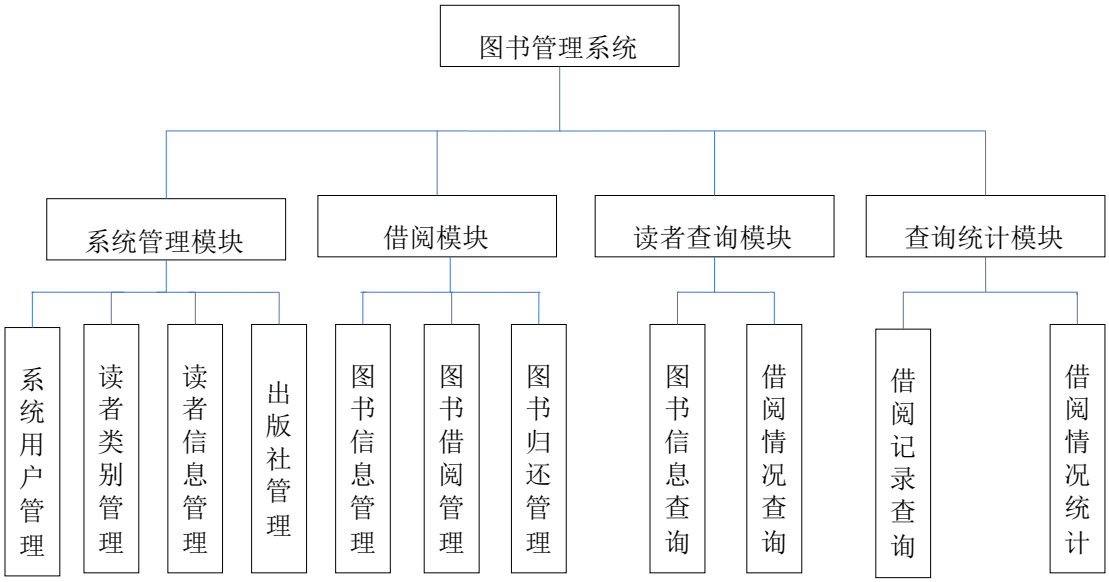


图 1-1 图书管理系统功能视图

用户必须登陆后才能使用相关功能，其中系统管理员可以使用“系统管理模块”和“查询统计模块”，图书馆工作人员可以使用“图书借阅模块”和“查询统计模块”，读者可以使用“读者查询模块”。

1.1.1 系统管理模块

1.系统用户管理

功能概述：实现对系统管理员和图书馆工作人员的管理功能。包括用户的添加、密码重置、用户注销等，其主界面如下图所示。系统管理员可通过上方的工具栏进行相关操作。

图 1-2 系统用户管理

功能剖析：该模块是一个典型的增删改模块，完成系统用户信息的增加（用户的添加）、删除（注销用户）和修改（密码重置）。**该功能反映到数据库中，就是完成系统用户表记录的增加、删除、修改。**

2. 读者类别管理

功能概述：读者按类别进行管理，如，某图书馆将读者分为本科生、研究生、教师等。可以为不同类别的读者设定默认的可借图书数量。

图 1-3 读者类别管理

功能剖析：该模块也是一个典型的增、删、改模块，其操作也直接反映为对应数据库表的增删改操作。同时，这里还体现了数据之间的关联，删除某读者类别时，如果已经存在该类别的读者，系统应如何进行响应？这是典型的外码违例处理模式，请读者思考。

3. 读者信息管理

功能概述：实现读者的注册、读者信息的修改、读者的注销、密码重置、挂失等功能。

图 1-4 读者管理

功能剖析：该模块和系统用户管理模块功能非常类似，也是完成读者信息表的记录增删改；需要注意的是，录入读者信息时，读者类别需要用户选择，而读者类别信息来自读者类别表；另外，选择读者类别后，应自动将读者的可借图书数量设置为读者类别中预定义的数量。

4. 出版社管理

功能概述：所有图书都属于指定的出版社，为简化例子，这里只列出了出版社的一小部分信息。

图 1-5 出版社管理

功能剖析：该模块也是一个实现简单增、删、改功能的模块，同样需要注意删除出版社时，如果该出版社下已经存在图书时的系统处理方式。

5. 图书信息管理

功能概述：完成图书的上架（也就是录入图书信息）、图书信息的修改、图书的下架（也就是图书信息的删除）。我们约定每本图书都是不同的，具有唯一的编号；对于同种书籍，每一本的编号也是不同的。已经借阅在外的图书不能进行下架。图书下架后不能进行借阅。所有的图书信息都不删除（即使已经下架，数据也保留在数据库中），方便后续查询。

图 1-6 图书管理

功能剖析：该模块和读者管理模块非常类似，用于完成图书信息表的数据维护。

1.1.2 图书借阅模块

图书借阅模块由图书馆工作人员操作，借书和还书工作一般由不同的人进行操作。因此将借阅和归还操作在两个模块中进行。

1. 图书借阅管理

功能概述：借阅时，首先通过条码扫描枪读取读者证号，系统立即显示该读者的基本信息，包括姓名和状态；并在下方显示该读者已经借阅的图书信息。然后由工作人员逐本扫描图书条码（图书编号），同时显示相应图书的信息，确认后点击“借阅”按钮完成图书的借阅。

图 1-7 图书借阅

功能剖析：图书借阅时，对数据库的操作可以概括为，增加一条借阅记录，修改图书的状态信息。这些操作将涉及多张表，因此，需要注意采用数据库事务的方式实现功能。

2. 图书归还管理

功能概述：图书的归还操作更为简单，只要扫描或输入图书编号，系统自动显示该图书的信息，以及借阅读者的信息（包括该读者借阅的所有图书信息）。如果该图书为超期未还图书，则自动计算出滞纳金。确认后，点击“还书”按钮即可完成还书操作。

图 1-8 图书归还

功能剖析：为能在图书归还后能查阅借阅记录，图书归还时，对数据库的操作可以概括为，修改借阅记录（写入归还时间和滞纳金），修改图书的状态信息。这些操作将涉及多张表，因此，需要注意采用数据库事务的方式实现功能。

1.1.3 查询模块

1. 图书借阅情况查询

功能概述：图书借阅情况查询模块提供按图书条码查询图书借阅情况的功能。

图 1-9 图书查询

功能剖析：这是一个典型的查询模块，需用程序利用界面中输入的数据，组织 SQL 语句进行数据库内容查询。

2. 读者借阅情况查询

功能概述：该模块查询读者的历史借阅数据。

图 1-10 读者借阅记录

功能剖析：这是也是一个典型的查询模块，需用程序利用界面中输入的数据，组织 SQL 语句进行数据库内容查询。

1.1.4 统计模块

1. 图书借阅统计

功能概述：允许图书馆工作人员和系统管理员按统计所有图书的借阅情况。

图 1-11 图书借阅情况统计

2. 读者借阅统计

功能概述：允许图书馆工作人员和系统管理员统计读者的借阅情况。

图 1-12 读者借阅情况统计

功能剖析：首先组织 SQL 语句查询相关内容。

1.2 图书管理系统的组成

前面一节，我们从用户的角度分析了图书管理系统的基本功能和组成部分。那么，站在开发者的角度，我们又应该怎么来看待这个系统？一般认为：

软件 = 数据 + 程序

也就是说：

图书管理系统 = 图书管理系统相关的数据 + 图书管理系统应用程序

因此，我们可以从程序和数据两个方面来剖析这个系统。在数据库系统原理课程中，我们已经学习了数据库系统的一般组成，以及通过数据库存储、处理数据的基本方法。即，通过**数据库管理系统（DBMS）**统一进行数据的管理。而**应用程序**的目的就是为用户提供操作界面、从 DBMS 中提取数据、将处理后的数据存储到 DBMS 中。当然，为实现这些功能，还需要在一定的**应用开发环境**中完成相关的程序设计。

1.2.1 数据库系统开发涉及的知识体系

《数据库系统原理》是第一门数据库系统相关的课程，其侧重点是让读者理解数据库内部的逻辑结构，以及如何通过 DBMS 进行数据定义、管理，这是建设数据库系统的基础知识。从上一节的例子中可以看出，几乎所有功能最终都体现为对数据库的操作，即利用 SQL 语句完成数据库记录的增、删、改等。

“数据库设计”的知识一般也在数据库系统原理课程中进行介绍，其核心目的是设计数据库的逻辑结构。只有在设计了合理的数据库逻辑结构的基础上，才能通过 SQL 语句实现各种业务逻辑。

“数据库应用程序设计”是实现数据库系统的一个重要步骤，通常情况下，数据库应用程序负责通过界面采集用户的输入信息，并将这些信息以及业务特点，组织成 SQL 语句，并递交 DBMS 执行后实现业务逻辑。

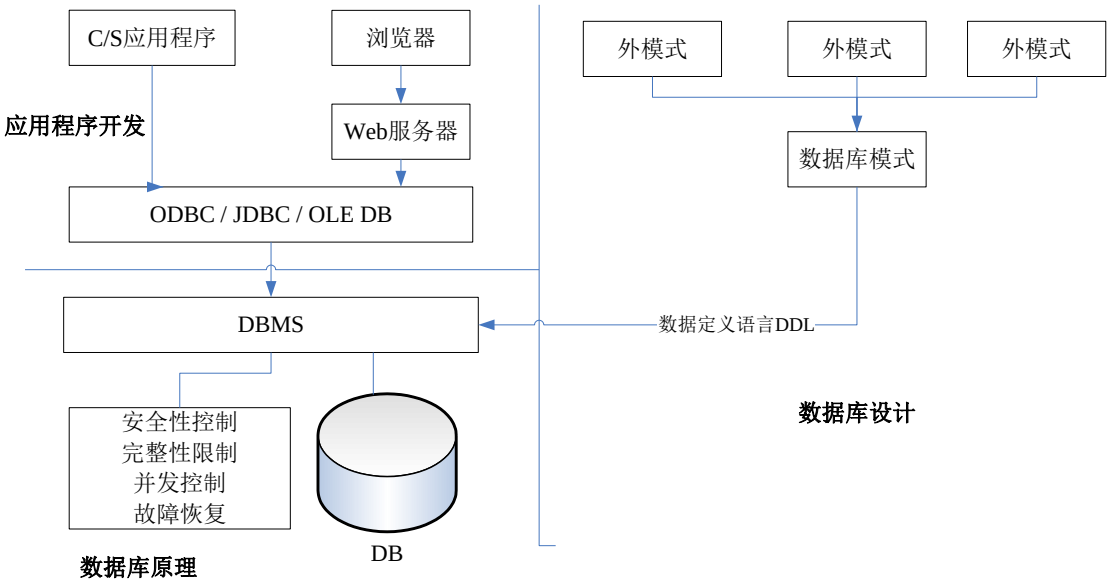


图 1-13 数据库系统开发涉及的知识体系

1.2.2 图书管理系统中的数据库管理系统

数据库管理系统是一种操纵和管理数据库的大型软件，用于建立、使用和维护数据库，简称 DBMS。它对数据库进行统一的管理和控制，以保证数据库的安全性和完整性。用户通过 DBMS 访问数据库中的数据，数据库管理员也通过 DBMS 进行数据库的维护工作。它可使多个应用程序和用户用不同的方法在同时或不同时刻去建立，修改和查询数据库。DBMS 提供数据定义语言 DDL 与数据操作语言 DML，供用户定义数据库的模式结构与权限约束，实现对数据的追加、删除等操作。目前，典型的数据库管理有甲骨文的 Oracle 系列、微软的 SQL Server 系列、IBM 公司的 DB2 系列、开源的 MySql 等数据库管理系统。图书管理系统采用 MySql 作为数据库服务器，下面简单介绍图书管理系统数据库的部署过程。

1. MySql 数据库服务器的安装配置，下载最新的 MySql 数据库服务器安装程序，目前的最新版本为 5.6。读者可自行在 MySql 官方网站下载。下面以随书光盘中的 32 位 windows 版的 MySql 安装为例介绍安装配置过程。

   第一步：运行光盘中的 mysql-installer-community-5.6.15.0.msi 程序，并在下图界面中，点击第一项：”install mysql products”，并在下一界面中选择同意 License terms，并点击“下一步”按钮；再下一界面中，提示用户是否进行新版本提醒，直接点击”Execute”进行数据安装。

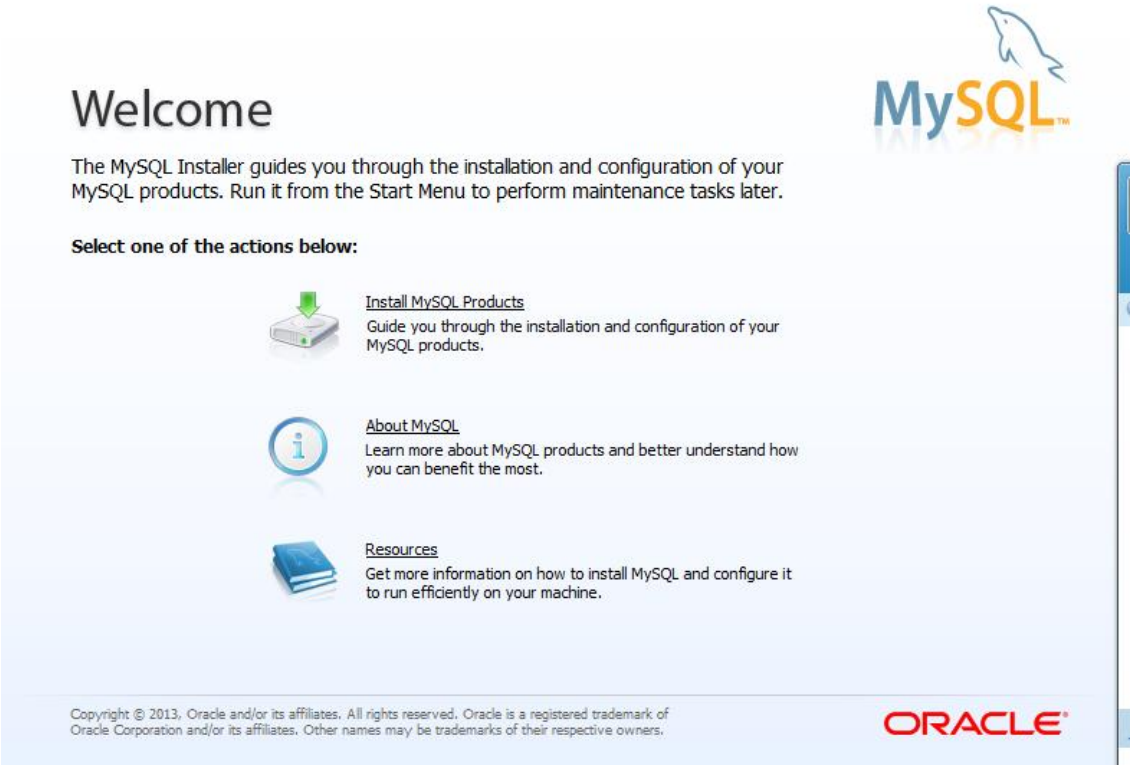


图 开始安装 MySql

第二步：选择安装选项，如下图所示，选择安装类别（采用默认）；安装位置等，请根据自身计算机情况选择。点击“Next”，并在下一界面中不做任何更改，直接点击“Execute”开始安装。

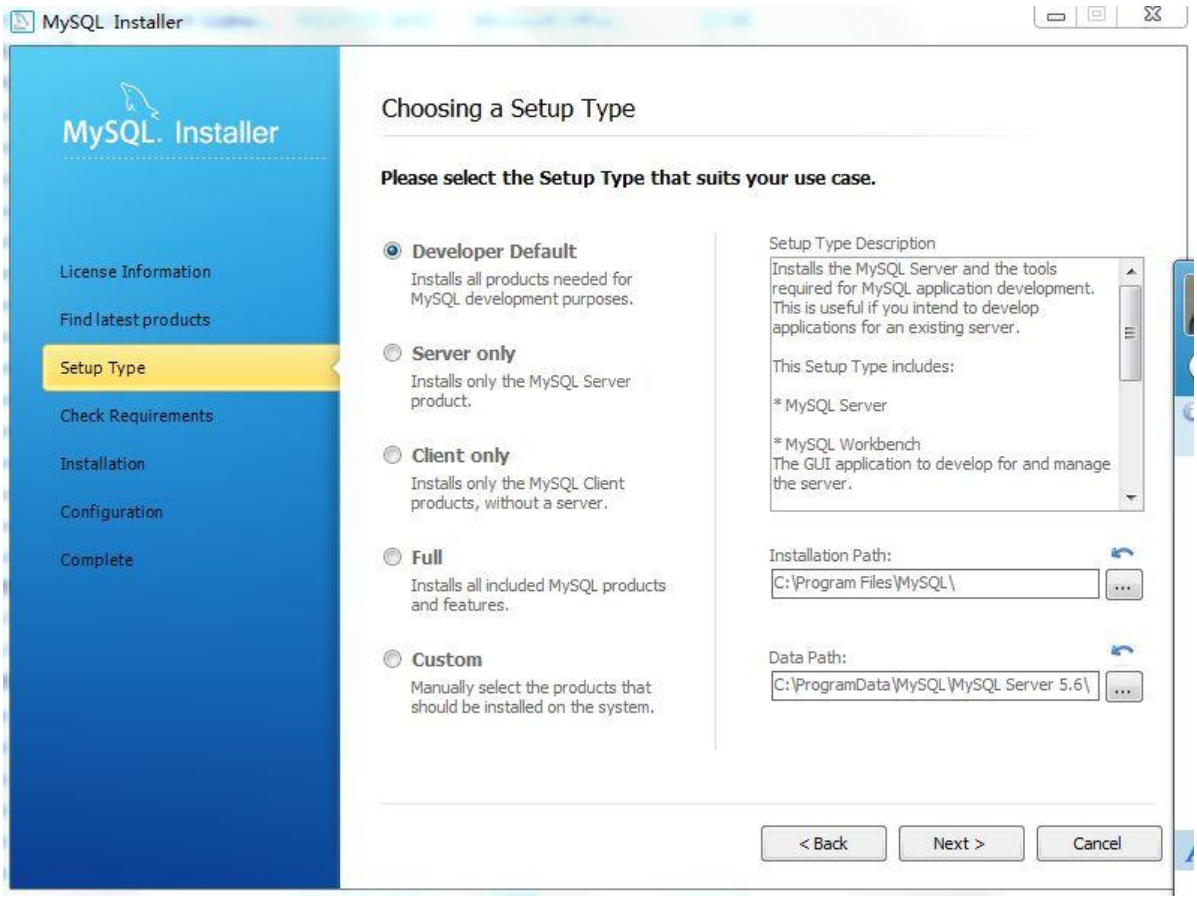

MySQL Installer
MySQL. Installer
License Information
Find latest products
Setup Type
Check Requirements
Installation
Configuration
Complete
Choosing a Setup Type
Please select the Setup Type that suits your use case.
Developer Default
Installs all products needed for MySQL development purposes.
Server only
Installs only the MySQL Server product.
Client only
Installs only the MySQL Client products, without a server.
Full
Installs all included MySQL products and features.
Custom
Manually select the products that should be installed on the system.
Setup Type Description
Installs the MySQL Server and the tools required for MySQL application development. This is useful if you intend to develop applications for an existing server.
This Setup Type includes:
* MySQL Server
* MySQL Workbench
The GUI application to develop for and manage the server.
Installation Path:
C:\Program Files\MySQL\
Data Path:
C:\ProgramData\MySQL\MySQL Server 5.6\
< Back
Next >
Cancel

图 Mysql 安装选项

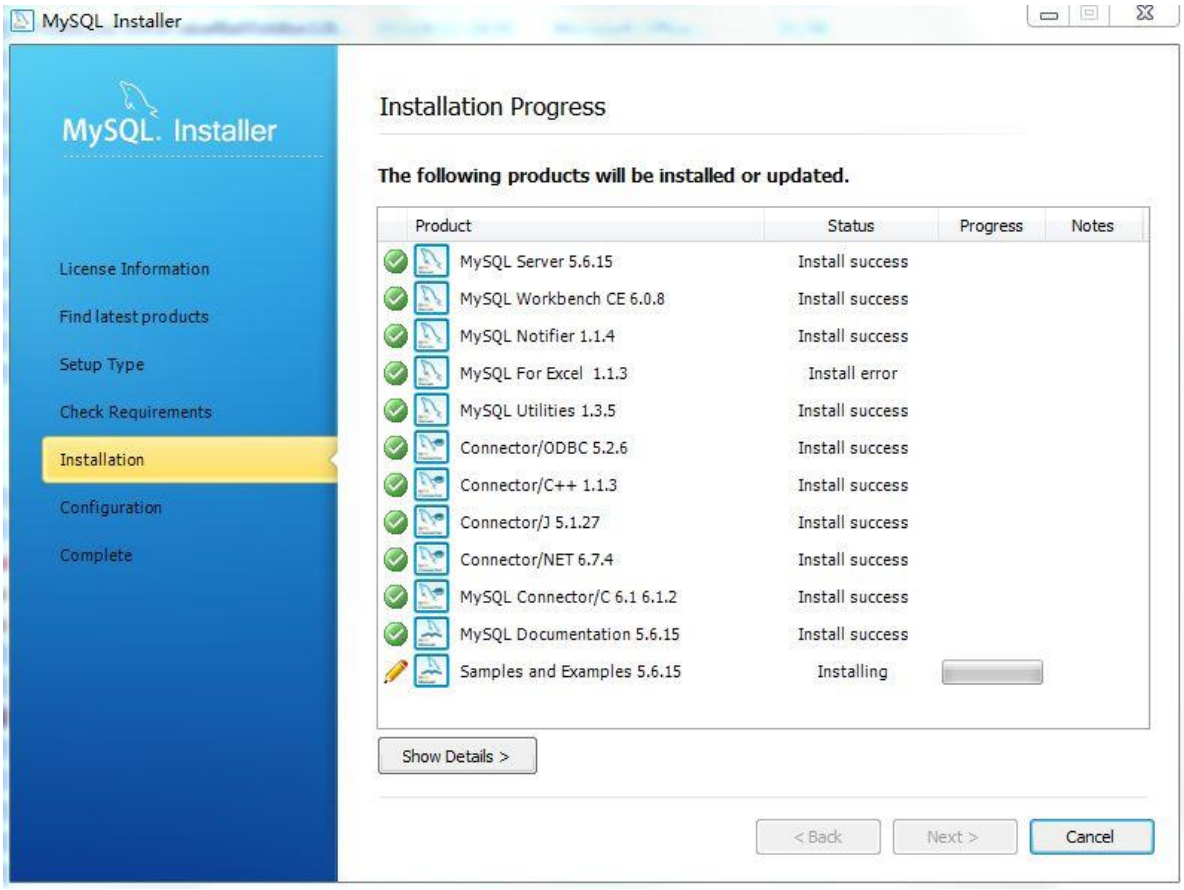


图 MySql 安装

第三步：MySql 基本配置。安装完成后，点击“Next”按钮显示如下界面进行配置。一般情况下不需要修改配置。点击“next”进入 MySql 用户配置。

图 MySql 基本配置

第四步：MySql 用户配置，在如下界面中设置数据库用户信息。其中默认管理员用户为“root”，请在此界面中设置其密码。如果需要添加其他用户用户，则请读者在界面的下方自行添加。

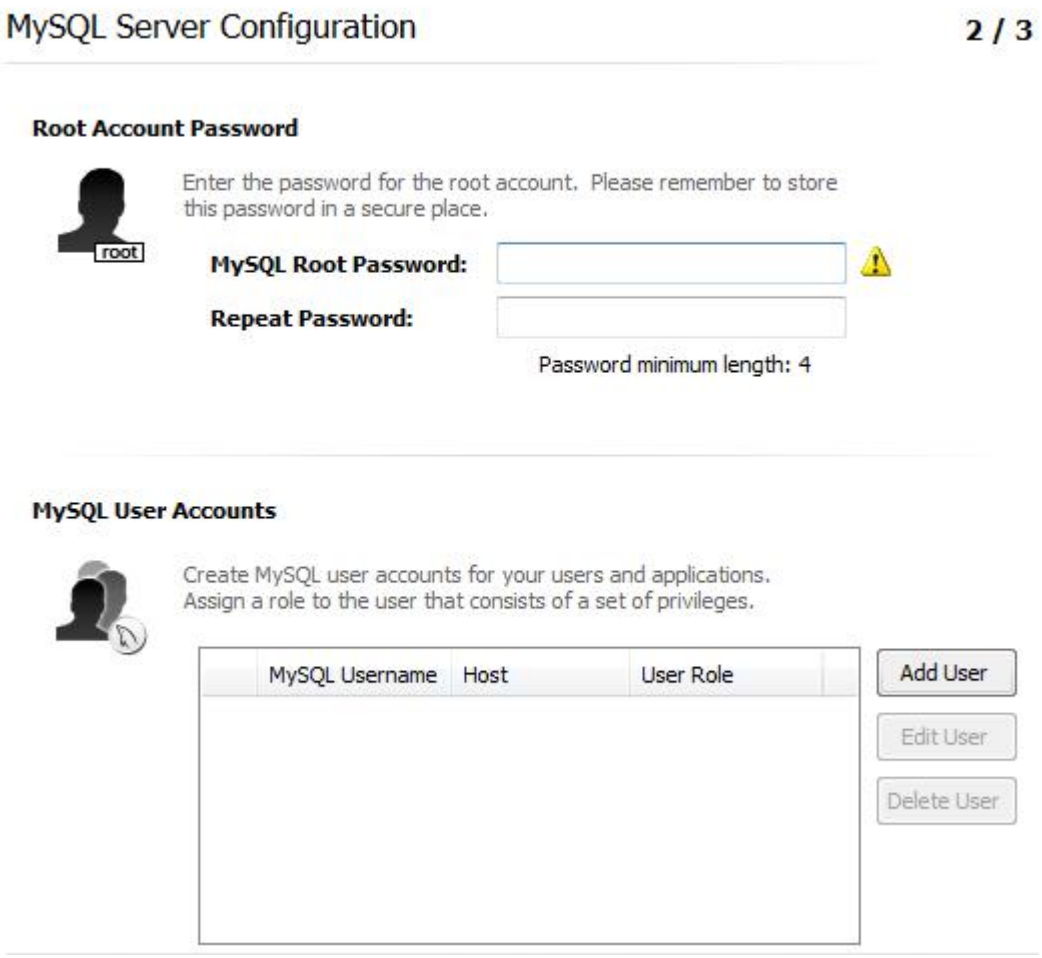


图 MySql 用户配置

第五步：配置 MySql 的 windows 服务名。用户配置完成后，点击下一步，进行 windows 服务配置，一般采用默认配置即可，如下图所示。

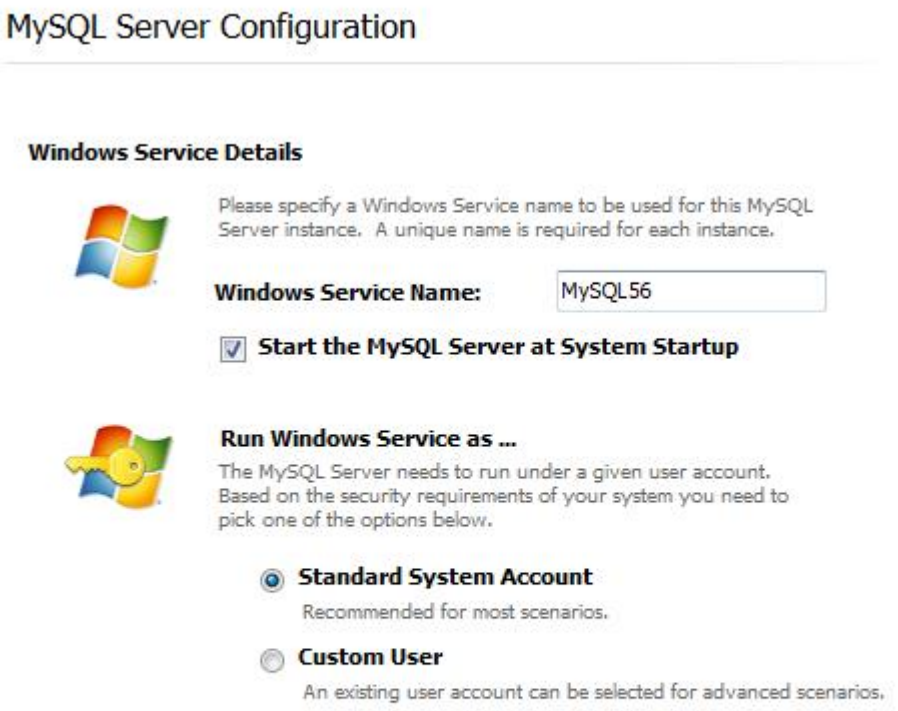
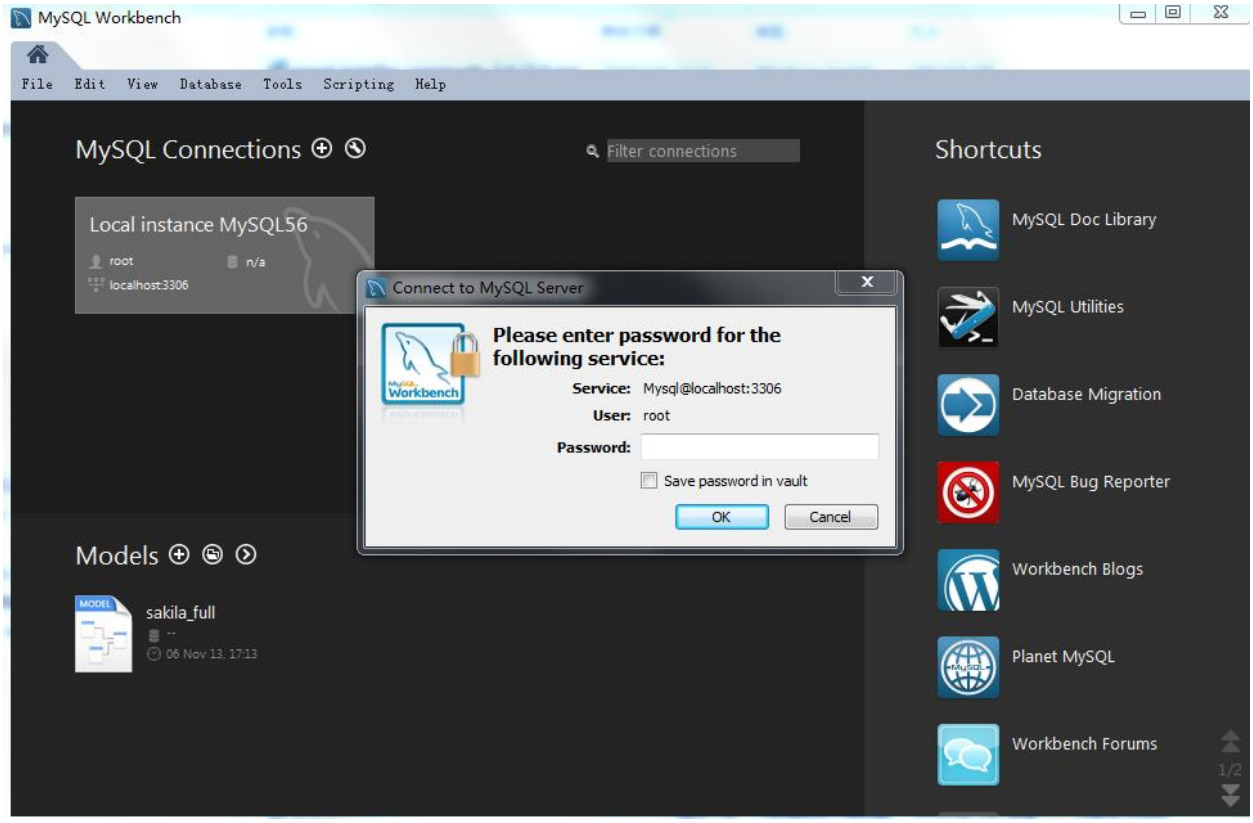


图 MySql Windows 服务配置

2. 图书管理系统数据库实施

第一步：MySql 数据库安装完成后，可通过“MySql Workbench”工具进行数据管理。在下图中，打开数据库服务器（点击左侧的“Local instance MySql56”后，输入安装时设置的 root 账户密码）。



第二步：建立图书管理系统数据库（booklib）。进入数据库服务器管理界面后，通过创建 Schema 工具栏建立新数据库。如下图所示。

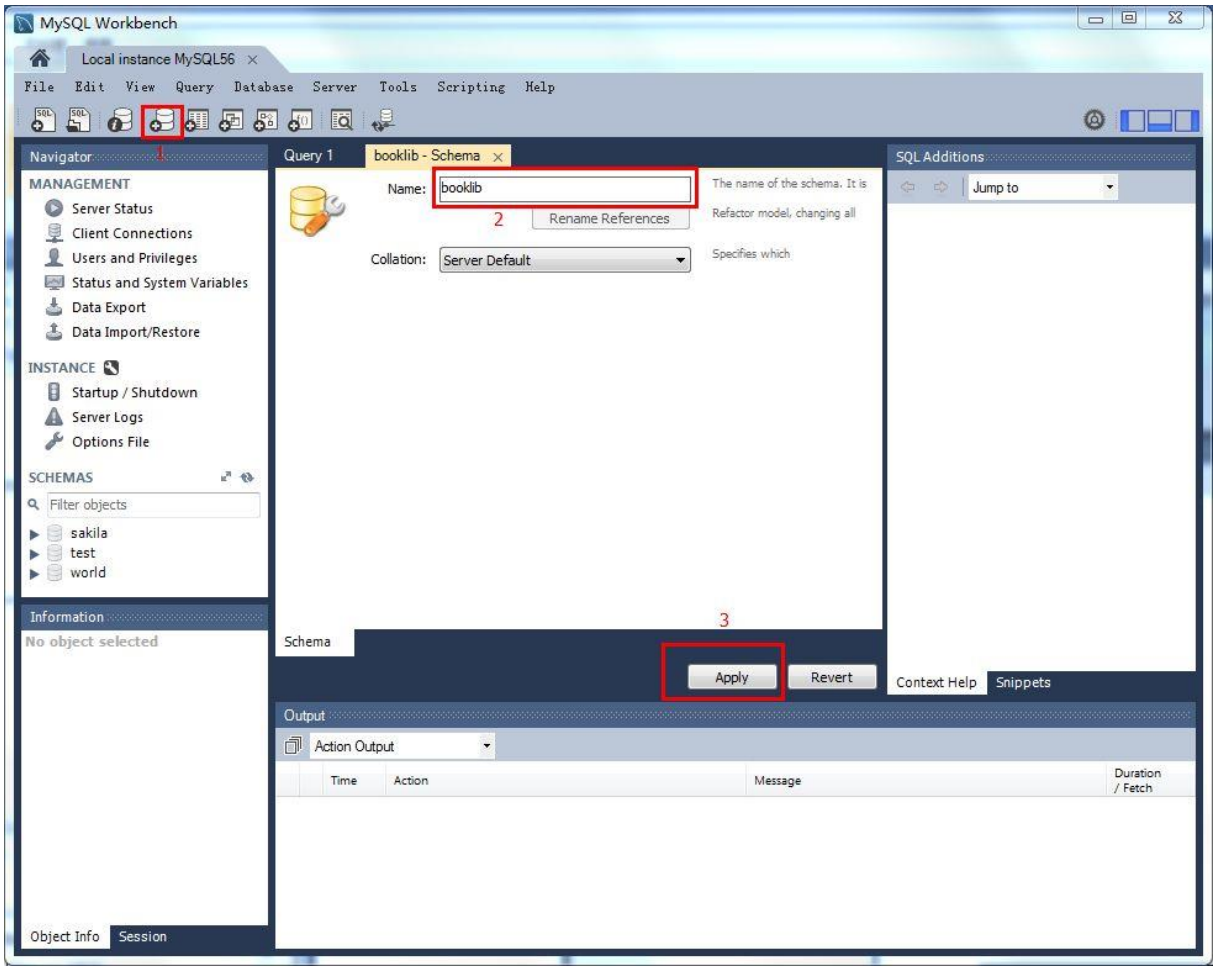


图 创建 booklib 数据库

第三步：执行 booklib 库的建库脚本，并内建默认用户 admin。打开随书光盘中的 booklib.sql 文件，如下图所示，并执行该脚本。

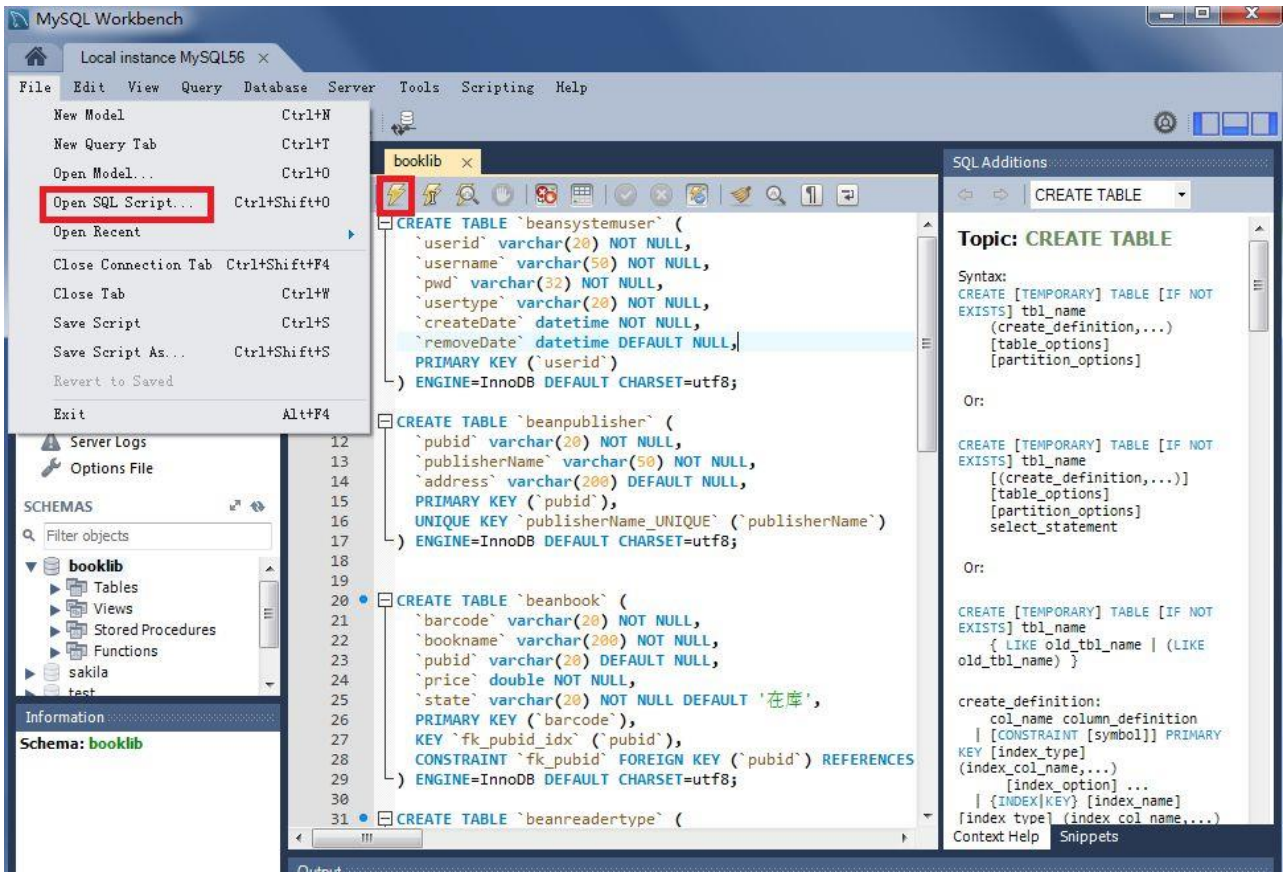


图 建库脚本

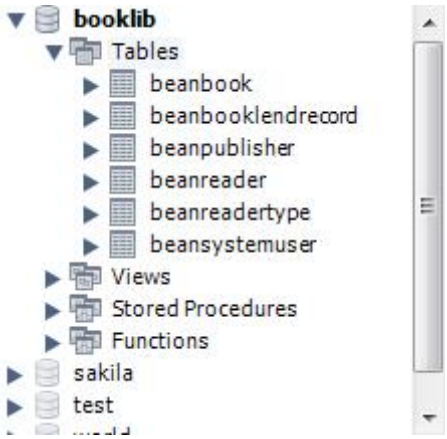


图 刷新后数据库表

1.2.3 数据库应用系统的基本架构

开发一个数据库应用系统首先要确定系统的总体结构。目前主要有两大体系结构：一是基于操作系统平台的客户机/服务器（Client/Server，C/S）结构，二是基于浏览器的浏览器/服务器结构。

1. 客户机 / 服务器结构

C/S 模式是 20 世纪 80 年代逐渐发展起来的系统模式，该系统模式将与 DBMS 相关的工作量分为两部分：服务器和客户机。客户机和服务器典型地是运行在不同的系统中的。

在客户机和服务器之间如何划分 DBMS 功能有不同的方案。大多数关系 DBMS 产品采用的方案是在服务器中包含集中式 DBMS 功能，为客户机提供一个 SQL 服务器，负责管理数据并进行事务管理；每一个客户机必须配置合适的 SQL 查询，并为用户提供用户接口和编程语言接口功能。客户机可以引用存在在各种 SQL 服务器中的包含数据分布信息的数据字典，还可以访问一些功能模块，将一个全局查询分解为若干可以在不同站点上执行的局部查询。

由于系统扩展性、维护成本、安全性等问题，两层客户机/服务器技术正逐步被三层体系结构所取代，尤其在 Web 应用中。如图 1-14 所示，在三层客户机/服务器体系结构中，有以下 3 个层次：

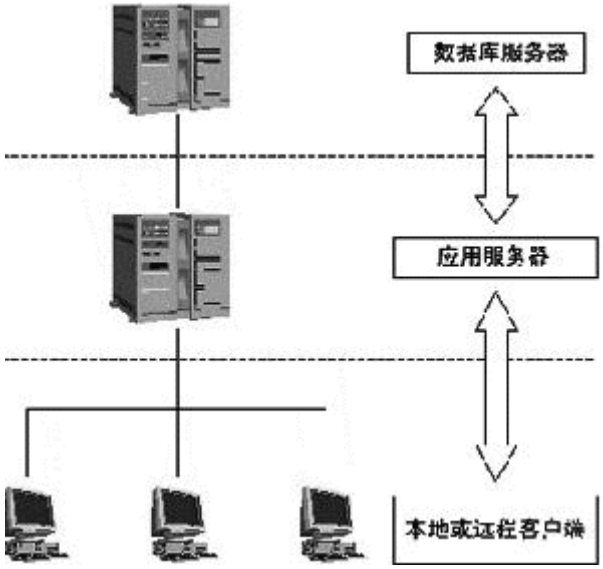


图 1-14 三层客户机/服务器体系结构

- 表示层（客户层）：这一层提供了用户界面并同用户进行交互。
- 应用层（业务逻辑层）：该层对应用逻辑进行编程，也是表示层和数据库服务器层的桥梁，它响应表示层的用户请求，从数据库服务器层抓取数据，执行业务处理，并将必要的数据传送给表示层以展示给用户。在这一层还可以处理附加的应用功能，例如安全检查、身份验证以及其他功能。需要时，应用层可以与一个或多个数据库或数据源进行交互。
- 数据库服务器层：这一层负责数据的存储、事务管理、数据完整性控制、故障恢复等，处理来自应用层的查询与更新请求，并发送结果。

2． 浏览器 / 服务器结构

随着 Internet 和 Web 的流行，以往的体系结构已无法满足当前的全球网络开放、互连、信息随处可见和信息共享的新要求，于是出现了浏览器 / 服务器（Browser/Server，B/S）结构的数据库系统。如图 1-15 所示，在该模式下，该结构由客户机、Web 服务器和数据库服务器三部分组成。只是客户机应用程序被浏览器所替代，用户通过浏览器访问服

务器，服务器接受相关浏览器的访问，对数据库进行操作，并将结果翻译成页面描述语言返回访问的浏览器。

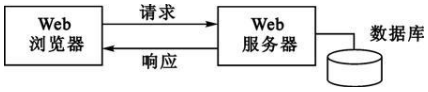


图 1-15 浏览器 / 服务器体系结构

浏览器/服务器结构显示了如下的优点。

- 在该结构中，客户端任何计算机只要安装了浏览器就可以访问应用程序。浏览器的界面是统一的，广大用户容易掌握，从而大大减少了培训时间和费用。
- 2.客户端的硬件与操作系统具有更长的使用寿命，因为它们只要能够支持浏览器软件即可。
- 3.由于应用系统的维护与升级工作都是在服务器上执行，因此不必安装、维护或升级客户端应用代码，大大减少了系统开发和维护代价。这种结构能够支持数万甚至更多的用户。

1.2.4 应用程序和开发环境

应用程序的目标是完成和 DBMS 的交互，以及和用户的交互，实现用户角度的业务数据和数据库存储的数据之间的转换。对于程序员而言，只要掌握和 DBMS 进行交互的方法、掌握程序设计语言实现数据库的逻辑处理、掌握用户交互界面的设计方法即可完成应用程序开发。应用程序开发环境为程序员提供源代码撰写工具，完成源代码的编译链接工作。通常情况下，一个开发环境对应一种编程语言，而一种编程语言会有多种开发环境（如，Visual C++和 borland C++的编程语言都为 C++）。同时，为简化编程，开发环境还会提供一系列类库（如 Visual C++的 MFC 库、Delphi 的 VCL 等等），实现一些常用的功能。通过这些类库，程序员仅需编写少量代码就可获取相应的功能。因此，在学习了编程语言的基础上，应用程序开发环境的学习还包括：①源代码编辑工具的学习；②相关类库的学习。

目前，Java 和.Net 是两大主流的开发平台。本教材将以 Java 平台为例介绍数据库应用系统的开发。

1. Java 平台及 Eclipse

java 最初被命名为 Oak，目标设定在家用电器等小型系统的编程语言，来解决诸如电视机、电话、闹钟、烤面包机等家用电器的控制和通讯问题。由于这些智能化家电的市场需求没有预期的高，Sun 放弃了该项计划。就在 Oak 几近失败之时，随着互联网的发展，Sun 看到了 Oak 在计算机网络上的广阔应用前景，于是改造了 Oak，以“Java”的名称正式发布。Java 编程语言的风格十分接近 C、C++ 语言，是一个纯面向对象的程序设计语言，它继承了 C++ 语言面向对象技术的核心，舍弃了 C ++语言中容易引起错误的指针、运算符重载、多重继承等特性，增加了垃圾回收器功能用于回收不再被引用的对象所占据的内存空间，使

得程序员不用再为内存管理而担忧。在 Java SE 1.5 版本中，Java 又引入了泛型编程、类型安全的枚举、不定长参数和自动装/拆箱等语言特性。

Java 不同于一般的编译执行计算机语言和解释执行计算机语言。它首先将源代码编译成二进制字节码(bytecode)，然后依赖各种不同平台上的虚拟机来解释执行字节码，从而实现了“一次编译、到处执行”的跨平台特性。不过，每次的编译执行需要消耗一定的时间，这同时也在一定程度上降低了 Java 程序的运行效率。与传统程序不同，Sun 公司在推出 Java 之际就将其作为一种开放的技术，全球数以万计的 Java 开发公司被要求所设计的 Java 软件必须相互兼容。

Eclipse 是一个开放源代码的、基于 Java 的可扩展开发平台。就其本身而言，它只是一个框架和一组服务，用于通过插件组件构建开发环境。Eclipse 附带了一个标准的插件集，包括 Java 开发工具。目前大多数 Java 开发者使用 Eclipse 作为开发环境。2001 年 11 月，IBM 公司捐出价值 4,000 万美元的源代码组建了 Eclipse 联盟，并由该联盟负责这种工具的后续开发。Eclipse 允许在同一 IDE 中集成来自不同供应商的工具，并实现了工具之间的互操作，从而显著改变了项目工作流程，使开发者可以专注在实际的嵌入目标上。利用 Eclipse，我们可以将高级设计(也许是采用 UML)与低级开发工具(如应用调试器等)结合在一起。Eclipse 的最大特点是它能接受由 Java 开发者自己编写的开放源代码插件。Eclipse 为工具开发商提供了更好的灵活性，使他们能更好地控制自己的软件技术。

2. .Net 平台及 Visual Studio .Net

.NET Framework 是一个集成在 Windows 中的组件，它支持生成和运行下一代应用程序与 XML Web Services。.NET Framework 具有两个主要组件：公共语言运行时和 .NET Framework 类库。 公共语言运行时是 .NET Framework 的基础。 可以将运行时看作一个在执行时管理代码的代理，它提供内存管理、线程管理和远程处理等核心服务，并且还强制实施严格的类型安全以及可提高安全性和可靠性的其他形式的代码准确性。.NET Framework 的另一个主要组件是类库，它是一个综合性的面向对象的可重用类型集合，可以使用它开发多种应用程序，这些应用程序包括传统的命令行或图形用户界面 (GUI) 应用程序，也包括基于 ASP.NET 所提供的最新创新的应用程序。

Visual Studio .NET 是一套完整的开发工具，用于生成 ASP Web 应用程序、XML Web services、桌面应用程序和移动应用程序。Visual Basic .NET、Visual C++ .NET、Visual C# .NET 和 Visual J# .NET 全都使用相同的集成开发环境，该环境允许它们共享工具并有助于创建混合语言解决方案。另外，这些语言利用了 .NET Framework 的功能，此框架提供对简化 ASP Web 应用程序和 XML Web services 开发的关键技术的访问。

1.2.5 图书管理系统开发环境配置

图书管理系统在 Eclipse 环境下通过 java 语言开发。

1. JDK 安装

读者可以自行下载最新的 JDK 版本，随书光盘中提供了一个 1.6 版本，解压缩即可。**注：JDK 有安装版和压缩版两种，安装版的安装过程比较简单，这里不再介绍。**

2. Eclipse 安装配置

读者可以在 Eclipse 官方网站下载最新的 Eclipse 版本，下面我们以随书光盘中的版本为例介绍其安装配置过程。

第一步：解压缩 Eclipse 到硬盘目录。

第二步：运行安装目录下 Eclipse.exe，并根据提示选择目录作为 workspace。

第三步：JDK 设置。（安装版的 JDK 安装后，eclispe 应该能自动识别），通过 eclipse 的 windows→perference 菜单，启动如下图所示对话框。如果右侧没有默认 Jre，这通过“add”按钮添加即可。

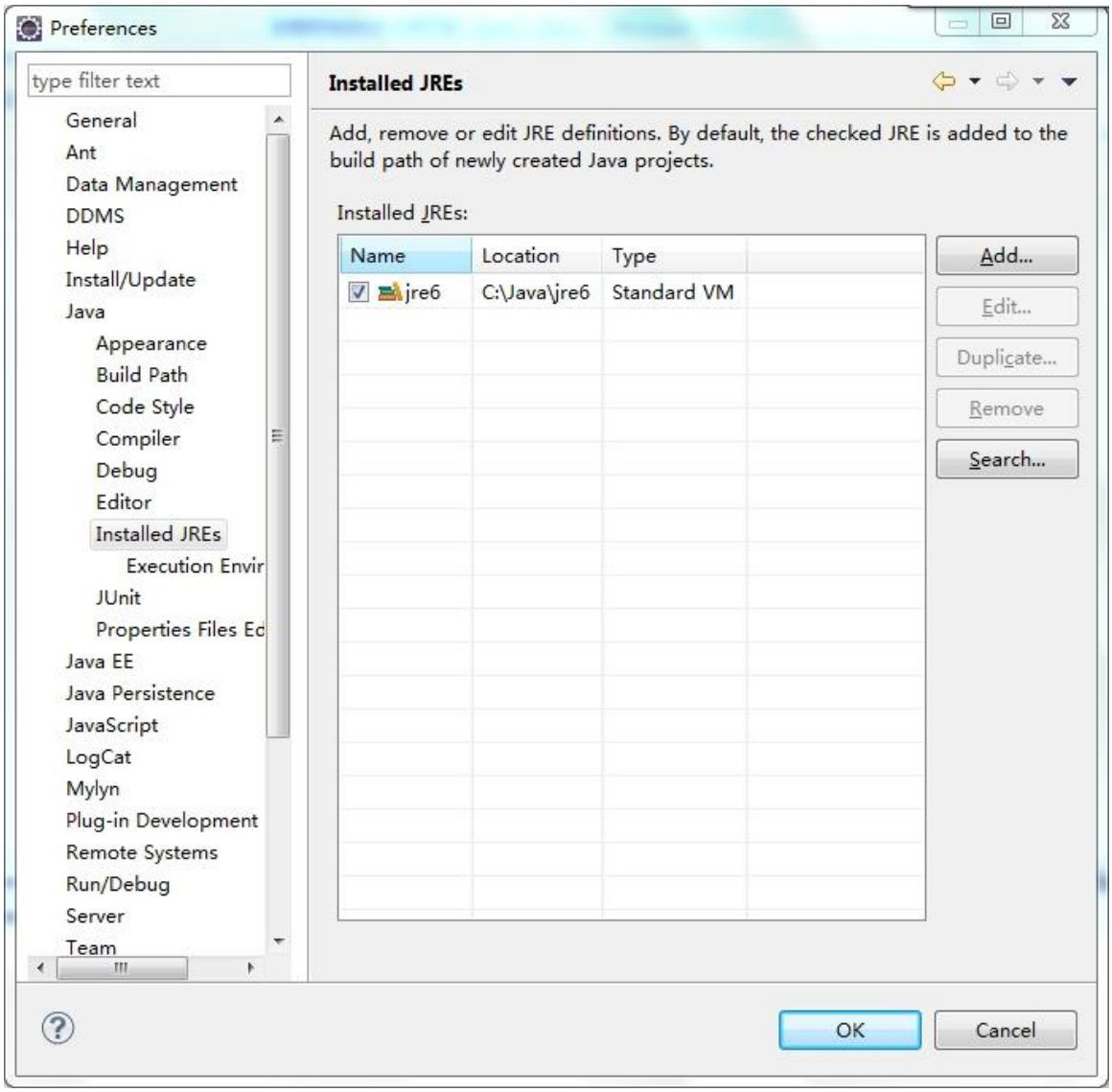


图 jre 设置

3. 图书管理系统工程创建

第一步：通过 eclipse 菜单 File→new→project…启动如下对话框。选择 java project 后，点击下一步。在下一界面中输入工程名称 booklib 后点击完成即可。

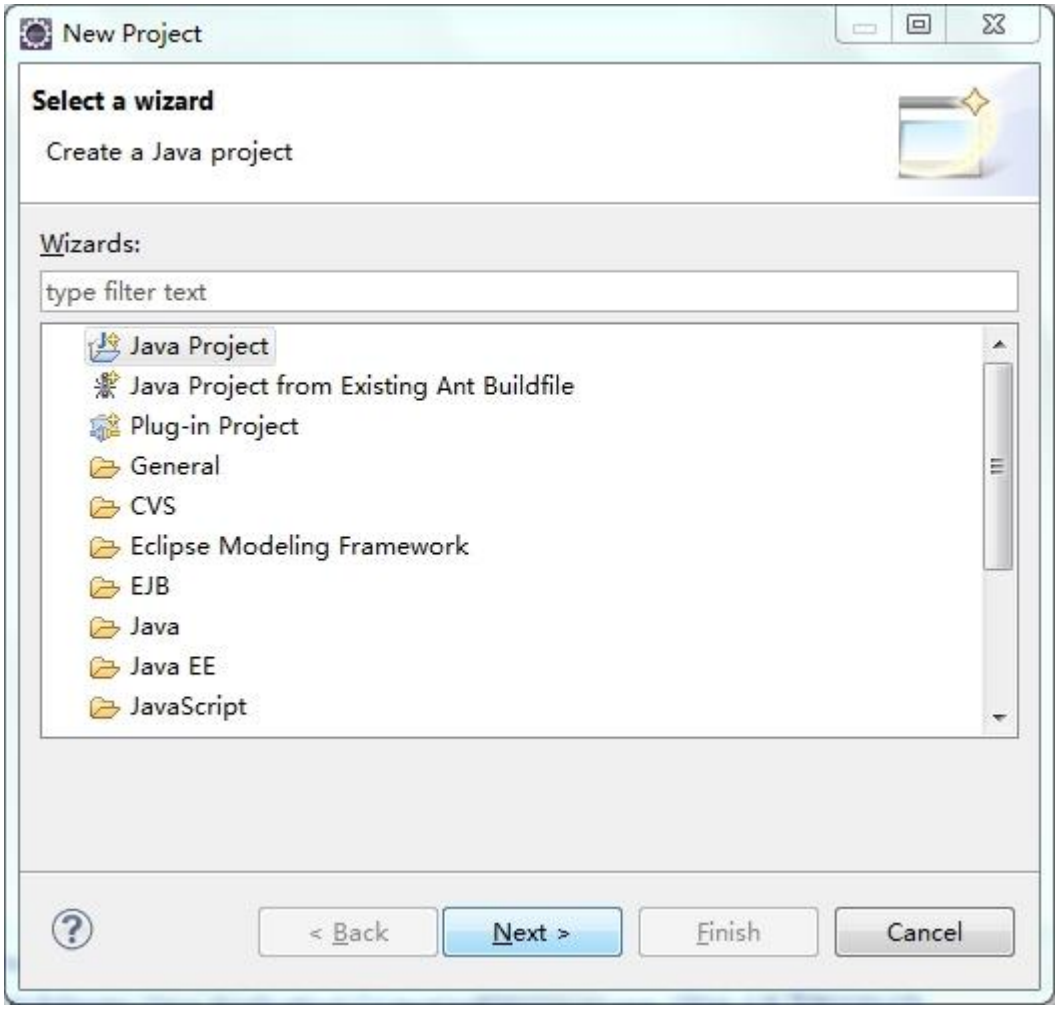
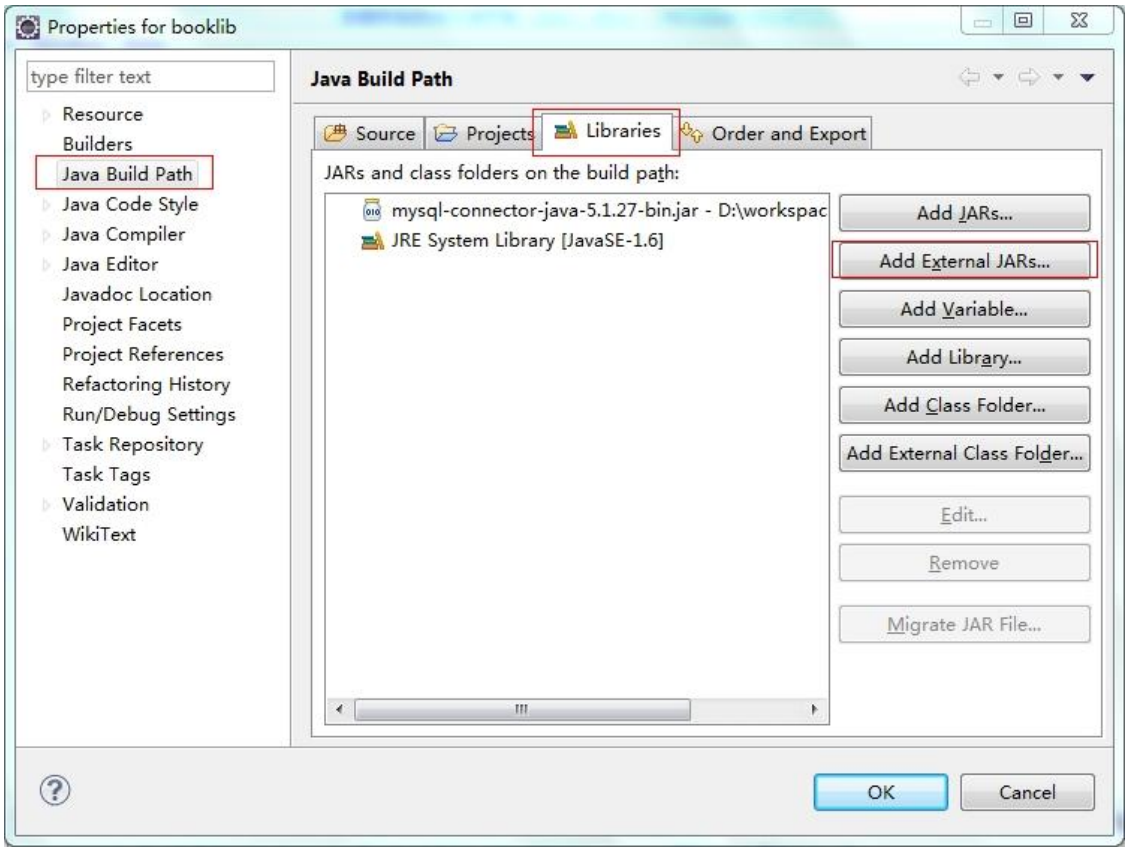


图 创建 java 工程

4. 代码导入。复制随书光盘的第一章目录下的 src 目录下的 cn 目录到工程的 src 目录即可（可以在 Eclipse 环境下，选中 src 后，通过 ctrl-v 快捷键直接黏贴）。

图 导入后代码结构

5. 类库导入。图书管理系统需要连接 MySql 数据库，需要导入 MySql 的 JDBC 类库，类库文件为随书光盘中的 mysql-connector-java-5.1.27-bin.jar 文件。通过 project→properties 菜单，启动如下对话框。点击“add external Jars”按钮选择 jar 文件即可。



6. 修改数据库连接信息，打开 cn.edu.zucc.booklib.util.DBUtil 类，并修改用户名、密码等信息即可。

```
public class DBUtil {
    private static final String jdbcUrl="jdbc:mysql://localhost:3306/booklib";
    private static final String dbUser="root";
    private static final String dbPwd="zucc";
    static{
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
    }
    public static Connection getConnection() throws java.sql.SQLException{
        return java.sql.DriverManager.getConnection(jdbcUrl, dbUser, dbPwd);
    }
}
```

7. 运行图书管理系统。右键点击右侧树形结构中的 cn.edu.zucc.booklib. BookLibStarter 类，选择 run→run as application 运行即可，默认账户和密码都是 admin

第2章　Jdbc 概述

Java 是一种可以开发跨平台应用软件的面向对象的程序设计语言，是由 Sun 公司于 1995 年 5 月推出的 Java 程序设计语言和 Java 平台（即 JavaSE, JavaEE, JavaME）的总称。Java 技术具有卓越的通用性、高效性、平台移植性和安全性，广泛应用于个人 PC、数据中心、游戏控制台、科学超级计算机、移动电话和互联网，同时拥有全球最大的开发者专业社群。Java 技术在数据库应用中，具有显著优势和广阔前景。

本章将介绍 java 平台开发数据库应用程序的基本方法，为突出数据库应用开发主题，本章仅涉及 Java 应用开发的一小部分内容，对于 Java 应用开发感兴趣的读者请参看相关书籍。

2.1 开发和运行环境准备

2.1.1 安装 JDK 和 Eclipse

JDK 现在最新版本是 JDK1.7（Java SE 7），可以从 java 官方网站下载 JDK 的最新版本以及历史版本。本教材的例子是基于 JDK1.6 的，读者可以自行下载安装。安装 JDK 很简单，只需要按照安装向导进行即可。安装完成后，在命令行窗口下，键入 java -version 命令可以查看到 JDK 版本信息。

JDK 安装完成后，读者可以从 Eclispe 官方网站“http://www.eclipse.org”中选择下载安装最新版本的 Eclipse。本教材采用目前最新的 3.7 版本，读者可以下载版本：Eclipse IDE for Java EE Developers。下载完成后，直接解压缩即可使用（运行目录下的 eclipse.exe 即可）。第一次启动 Eclipse 时，需求设置其工作目录（workspace），如下图 2-1 所示。

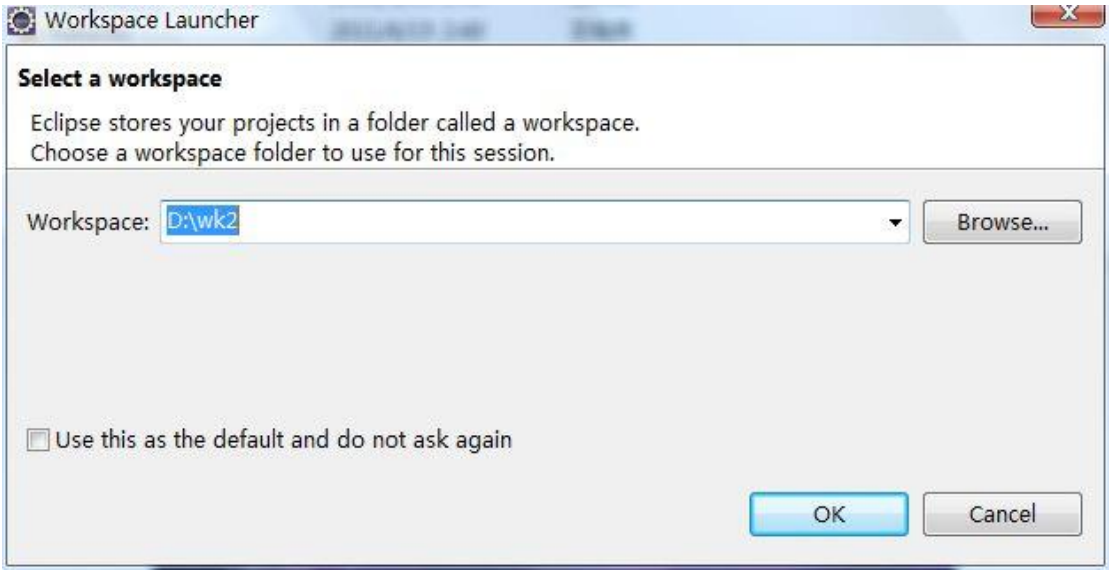


图 2-1 设置 Eclipse 的工作目录

2.1.2 在 Eclipse 中使用 JDBC

Java 应用程序通过 JDBC 连接数据库，完成对数据库的各项操作。为使 Java 应用程序能够访问特定的数据库，需要加载相应的 JDBC 驱动程序。这里我们以 MySql 下的图书管理系统数据库为例，介绍在 Eclipse 中，如何使应用程序能够连接到数据库。读者可以从 MySql 官方网站下载到 MySql 的 Java 驱动程序（在 MySql 的下载主页中，选择 MySql Connectors 中的 Connector/J）。在 Eclipse 中，加载 JDBC 驱动程序的方法有很多，只要在工程的 Build Path 中能够找到对应的 Jar 文件即可。工程的 Build Path 的查看方法是：菜单 Project→properties，并在弹出菜单中选择 Java Build Path→Libraries，如下图所示。

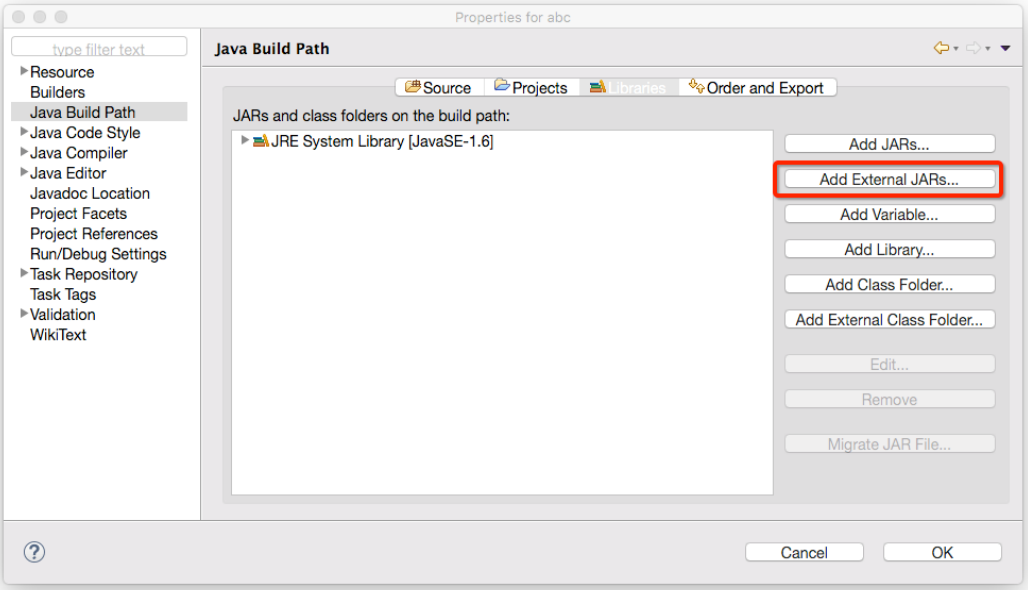


图 2-2 工程的 Build Path

在上图中，看到刚刚建立的工程的所有类库，我们需要把 jdbc 驱动程序的类库加入该工程中。解压缩下载的 MySql JDBC 驱动程序，复制其中的 mysql-connector-java-5.1.17-bin.jar 文件到磁盘后，点击“add external jars…”选中这个文件即可。

最后，可以建一个类进行数据库连接的测试。

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
public class TestJDBC {
    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
            String url = "jdbc:mysql://localhost:3306/booksys?user=root&password=your_password";
            Connection con = DriverManager.getConnection(url);
```

```
                Statement stmt = con.createStatement();
                String query = "select book_id,book_name from book";
                ResultSet rs = stmt.executeQuery(query);
                while (rs.next()) {
                    System.out.println("图书编号:"+rs.getString(1)
                        +",图书名称:"+rs.getString(2));
                }
              rs.close();
                stmt.close();
                con.close();
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
```

代码编写完成后，直接在代码编辑窗口点击鼠标右键，点击弹出菜单的“Run As→Java Application”菜单项即可运行该程序，查看是否运行成功。

2.2 用 JDBC 编写 Java 数据库应用

配置并测试完成上述过程后，再来看一下通过 JDBC 连接数据库，进行数据处理的基本方法。

1. JDBC 的工作机制

JDBC 定义了 Java 语言同 SQL 数据之间的程序设计接口。使用 JDBC 来完成对数据库的访问包括以下四个主要组件：Java 的应用程序、JDBC 驱动程序管理器、JDBC 驱动程序和数据源。JavaSoft 公司开发了 JDBC API，它是一个标准统一的 SQL 数据存取接口，为 Java 程序提供了一个统一地操作各种数据库的方法，程序员编程时，可以不关心它所要操作的数据库是哪个厂家的产品，从而提高了软件的通用性，只要系统安装了正确的驱动程序，JDBC 应用程序就可以访问其相关的数据库。

用 JDBC 来实现访问数据库记录可以采用下面的几个步骤：

① 通过驱动程序管理器获取连接接口。

② 获得 Statement 或它的子类。

③ 设置 Statement 中的参数。

④ 执行 Statement。

⑤ 遍历执行结果。

⑥ 关闭 Statement。

⑦ 处理其它的 Statement 。

⑧ 关闭连接接口。

2. 连接数据库

Java 程序连接数据库是通过相应的数据库提供的 JDBC 驱动程序进行的，因此，连接数据库首先需要加载这些驱动程序，加载方法和普通 Java 类的动态加载方式一样，也就是说，驱动程序就是一组类库。例如，加载 mysql 的驱动程序：

```
Class.forName("com.mysql.jdbc.Driver");
```

加载驱动程序后，需要指明所连接数据库的信息，在 JDBC 中，通过 JDBC Url 的方式来描述相应的数据库服务器信息。例如，MySql 数据库的 JDBC Url 的结构如下：

```
String url = "jdbc:mysql://localhost:3306/booksys?user=root&password=your_password";
Connection con = DriverManager.getConnection(url);
```

上述代码中，java.sql.DriverManager 类为 JDBC 驱动程序管理器，它可以根据 JDBC Url 识别连接数据库所需要的驱动程序以及其他信息，并通过驱动程序建立数据库连接对象 java.sql.Connection。

3. 数据查询及遍历

连接上数据库后，可通过个数据库连接建立 SQL 语句执行对象（在 JDBC 中，通过 API java.sql.Statement 接口表达）。

```
Statement stmt = con.createStatement();
```

java.sql.Statement 接口可以执行各种类型的 SQL 语句，包括 DDL、QL、DML 等。如果进行查询，则可以通过其 executeQuery 方法获取一个数据集对象（java.sql.ResultSet）。

例 2-1 查询并显示所有图书信息。

```
String query = "select book_id,book_name from book";
ResultSet rs = stmt.executeQuery(query);
```

对于 java.sql.ResultSet 对象可以通过下述方式进行数据集遍历。

```
while (rs.next()) {
    System.out.println("图书编号:"+rs.getString(1) +",图书名称:"+rs.getString(2));
}
```

4. 数据修改

通过 java.sql.Statement 接口也可以执行 DML 语句进行数据内容修改。

例 2-2 增加一种读者类别。

```
String   sql= "insert into reader_type(reader_type_code, reader_type_name, allow_count) "
            +"values ('005','临时读者',5)";
stmt.execute(sql);
```

通过上述代码可以看出，sql 语句对于 java 语言而言就是一个字符串，因此，只要能够根据业务需求组织好这个字符串，并通过相关 API 执行这个字符串即可实现各种业务逻辑。

5. 执行带参数的 SQL 语句

在大多数情况，特定业务逻辑的 SQL 语句结构式稳定的，不同时刻的业务过程，只是改变了语句中的某些值，如前面增加读者类别的代码中，读者类别信息通常是用户输入的，那么可以采用如下方式改造代码。

例 2-3 用带参数的 SQL 语句增加读者类别。

```
String strTypeCode="005";
String strTypeName="临时读者";
int nCount=5;
String    sql= "insert into reader_type(reader_type_code, reader_type_name, allow_count)"
        +" values (?,?,?)";
java.sql.PreparedStatement pst=con.prepareStatement(sql);
pst.setString(1, strTypeCode);
pst.setString(2, strTypeName);
pst.setInt(3, nCount);
pst.execute();
```

上例中，看到了一个新的 API，java.sql.PreparedStatement。通过这个 API，可以设计带参数的 SQL 语句。

2.3 JDBC 连接池

通过上述简单例子可知，利用 JDBC API 进行数据库的各种操作涉及的知识内容并不多。只要读者掌握了 java.sql 包下面的 DriverManager、Connection、Statement、PreparedStateme、ResultSet 等有限的几个 API，即可完成绝大多数数据库应用程序的编写。进一步分析，可以发现，对于数据库操作代码，能进行的性能优化工作很少，一般采用固定的模式进行。但是，如果每次需要执行数据库任务时，需要去连接数据库，结束后断开数据库连接，会造成频繁的连接断开操作，从而影响性能。

一般情况下，采用数据库连接池技术来解决这个问题。即，设置一个公共区域，所有应用模块向这个公共区域请求数据库连接，用完后放回这个区域。这样这些数据库连接可以被重用，读者可以自行编写连接池代码。这里给出一个简单的实现方法。

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
public class ConnectionPool {
private static Map<String,ConnectionPool> pools=new HashMap<String,ConnectionPool>();
static{
        //读取连接池配置,这里直接把连接信息写在代码中了
        try {
```

```
            ConnectionPool a=new ConnectionPool("com.mysql.jdbc.Driver",
            "jdbc:mysql://localhost:3306/booksys?user=root&password=your_password");
            pools.put("a",a);
            ConnectionPool b=new ConnectionPool("com.mysql.jdbc.Driver",
            "jdbc:mysql://localhost:3306/booksys2?user=root&password=your_password");
                pools.put("b",b);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
}
public static ConnectionPool getPool(String poolName){
            return pools.get(poolName);
}
private ConnectionPool(String driver,String url) throws ClassNotFoundException{
        Class.forName(driver);
        this.url=url;
}
private String url;
//当前空闲的数据库连接集合
private List<Connection> freeConnections=new ArrayList<Connection>();
//当前为特定模块提供服务的数据库连接结合
private List<Connection> busyConnections=new ArrayList<Connection>();
public synchronized Connection getConnection() throws SQLException{
        if(this.freeConnections.size()>0){
            Connection result=this.freeConnections.remove(this.freeConnections.size()-1);
            this.busyConnections.add(result);
            return result;
        }
        Connection result=DriverManager.getConnection(url);
        this.busyConnections.add(result);
        return result;
}
public synchronized void release(Connection conn){
        this.busyConnections.remove(conn);
        this.freeConnections.add(conn);
}
public static void main(String[] args){
        try {
            Connection conn=ConnectionPool.getPool("a").getConnection();
            //业务代码

           ConnectionPool.getPool("a").release(conn);
        } catch (SQLException e) {
            e.printStackTrace();
```

```
    }
}
}
```

ConnectionPool 类管理了一个连接池组，上例中，有两个连接池分别为“a”和“b”。最后是使用该连接池的实例。这样在多线程环境下，可以实现数据库连接池的简单管理。目前，由很多的数据库连接池产品，通过配置文件即可实现连接池的配置，同时也提供了各种 API 使用连接池。

第3章 jdbc 详解

JDBC（Java Data Base Connectivity,java 数据库连接），由一些接口和类构成的 API。J2SE 的一部分，由 java.sql,javax.sql 包组成。

应用程序、JDBC API、数据库驱动及数据库之间的关系

3.1 JDBC 的使用步骤

3.1.1 驱动注册

方式一：Class.forName(“com.mysql.jdbc.Driver”);

推荐这种方式，不会对具体的驱动类产生依赖。

方式二：DriverManager.registerDriver(com.mysql.jdbc.Driver);

会造成 DriverManager 中产生两个一样的驱动，并会对具体的驱动类产生依赖。

方式三：System.setProperty(“jdbc.drivers”, “driver1:driver2”);

虽然不会对具体的驱动类产生依赖；但注册不太方便，所以很少使用。

3.1.2.建立连接(Connection)

```
Connection conn = DriverManager.getConnection(url, user, password);
```

url 格式：

JDBC:子协议:子名称//主机名:端口/数据库名？属性名=属性值&…

User,password 可以用“属性名=属性值”方式告诉数据库；

其他参数如：useUnicode=true&characterEncoding=GBK。

3.1.3 创建执行 SQL 的语句(Statement)

```
Statement st = conn.createStatement();
st.executeQuery(sql);
PreparedStatement
String sql = “select * from table_name where col_name=?”;
PreparedStatement ps = conn.preparedStatement(sql);
ps.setString(1, “col_value”);
ps.executeQuery();
```

3.1.4.处理执行结果(ResultSet)

```
ResultSet rs = statement.executeQuery(sql);
While(rs.next()){
rs.getString(“col_name”);
rs.getInt(“col_name”);
//…
}
```

3.1.5.释放资源

释放 ResultSet, Statement,Connection.

数据库连接（Connection）是非常稀有的资源，用完后应马上释放，如果 Connection 不能及时正确的关闭将导致系统宕机。Connection 的使用原则是尽量晚创建，尽量早的释放。

下面来看一下完整的 Demo:

工具类：JdbcUtils

```
package com.weijia.firstdemo;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class JdbcUtils {
```

```
    private static String user = "root";
    private static String password = "123456";
    private static String dbName = "test";
    private static  String url = "jdbc:mysql://localhost:3306/"+dbName+"?user="+user+"&password="+password+"
&useUnicode=true&characterEncoding=gb2312";
   /**
     * 加载驱动
     */
    static{
        try{
            Class.forName("com.mysql.jdbc.Driver");
        }catch(Exception e){
            System.out.println("Exception:"+e.getMessage()+"");
            throw new ExceptionInInitializerError(e);
        }
    }

    private JdbcUtils(){
    }

    /**
     * 获取连接
     * @return
     * @throws SQLException
     */
    public static Connection getConnection() throws SQLException{
        return DriverManager.getConnection(url);
    }



    /**
     * 释放资源
     * @param rs
     * @param st
     * @param conn
     */
    public static void free(ResultSet rs,Statement st,Connection conn){
        try{
            if(rs != null){
                rs.close();
            }
        }catch(SQLException e){
            e.printStackTrace();
        }finally{
```

```
            try{
                if(st != null){
                    st.close();
                }
            }catch(SQLException e){
                e.printStackTrace();
            }finally{
                try{
                    if(conn != null){
                        conn.close();
                    }
                }catch(SQLException e){
                    e.printStackTrace();
                }
            }
        }

    }

}
```

测试类：

```
package com.weijia.firstdemo;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class Demo {

    public static void main(String[] args) throws Exception{
        //测试代码：
        test();
        //标准规范代码：
        template();
    }

    //模板代码
    public static void template(){
        Connection conn = null;
        Statement st = null;
```

```
    ResultSet rs = null;
    try {
        conn = JdbcUtils.getConnection();
        //创建语句
        st = conn.createStatement();
        //执行语句
        rs = st.executeQuery("select * from user");
        //处理结果
        while(rs.next()){
            System.out.println(rs.getObject(1) + "\t" + rs.getObject(2) + "\t" + rs.getObject(3) + "\t");
        }
    }catch(SQLException e){
        e.printStackTrace();
    }catch(Exception e){
        e.printStackTrace();
    }finally{
        JdbcUtils.free(rs, st, conn);
    }
}

//测试
static void test() throws Exception{
    //注册驱动
    DriverManager.registerDriver(new com.mysql.jdbc.Driver());
    //通过系统属性来注册驱动
    System.setProperty("jdbc.drivers","");
    //静态加载驱动
    Class.forName("com.mysql.jdbc.Driver");

    //建立连接
    String url = "jdbc:mysql://localhost:3306";
    String userName = "root";
    String password = "";
    Connection conn = DriverManager.getConnection(url,userName,password);

    //创建语句
    Statement st = conn.createStatement();

    //执行语句
    ResultSet rs = st.executeQuery("select * from user");

    //处理结果
    while(rs.next()){
        System.out.println(rs.getObject(1) + "\t" + rs.getObject(2) + "\t" + rs.getObject(3) + "\t");
```

```
        }

        //释放资源
        rs.close();
        st.close();
        conn.close();
    }

}
```

注意：这里还要记住引入额外的 jar.这个网上很多的，这里使用的是 MySql,搜一下 MySql 驱动的 jar 就行了。这里我们将一些操作都放到一个工具类中，这种方式是很优雅的。

3.2 使用 JDBC 来实现 CRUD 的操作

我们这里就采用分层操作：Dao 层，Service 层

首先看一下 domain 域中的 User 实体

```
package com.weijia.domain;

import java.util.Date;

public class User {

    private int id;
    private String name;
    private Date birthday;
    private float money;

    public User(){

    }

    public User(int id,String name,Date birthday,float money){
        this.id = id;
        this.name = name;
        this.birthday = birthday;
        this.money = money;
    }

    public int getId() {
        return id;
    }
    public void setId(int id) {
```

```
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Date getBirthday() {
        return birthday;
    }
    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }
    public float getMoney() {
        return money;
    }
    public void setMoney(float money) {
        this.money = money;
    }

    @Override
    public String toString(){
        return "[id="+id+",name="+name+",birthday="+birthday+",money="+money+"]";
    }
}
```

再来看一下 Dao 层结构：

```
接口定义：
package com.weijia.domain;
public interface UserDao {
    //添加用户
    public void addUser(User user);
    //通过 userid 查询用户,id 是唯一的,所以返回的是一个 user
    public User getUserById(int userId);
    //更新用户信息
    public int update(User user);
    //删除用户信息
    public int delete(User user);
}


实现类：
package com.weijia.domain;
```

```
import java.sql.Connection;
import java.sql.Date;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

import com.weijia.firstdemo.JdbcUtils;

public class UserDaoImpl implements UserDao{

   /**
    * 添加用户
    */
   public void addUser(User user) {
      Connection conn = null;
      PreparedStatement st = null;
      try{
         conn = JdbcUtils.getConnection();
         String sql = "insert into user(id,name,birthday,money) values(?,?,?,?)";
         st = conn.prepareStatement(sql);
         st.setInt(1,user.getId());
         st.setString(2,user.getName());
         //日期格式的转换(utils.date 转化成 sql.date)
         st.setDate(3,new Date(user.getBirthday().getTime()));
         st.setFloat(4, user.getMoney());
         int count = st.executeUpdate();
         System.out.println("添加记录条数:"+count);
      }catch(Exception e){
         throw new DaoException(e.getMessage(),e);
      }finally{
         JdbcUtils.free(null, st, conn);
      }
   }

   /**
    * 删除用户
    */
   public int delete(User user) {
      Connection conn = null;
      PreparedStatement st = null;
      try{
         conn = JdbcUtils.getConnection();
         String sql = "delete from user where id=?";
         st = conn.prepareStatement(sql);
         st.setInt(1,user.getId());
```

```
            int count = -1;
            count = st.executeUpdate();
            System.out.println("删除记录条数:"+count);
            return count;
        }catch(Exception e){
            throw new DaoException(e.getMessage(),e);
        }finally{
            JdbcUtils.free(null, st, conn);
        }
    }

    /**
     * 通过 userId 获取用户信息
     */
    public User getUserById(int userId) {
        Connection conn = null;
        PreparedStatement st = null;
        ResultSet rs = null;
        try{
            conn = JdbcUtils.getConnection();
            String sql = "select * from user where id=?";
            st = conn.prepareStatement(sql);
            st.setInt(1,userId);
            rs = st.executeQuery();
            if(rs.next()){
                User user = new User();
                user.setId(userId);
                user.setName(rs.getString("name"));
                user.setBirthday(rs.getDate("birthday"));
                user.setMoney(rs.getFloat("money"));
                return user;
            }
        }catch(Exception e){
            throw new DaoException(e.getMessage(),e);
        }finally{
            JdbcUtils.free(rs, st, conn);
        }
        return null;
    }

    /**
     * 更新用户信息
     */
    public int update(User user){
```

```
        Connection conn = null;
        PreparedStatement st = null;
        try{
            conn = JdbcUtils.getConnection();
            String sql = "update user set name=?,birthday=?,money=? where id=?";
            st = conn.prepareStatement(sql);
            st.setString(1,user.getName());
            st.setDate(2,new Date(user.getBirthday().getTime()));
            st.setFloat(3,user.getMoney());
            st.setInt(3,user.getId());
            int count = 0;
            count = st.executeUpdate();
            System.out.println("更新的记录数:"+count);
            return count;
        }catch(Exception e){
            throw new DaoException(e.getMessage(),e);
        }finally{
            JdbcUtils.free(null, st, conn);
        }
    }

}
```

然后是 Servic 层：

```
package com.weijia.domain;

public class UserService {

    private UserDao userDao;

    public UserService(){
        //通过工厂实例化 UserDao 对象
        userDao = DaoFactory.getInstance().createUserDao();
        System.out.println("userDao:"+userDao);
    }

    /**
     * 注册用户
     * @param user
     */
    public void regist(User user){
        if(user == null){
            System.out.println("注册信息无效!!");
```

```
    }else{
        userDao.addUser(user);
    }

}

/**
 * 查询用户
 * @param userId
 * @return
 */
public User query(int userId){
    User user = userDao.getUserById(userId);
    if(user == null){
        System.out.println("查询结果为空!!");
    }else{
        System.out.println(user.getId()+"\t"+user.getName()+"\t"+user.getBirthday()+"\t"+user.getMoney());
    }
    return userDao.getUserById(userId);
}

/**
 * 更新用户
 * @param user
 */
public void update(User user){
    if(user.getId()<=0){
        System.out.println("用户 id 无效,无法更新");
    }else{
        userDao.update(user);
    }
}

/**
 * 删除用户
 * @param user
 */
public void delete(User user){
    if(user.getId()<=0){
        System.out.println("用户 id 无效,无法删除!!");
    }else{
        userDao.delete(user);
    }
}
```

```
}
```

这里我们还需要额外的两个类：

一个是异常类，因为我们需要自定义我们自己的一个异常，这样方便进行捕获：

```
package com.weijia.domain;

public class DaoException extends RuntimeException{

    private static final long serialVersionUID = 1L;

    public DaoException(){

    }

    public DaoException(Exception e){
        super(e);
    }

    public DaoException(String msg){
        super(msg);
    }

    public DaoException(String msg,Exception e){
        super(msg,e);
    }

}
```

同时，我们这里面采用工厂模式进行实例化 UserDao 对象：

```
package com.weijia.domain;

import java.io.FileInputStream;
import java.util.Properties;

public class DaoFactory {
    /**
     * 单例模式
     */
    private static UserDao userDao = null;
    private static DaoFactory instance = new DaoFactory();
```

```
    private DaoFactory(){
        /**
         * 通过读取属性文件来动态的加载 Dao 层类
         */
        Properties prop = new Properties();
        try{
            FileInputStream fis = new FileInputStream("src/com/weijia/domain/daoconfig.properties");
            prop.load(fis);
            String className = prop.getProperty("userDaoClass");
            Class<?> clazz = Class.forName(className);
            userDao = (UserDao)clazz.newInstance();
            fis.close();
        }catch(Throwable e){
            throw new ExceptionInInitializerError(e);
        }
    }

    public static DaoFactory getInstance(){
        return instance;
    }

    public UserDao createUserDao(){
        return userDao;
    }

}
```

这里面是读取 properties 文件，然后去读取类名进行加载，这种方式是很灵活的

```
package com.weijia.domain;

import java.util.Date;

public class TestDemo {

    public static void main(String[] args) throws Exception{
        UserService userService = new UserService();
        System.out.println("添加用户:");
        userService.regist(new User(1,"jiangwei",new Date(System.currentTimeMillis()),300));
    }

}
```

这里我们看到其实这些操作真的很简单，就是按照那样的几个步骤来操作即可，同时我们还需要将结构进行分层，以便管理，我们这里面测试的时候，撇开了创建数据库的一个环节，至于那个环节，也是不难的，可以从网上搜索一下即可。

3.3 Statement 中的 sql 注入的问题

接着来看一下关于我们上面的例子中使用了 Statement 进行操作的，其实这里面是存在一个问题的，就是会有 sql 注入的问题，我们先来看一下这个问题：

查询学生信息：

```
/**
    * 使用 Statement 读取数据
    * @param name
    * @throws SQLException
    */
   static void read(String name) throws SQLException{
      Connection conn = null;
      Statement st = null;
      ResultSet rs = null;
      try {
         conn = JdbcUtils.getConnection();
         //创建语句
         st = conn.createStatement();
         //执行语句(不建议使用*)
         String sql = "select id,name from user where name='"+name+"'";
         rs = st.executeQuery(sql);
         //根据列名取数据
         while(rs.next()){
            System.out.println(rs.getObject("id") + "\t" + rs.getObject("name") + "\t");
         }
      }catch(SQLException e){
         e.printStackTrace();
      }catch(Exception e){
         e.printStackTrace();
      }finally{
         JdbcUtils.free(rs, st, conn);
      }
   }
```

我们使用代码测试一下：

```
read("'or 1 or'");
```

我们运行会发现，将查询出所有的学生的记录，这个是什么原因呢？我们不妨将 sql 打印一下会发现：

```
select id,name from user where name='or 1 or'
```

因为 sql 语句中把 1 认为是 true,又因为是或的关系，所以将所有的学生的信息查询出来了，这个就是 sql 注入，因为 Statement 会把传递进来的参数进行一下转化操作，用引号包含一下，所以会出现这个问题，那么我们该怎么解决呢？有的同学说我们可以添加一句过滤的代码，将传递的参数取出单引号，这个方法是可行的的，但是这个只能解决那些使用单引号的数据库，可能有的数据库使用的是双引号包含内容，那就不行了，所以应该想一个全套的方法，那么这里我们就是用一个叫做：PreparedStatement 类，这个类是 Statement类的子类：

我们这里只看这个 sql 注入的问题：

我们将上面读取用户信息的代码改写成 PreparedStatement：

```
/**
    * 使用 PreparedStatement
    * @param name
    * @throws SQLException
    */
   static void readPrepared(String name) throws SQLException{
      Connection conn = null;
      PreparedStatement st = null;
      ResultSet rs = null;
      try{
         conn = JdbcUtils.getConnection();
         //执行语句(不建议使用*)
         String sql = "select id,name from user where name=?";
         //创建语句
         st = conn.prepareStatement(sql);
         st.setString(1, name);
         rs = st.executeQuery();
         //根据列名取数据
         while(rs.next()){
            System.out.println(rs.getObject("id") + "\t" + rs.getObject("name") + "\t");
         }
      }catch(Exception e){


      }
   }
```

之后我们在执行：

```
readPrepared("'or 1 or'");
```

就不会全部查出来了，只会查询空结果，因为表中没有一个学生的名字叫做 'or 1 or'。

3.4 JDBC 中典型数据类型的操作问题

3.4.1 日期

我们在操作日期问题的时候会发现，使用 PreparedStatement 进行参数赋值的时候，有一个方法是：setDate(...)，但是这个方法接收的参数是 sql 中的 Date 类，而不是我们平常使用的 util 中的 Date 类，所以我们要做一次转化，通常我们是这样做的，就是在定义实体类的时候将其日期型的属性定义成 util 中的 Date 类型，在进行数据库操作的时候.

进行一次转换：setDate(x,new Date(birthday.getTime());，这里 birthday 就是一个 util.Date 类型的一个属性，而 new Date 是 sql.Date 类型的，这样转化就可以了，同样我们在读取数据的时候将转化操作反过来即可。

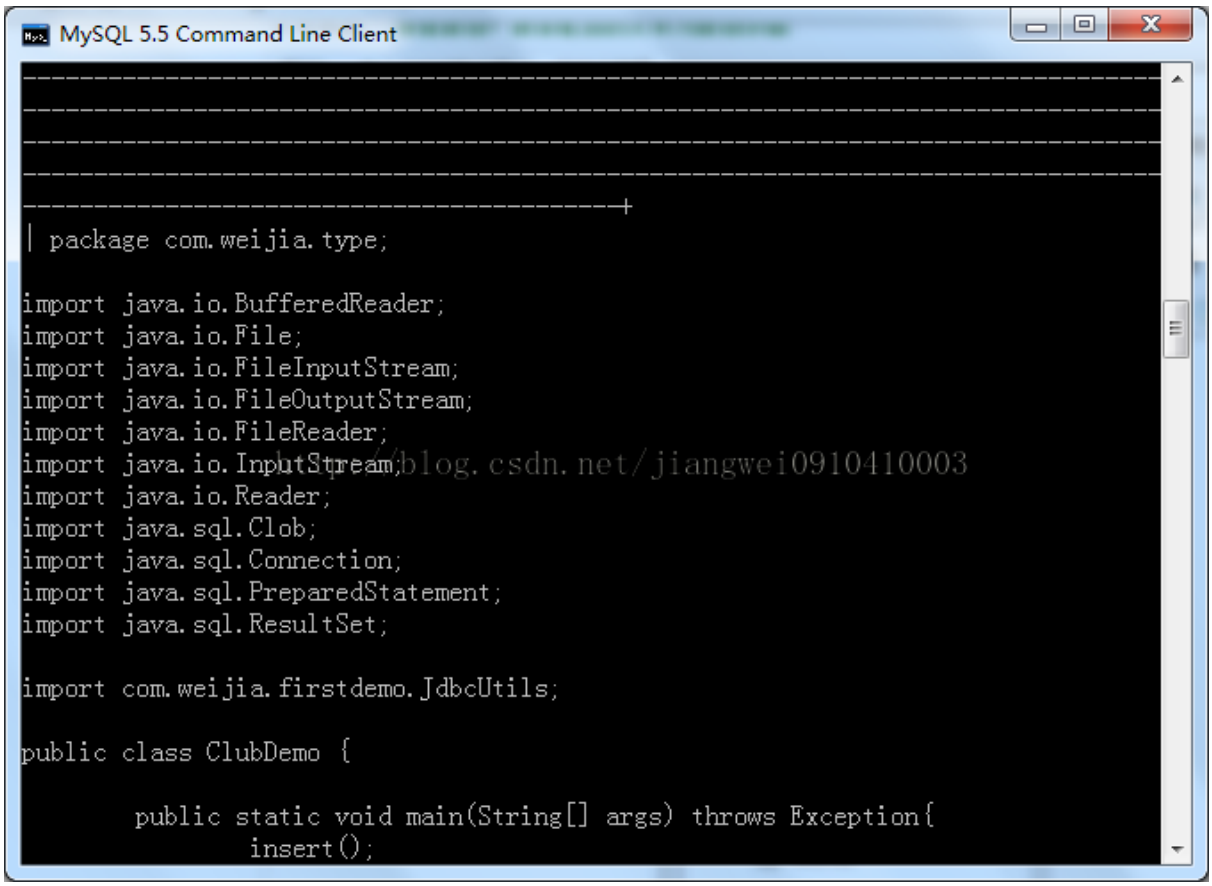
3.4.2 大文本数据

因为有时候我们会存入一些文本内容，因为 varchar 的大小在 mysql 中也是有上线的，所以我们这里要使用 blob 类型了，我们这里来看一下实例：

```
/**
    * 插入大文本
    */
   static void insert(){
      Connection conn = null;
      PreparedStatement ps = null;
      ResultSet rs = null;
      try{
         conn = JdbcUtils.getConnection();
         String sql = "insert into clob_test(bit_text) values(?)";
         ps = conn.prepareStatement(sql);
         File file = new File("src/com/weijia/type/ClubDemo.java");
         Reader reader = new BufferedReader(new FileReader(file));
         //ps.setAsciiStream(1, new FileInputStream(file), (int)file.length());//英文的文档
         ps.setCharacterStream(1, reader, (int)file.length());
         ps.executeUpdate();
         reader.close();
      }catch(Exception e){
         e.printStackTrace();
      }finally{
         JdbcUtils.free(rs,ps,conn);
      }
   }
```

我们将一个 Java 代码文件插入到数据库中

我们查询一下 clob_test 表：



我们看到文件内容存入到库中了。同样我们也可以从表中读取一段文本出来，使用

```
Clob clob = rs.getClob(1);
InputStream is = clob.getAsciiStream();
```

或者读取一个 Reader 也是可以的，这里的 InputStream 和 Reader 是针对不同流，一个字节流，这个不需要关心编码问题的，Reader 是字符流需要关心编码问题。

3.5 JDBC 中的事务

我们当初在学习数据库的时候就了解事务的概念了，事务在数据库中的地位是很重要的。在 JDBC 中默认情况事务是自动提交的，所以我们在进行 CRUD 操作的时候不需要关心开启事务，提交事务，事务回滚的一些操作，那么下面我们就来看一下怎么手动的操作一些事务：

下载我们假定这样的一个场景：

有来两个用户 1 和 2，现在

将用户 1 中的账户的钱减少 10

查询用户 2 中的账户的钱,如果钱少于 300，就增加 10,否则抛出异常

看一下代码：

```
static void test() throws Exception{
    Connection conn = null;
    Statement st = null;
    ResultSet rs = null;
    try{
      conn = JdbcUtils.getConnection();
      /**********事务 START*****************/
      conn.setAutoCommit(false);
      st = conn.createStatement();

      String sql = "update user set money=money-10 where id=1";
      st.executeUpdate(sql);

      sql = "select money from user where id=2";
      rs = st.executeQuery(sql);
      float money = 0.0f;
      if(rs.next()){
        money = rs.getFloat("money");
      }
      if(money>300){
        throw new RuntimeException("已经超过最大值");
      }
      sql = "update user set money=money+10 where id=2";
      st.executeUpdate(sql);
      conn.commit();
      /*****************事务 END*****************/
    }catch(RuntimeException e){

    }finally{
      JdbcUtils.free(rs, st, conn);
    }
  }
```

我们运行测试一下，因为我们这里想让它抛出异常，所以我们将用户 2 中的钱改成大于 300 的，运行一下，结果抛出异常了，但是我们发现了用户 1 中的钱少了 10，但是由于抛出异常，所以后面的代码不执行了，用户 2 中的钱没有变化，那么这样的操作明显不对的，所以我们这时候要解决这个问题，使用事务的回滚操作，在捕获到异常的时候需要做回滚操作：

```
if(conn != null){
  conn.rollback();
}
```

这样即使抛出了异常，这些操作也会进行回滚的，那么用户 1 中的钱就不会少 10 了。

同时上面我们看到，我们是在开始的时候手动的关闭事务的自动提交，然后再手动的提交事务，下面再来看一下事务的保存点的问题。

场景：在上面的基础上，我们添加一个用户 3，同时对用户 1 和用户 3 中的钱进行减少 10，用户 2 的操作不变，但是当抛出异常的时候，我们希望用户 1 的操作还是有效的，用户 3 的操作还原，这时候我们需要将事务回滚到用户 3 的那个点就可以了，这就是事务的保存点的概念，看一下代码：

```
static void test() throws Exception{
    Connection conn = null;
    Statement st = null;
    ResultSet rs = null;
    Savepoint sp = null;
    try{
      conn = JdbcUtils.getConnection();
      /**********事务 START**********/
      conn.setAutoCommit(false);
      st = conn.createStatement();

      String sql = "update user set money=money-10 where id=1";
      st.executeUpdate(sql);
      sp = conn.setSavepoint();//设置回滚点

      sql = "update user set money=money-10 where id=3";
      st.executeUpdate(sql);

      sql = "select money from user where id=2";
      rs = st.executeQuery(sql);
      float money = 0.0f;
      if(rs.next()){
        money = rs.getFloat("money");
      }
      System.out.println("money:"+money);
      if(money>300){
        throw new RuntimeException("已经超过最大值");
      }
      sql = "update user set money=money+10 where id=2";
      st.executeUpdate(sql);
      conn.commit();
      /**********事务 END**********/
    }catch(SQLException e){
      if(conn != null && sp != null){
```

```
            conn.rollback(sp);
            conn.commit();
        }
    }finally{
        JdbcUtils.free(rs, st, conn);
    }
}
```

我们在用户 1 之后设置一个保存点，在异常中只需要回滚到保存点就可以了。

3.6 JDBC 实现批处理功能

我们在前面的例子中会发现，每次都是执行一条语句，然后关闭连接，这样效率可能会很低，如果我们想一次插入几千条数据的话，这时候可以使用批处理的功能，所谓批处理就是将多个执行语句进行捆绑然后去执行，但是效率上并非就一定高，因为我们知道这个数据库连接是 tcp 的，所以在将多个语句捆绑在一起的时候，在传输的过程中也是会进行分包发送的，这个包的大小也不是固定的，这个大小很难掌控的，我们之后经过多次测试之后，才能得到一次批量处理的适宜数量。下面来看一下实例吧：

首先是普通的插入一条数据：

```
static void create() throws Exception{
    //建立一个连接的是很耗时间的
    //执行一个 sql 语句也是很耗时间的
    //优化的措施：批处理
    Connection conn = null;
    PreparedStatement ps = null;
    ResultSet rs = null;
    try{
        conn = JdbcUtils.getConnection();
        String sql = "insert user(name,birthday,money) values(?,?,?)";
        ps = conn.prepareStatement(sql,Statement.RETURN_GENERATED_KEYS);
        ps.setString(1,"jiangwei");
        ps.setDate(2,new Date(System.currentTimeMillis()));
        ps.setFloat(3,400);
        ps.executeUpdate();
    }catch(Exception e){
        e.printStackTrace();
    }finally{
        JdbcUtils.free(rs, ps, conn);
    }
}
```

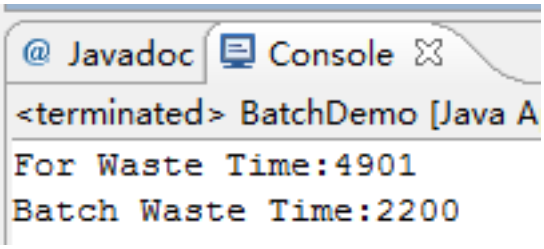
然后是批处理插入 100 条数据：

```
static void createBatch() throws Exception{
    //建立一个连接的是很耗时间的
    //执行一个 sql 语句也是很耗时间的
    //优化的措施：批处理
    Connection conn = null;
    PreparedStatement ps = null;
    ResultSet rs = null;
    try{
      conn = JdbcUtils.getConnection();
      String sql = "insert user(name,birthday,money) values(?,?,?)";
      ps = conn.prepareStatement(sql,Statement.RETURN_GENERATED_KEYS);

      //打包的话容量也不是越大越好，因为可能会内存溢出的，同时网络传输的过程中也是会进行拆包传输的，这个包的大小是不一定的
      //有时候打包的效率不一定就会高，这个和数据库的类型，版本都有关系的，所以我们在实践的过程中需要检验的
      for(int i=0;i<100;i++){
        ps.setString(1,"jiangwei");
        ps.setDate(2,new Date(System.currentTimeMillis()));
        ps.setFloat(3,400);
        //ps.addBatch(sql);
        ps.addBatch();
      }
      ps.executeBatch();
    }catch(Exception e){
      e.printStackTrace();
    }finally{
      JdbcUtils.free(rs, ps, conn);
    }
  }
```

测试代码：

```
public static void main(String[] args) throws Exception{
    long startTime = System.currentTimeMillis();
    for(int i=0;i<100;i++){
      create();
    }
    long endTime = System.currentTimeMillis();
    System.out.println("For Waste Time:"+(endTime-startTime));
    createBatch();
    System.out.println("Batch Waste Time:"+(System.currentTimeMillis()-endTime));
  }
```

我们在控制台中看到他们分别消耗的时间：



我们可以看到这个批处理消耗的时间明显很少。。当然我们在开始的时候也说过了，这个批处理的最适宜的大小要掌控好。

3.7 JDBC 中的滚动结果集和分页技术

我们在前面的例子中可以看到，在处理结果集的时候，我们都是一条一条向后处理的，但是有时候我们需要人为的控制结果集的滚动，比如我们想往前滚动，想直接定位到哪个结果集记录等操作，当然 JDBC 也是提供了一套 Api 让我们来操作的

```
static void test() throws Exception{
    Connection conn = null;
    Statement st = null;
    ResultSet rs = null;
    try{
      conn = JdbcUtils.getConnection();
      //结果集可滚动的
      /**
       * 参数的含义：
       *  ResultSet.RTYPE_FORWORD_ONLY：这是缺省值，只可向前滚动；
         ResultSet.TYPE_SCROLL_INSENSITIVE：双向滚动，但不及时更新，就是如果数据库里的数据修改过，并不在 ResultSet 中反应出来。
         ResultSet.TYPE_SCROLL_SENSITIVE：双向滚动，并及时跟踪数据库的更新,以便更改 ResultSet 中的数据。
         ResultSet.CONCUR_READ_ONLY：这是缺省值，指定不可以更新 ResultSet
         ResultSet.CONCUR_UPDATABLE：指定可以更新 ResultSet
       */
      st = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_READ_ONLY);

      rs = st.executeQuery("select id,name,money,birthday from user");
      //开始的时候这个游标的位置是第一条记录之前的一个位置
      //当执行 rs.next 的时候这个游标的位置就到第一条记录了
      /*while(rs.next()){
        //print result
      }*/
      //上面的代码执行之后，这个游标就到最后一条记录的下一个位置了
```

```
        //所以这里在调用 previous 方法之后，这个游标就回到了最后一条记录中，所以打印了最后一条记录的值
        /*if(rs.previous()){
          System.out.println("id="+rs.getInt("id")+"\tname="+rs.getString("name")+"\tbirthday="+rs.getDate("birthday")+"\tmoney="+rs.getFloat("money"));
        }*/

        //绝对定位到第几行结果集
        //这里传递的参数的下标是从 1 开始的，比如这里查询出来的记录有 3 条，那么这里的参数的范围是:1-3,如果传递的参数不在这个范围内就会报告异常的
        rs.absolute(2);
        System.out.println("id="+rs.getInt("id")+"\tname="+rs.getString("name")+"\tbirthday="+rs.getDate("birthday")+"\tmoney="+rs.getFloat("money"));

        //滚到到第一行的前面(默认的就是这种情况)
        rs.beforeFirst();

        //滚动到最后一行的后面
        rs.afterLast();

        rs.isFirst();//判断是不是在第一行记录
        rs.isLast();//判断是不是在最后一行记录
        rs.isAfterLast();//判断是不是第一行前面的位置
        rs.isBeforeFirst();//判断是不是最后一行的后面的位置

        //以上的 api 可以实现翻页的效果(这个效率很低的，因为是先把数据都查询到内存中，然后再进行分页显示的)

        //效率高的话是直接使用数据库中的分页查询语句：
        //select * from user limit 150,10;

        //以上的 api 实现的分页功能是针对于那些本身不支持分页查询功能的数据库的，如果一个数据库支持分页功能，上面的代码就不能使用的，因为效率是很低的
      }catch(Exception e){
        e.printStackTrace();
      }finally{
        JdbcUtils.free(rs,st,conn);
      }
    }
```

我们看到结果集：rs 有很多方法的，我们一次来看一下：

next()：这个很常用的，就是将结果集向后滚动

previous()：这个方法和 next 是相反的，将结果集向前滚动

absolute(int index)：这个方法是将结果集直接定位到指定的记录上，这个参数是从 1 开始的，不是 0，如果不在指定的范围内的话，会报告异常的

beforeFirst()：这个方法是将结果集直接滚动到第一条记录的前面的位置(默认情况是这样的，所以我们每次在取出数据的时候，需要使用 next 方法，将结果集滚动到第一条记录上)

afterLast()：这个方法是将结果集直接滚动到最后一条记录的后面的位置

isFirst()：判断是不是在第一行记录

isLast()：判断是不是在最后一行记录

isAfterLast()：判断是不是第一行前面的位置

isBeforeFirst()：判断是不是最后一行的后面的位置

当然我们要向实现可滚动的结果集，还要设置一下参数：

```
st = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_READ_ONLY);
```

参数的含义：

ResultSet.RTYPE_FORWORD_ONLY：这是缺省值，只可向前滚动；

ResultSet.TYPE_SCROLL_INSENSITIVE：双向滚动，但不及时更新，就是如果数据库里的数据修改过，并不在 ResultSet 中反应出来。

ResultSet.TYPE_SCROLL_SENSITIVE：双向滚动，并及时跟踪数据库的更新,以便更改 ResultSet 中的数据。

ResultSet.CONCUR_READ_ONLY：这是缺省值，指定不可以更新 ResultSet

ResultSet.CONCUR_UPDATABLE：指定可以更新 ResultSet(这个后面会说到)

同时在这里我们只需要使用 absolute 方法就可以实现分页的功能，因为他可以随便的定位到指定的记录集中，但是这个是在全部将结果集查询处理的基础上来实现的，就是首先将所有符合条件的结果集查询出来放到内存中，然后再就行分页操作，那么这种分页的效率就很低了，所以我们强烈建议在数据库查询数据的时候就进行分页操作，比如 MySql 中使用 limit 关键字进行操作，MSSQL 中使用 top 关键字，Oracle 中使用 number 关键字，但是有的数据库中不支持分页查询操作，所以这时候我们只能使用上面的形式来进行分页了。

3.8 JDBC 中的可更新以及对更新敏感的结果集操作

我们有时候可能有这样的需求，就是在查询出结果集的时候，想对指定的记录进行更新操作，说白了，就是将查询和更新操作放到一起进行，来看一下代码：

```
static void test() throws Exception{
    Connection conn = null;
```

```
    Statement st = null;
    ResultSet rs = null;
    try{
      conn = JdbcUtils.getConnection();
      //第三个字段的含义是，在读取数据的时候(已经返回了结果集到内存中了)，
      //再去修改结果集中的数据，这时候数据库中的数据就可以感知到结果集中的变化了进行修改
      st = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);

      rs = st.executeQuery("select * from user");
      //这种操作是不可取的，因为查询和更新交互在一起，逻辑就乱了，只有在特定的场合中使用
      while(rs.next()){
        //这里我们获取到 name 列的值，如果是 lisi 我们就将结果集中的他的记录中的 money 变成
170，
        //然后再更行行信息，这时候数据库中的这条记录的值也发生变化了,
        //内存中的结果集中的记录的值发生改变了，影响到了数据库中的值
        String name = rs.getString("name");
        if("jiangwei".equals(name)){
          rs.updateFloat("money",170);
          rs.updateRow();
        }
      }
    }catch(Exception e){
      e.printStackTrace();
    }finally{
      JdbcUtils.free(rs,st,conn);
    }
  }
```

我们看到在循环处理结果集的时候，我们将 name 是 jiangwei 的记录的钱修改成 170，并且反映到数据库中

这里一定要记得设置参数：ResultSet.CONCUR_UPDATABLE，不然会报异常的，这个参数的功能就是将更新的操作同步到数据库中的

这里我们是不建议这种做法的，因为将查询和更新的操作放到一起来操作的话，维护是很差的，我们一定要将 CRUD 操作进行分开处理，这里只是介绍一下相关知识，不推荐使用的。

下面来看一下通过反射技术，来实现将结果集填充到指定的实体类中，其实这部分的内容很简单的，直接上代码：

```
/**
   * 使用泛型
   * @param <T>
   * @param sql
   * @param clazz
   * @return
```

```
     * @throws Exception
     */
    static <T> T test(String sql,Class<T> clazz) throws Exception{
        Connection conn = null;
        PreparedStatement ps = null;
        ResultSet rs = null;
        try{
            conn = JdbcUtils.getConnection();
            ps = conn.prepareStatement(sql);
            rs = ps.executeQuery();
            ResultSetMetaData rsmd = rs.getMetaData();
            int count = rsmd.getColumnCount();
            String[] colNames = new String[count];
            for(int i=1;i<=count;i++){
                colNames[i-1] = rsmd.getColumnLabel(i);//使用别名，让列名和 User 中的属性名相同
            }
            T user = clazz.newInstance();
            //使用反射获取 set 方法来进行赋值
            if(rs.next()){
                Method[] ms = user.getClass().getMethods();
                for(int i=0;i<colNames.length;i++){
                    String colName = colNames[i];
                    String methodName = "set" + colName;
                    for(Method method:ms){
                        //通过列名来找到实体类中的属性方法(这里要注意的是 set 方法的格式是:setXxx 首字母是大
写的)
                        //这里直接使用忽视大小写的相等的方法
                        //或者使用上面的重命名来解决这个问题
                        if(methodName.equalsIgnoreCase(method.getName())){
                            method.invoke(user, rs.getObject(colNames[i]));
                        }
                    }
                }
            }
            return user;
        }catch(Exception e){
            e.printStackTrace();
        }finally{
            JdbcUtils.free(rs,ps,conn);
        }
        return null;
    }
```

测试代码：

```
public static void main(String[] args) throws Exception{
```

```
    //User user = test("select * from user",User.class);
    //使用别名来规定列名和属性名相同
    User user = test("select id as Id,name as Name,birthday as Birthday,money as Money from user",User.class);
    System.out.println(user);
  }
```

其实就是使用反射技术将得到实体类中所有属性的 set 方法，然后通过 set 方法进行属性值的填充，这里唯一要注意的问题就是返回的结果集中的字段名称必须要和实体类中的属性名称相同，要做到这一点我们又两种方式：

一种是直接将表中的字段名称和实体类中的属性名称相同

一种是使用别名的方式来操作，将别名设置的和实体类中的属性名称相同

其实我们会发现，后面说到的 Hibernate 框架就是采用这种机制的

3.9 元数据

3.9.1 数据库的元数据

就是数据库的相关信息，如数据库的版本号，驱动名称，是否支持事务操作等信息，JDBC 提供了接口让我们可以获取这些信息的:

```
static void test() throws Exception{
    Connection conn = JdbcUtils.getConnection();
    //这些信息对于那些框架的编写就很有用了，因为框架是要兼容各个数据库类型的,如 Hibernate 中有一个方言设置
    //如果没有设置的话，他就会自己使用以下的 api 进行查找是那个数据库
    DatabaseMetaData metaData = conn.getMetaData();
    System.out.println("databaseName:"+metaData.getDatabaseProductName());
    System.out.println("driverName:"+metaData.getDriverName());
    System.out.println("isSupportBatch:"+metaData.supportsBatchUpdates());
    System.out.println("isSupportTransaction:"+metaData.supportsTransactions());
  }
```

这些信息对于我们使用人员来说可能没有太大的用处，但是对于开发框架的人来说用处很大的，比如 Hibernate 框架，他要做到兼容所有的数据库特性的话，必须要将不同特性统一起来，所以他肯定要获取数据库的元数据信息的，后面会说到 Hibernate 中有一个配置叫做：方言，这个就是用来设置数据库名称的。

3.9.2 查询参数的元数据信息

当我们在使用 PreparedStatement 来进行参数填充的时候，我们想知道参数的一些信息，

```
static void test(String sql,Object[] params) throws Exception{
    Connection conn = JdbcUtils.getConnection();
```

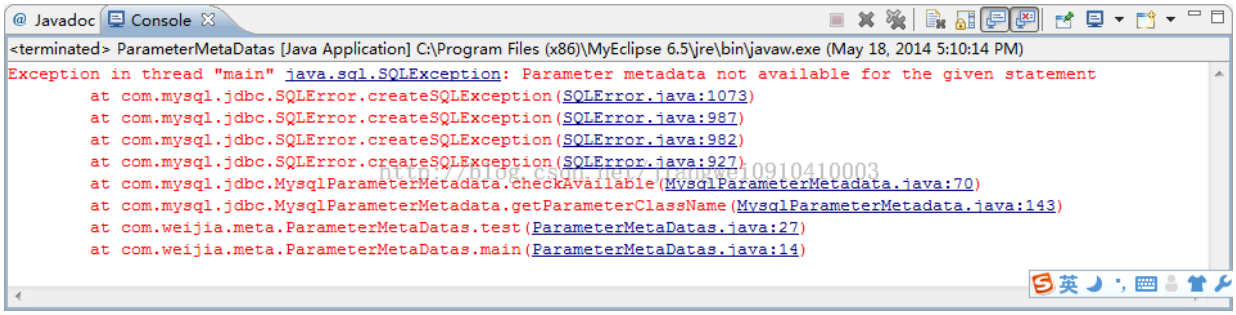
```
    //参数的元数据信息:Statement 是没有参数的元数据信息的(因为 Statement 不支持?),查看源代码，返回的都是 varchar
    PreparedStatement ps = conn.prepareStatement(sql);
    //必须在连接数据库中的时候添加这个参数 generateSimpleParameterMetadata=true
    //不然是获取不到参数的，而且会报异常
    ParameterMetaData pMetaData= ps.getParameterMetaData();
    int count = pMetaData.getParameterCount();
    for(int i=1;i<=count;i++){
      //因为 mysql 没有去查询库，所以不能根据查询的字段就能获取字段的类型，所有都返回 varchar
      System.out.println(pMetaData.getParameterClassName(i));
      System.out.println(pMetaData.getParameterType(i));
      System.out.println(pMetaData.getParameterTypeName(i));
      //假定我们传入的参数的顺序和 sql 语句中的占位符的顺序一样的
      ps.setObject(i,params[i-1]);
    }
    ps.executeQuery();
  }
```

测试代码：

```
public static void main(String[] args) throws Exception{
    String sql = "select name,birthday,money from user where name=?";
    Object[] params = new Object[]{"jiangwei"};
    test(sql,params);
  }
```

我们知道 Statement 是不支持参数填充的，所以不可能获取到参数的元数据信息的

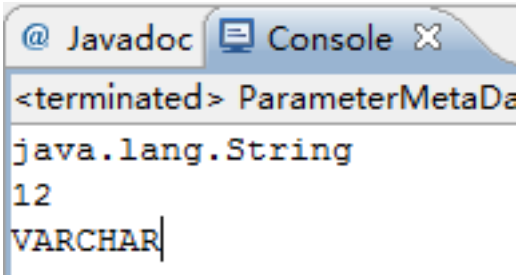
我们运行测试代码，会看到如下异常信息：



这时候我们就要注意了，如果想获取到元数据信息的话，我们还需要在连接数据的 url 后面添加一个参数：

```
generateSimpleParameterMetadata=true
```

添加完之后，我们运行结果如下：



我们看到可以获取参数在 Java 中的类型，12 代表的是 sql 包中的类型：java.sql.Types.VARCHAR，这个字段是个整型值，值就是 12，他对应到数据库中 varchar 类型的

这里要注意一点：有时候我们会发现这里获取到数据库中的类型是错误的，比如这里我们如果将数据库中的 name 字段的类型修改成 char 类型的，这里获取到的还是 varchar，这一点想一想也是对的，你想想这个是获取查询参数的信息，我们还没有进行查询操作的，系统不可能那么智能的获取到数据库中准确的字段的类型的，所以他这里就做了一个大致的对应关系，将 Java 中的类型和数据库中的类型对应起来的，因为数据库中 char 和 varchar 都是字符串的。所以我们不能相信这里得到的数据库中字段的类型的，需要通过结果集中的元数据类型。

3.9.3 结果集中元数据信息

就是查询结果的一般信息，比如字段的相关信息

我们在上面看到要想获取数据库中字段的真实类型的话，只有先进行查询操作才可以，在这里我们就可以获取到正确的类型了，上代码：

```
static void test(String sql) throws Exception{
    Connection conn = null;
    PreparedStatement ps = null;
    ResultSet rs = null;
    try{
      conn = JdbcUtils.getConnection();
      ps = conn.prepareStatement(sql);
      rs = ps.executeQuery();
      ResultSetMetaData rsmd = rs.getMetaData();
      int count = rsmd.getColumnCount();
      String[] colNames = new String[count];
      for(int i=1;i<=count;i++){
        //这里是可以获取到真实的类型的，因为这个是已经从数据库中查询了
        System.out.println(rsmd.getColumnClassName(i));
        System.out.println(rsmd.getColumnName(i));
        System.out.println(rsmd.getColumnType(i));
        System.out.println(rsmd.getColumnLabel(i));//列的别名:select name as n from user;,有别名的话就返回的是别名，而不是原始的列名了
```

```
            colNames[i-1] = rsmd.getColumnName(i);
        }
        //将结果构建一个 Map，列名是 key,列的值是 value
        Map<String,Object> data = null;
        //假设查询的数据只有一条，如果是多条的话我们可以定义一个 List<Map<...这样的结构
        if(rs.next()){
            data = new HashMap<String,Object>();
            for(int i=0;i<colNames.length;i++){
                data.put(colNames[i], rs.getObject(colNames[i]));
            }
        }
    }catch(Exception e){
        e.printStackTrace();
    }finally{
        JdbcUtils.free(rs,ps,conn);
    }
  }
```

我们这里可以获取到结果集中字段的总数 count，以及字段的类型，名称，别名等信息，同时我们这里还穿插了一段代码，就是将结果集封装成一个 HashMap 结构，字段名做 key,字段值做 value。

3.10 JDBC 中的数据源

首先我们要知道什么是数据源，为什么要有数据源，我们从上面的例子看到，我们每次执行操作的时候，都是打开连接，关闭连接，这个连接的建立和关闭是很耗资源和时间的，所以我们就在想一个策略怎么才能优化呢？所以数据源的概念就出来的，数据源就是用来管理连接的一个池子，使用高效的算法进行调度，这样在执行操作的时候是很方便的，为了容易理解数据源的相关概念，我们自己编写一个数据源：

```
package com.weijia.datasource;

import java.io.PrintWriter;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.LinkedList;

import javax.sql.DataSource;

/**
 * 大部分时间都是浪费在数据库连接这一块
```

```
 * 这个类是我们自己编写的一个数据源
 * @author weijiang204321
 *
 */
public class MyDataSource implements DataSource{

   private static String user = "root";
   private static String password = "123456";
   private static String dbName = "test";
   private static  String url = "jdbc:mysql://localhost:3306/"+dbName+"?user="+user+"&password="+password+"
&useUnicode=true&characterEncoding=gb2312";

   private static int initCount = 5;//初始化的连接数
   private static int maxCount = 10;//最大连接数
   private static int currentCount = 0;//当前的连接数
   //可能频繁的取出连接和删除连接，所以用 LinkedList
   private LinkedList<Connection> connectionsPool = new LinkedList<Connection>();

   public MyDataSource(){
      try{
         for(int i=0;i<initCount;i++){
            this.connectionsPool.addLast(createConnection());
            currentCount++;
         }
      }catch(Exception e){
         throw new ExceptionInInitializerError(e);
      }
   }

   public Connection getConnection() throws SQLException{
      //也有可能获取不到连接，而且这个方法也是可能被多线程访问的
      synchronized(connectionsPool){
         if(connectionsPool.size() > 0){
            return this.connectionsPool.removeFirst();
         }
         if(currentCount < maxCount){
            currentCount++;
            return createConnection();
         }

         //在这里可以让当前线程等待，抛出异常，返回 null 都是可以的，要视情况而定
         throw new SQLException("已经没有连接了");
```

```
        //不能无限制的创建连接的，因为这样的话对数据库的压力很大，连接越多，最后数据库的运行速度就会变得很慢了（很硬件相关）
        //如果内存够大，cpu 给力的话，数据库可以建立的连接数也会增加的
      }
    }

    public void free(Connection conn) throws SQLException{
      this.connectionsPool.addLast(conn);
    }

    private Connection createConnection() throws SQLException{
      return DriverManager.getConnection(url);
    }

    public Connection getConnection(String username, String password)throws SQLException {
      return null;
    }

    public PrintWriter getLogWriter() throws SQLException {
      return null;
    }

    public int getLoginTimeout() throws SQLException {
      return 0;
    }

    public void setLogWriter(PrintWriter arg0) throws SQLException {

    }

    public void setLoginTimeout(int arg0) throws SQLException {

    }

}
```

然后修改一下 JdbcUtils 中的代码：

```
package com.weijia.firstdemo;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import javax.sql.DataSource;
```

```
import com.weijia.datasource.MyDataSource;

public class JdbcUtils {

    private static String user = "root";
    private static String password = "123456";
    private static String dbName = "test";
    private static  String url = "jdbc:mysql://localhost:3306/"+dbName+"?user="+user+"&password="+password+"
&useUnicode=true&characterEncoding=gb2312&generateSimpleParameterMetadata=true";

    private static MyDataSource dataSource = null;

    /**
     * 加载驱动
     */
    static{
        try{
            Class.forName("com.mysql.jdbc.Driver");
            dataSource = new MyDataSource();//初始化数据源
        }catch(Exception e){
            System.out.println("Exception:"+e.getMessage()+"");
            throw new ExceptionInInitializerError(e);
        }
    }

    private JdbcUtils(){
    }

    /**
     * 获取连接
     * @return
     * @throws SQLException
     */
    public static Connection getConnection() throws SQLException{
        return dataSource.getConnection();
    }

    public static DataSource getDataSource(){
        return dataSource;
    }

    /**
     * 释放资源
```

```
     * @param rs
     * @param st
     * @param conn
     */
    public static void free(ResultSet rs,Statement st,Connection conn){
        try{
            if(rs != null){
                rs.close();
            }
        }catch(SQLException e){
            e.printStackTrace();
        }finally{
            try{
                if(st != null){
                    st.close();
                }
            }catch(SQLException e){
                e.printStackTrace();
            }finally{
                try{
                    dataSource.free(conn);
                }catch(SQLException e){
                    e.printStackTrace();
                }
            }
        }

    }

}
```

我们看到，在我们自定义的数据源中，主要有这么几个变量：

初始化连接数，最大连接数，当前的连接数，连接池(因为我们可能需要频繁的添加连接和删除连接所以使用 LinkedList,因为这个 list 是链表结构的，增加和删除效率高)

主要流程是：初始化数据源的时候，初始化一定量的连接放到池子中，当用户使用 getConnection()方法取出连接的时候，我们会判断这个连接池中还有没有连接了，有就直接取出第一个连接返回，没有的话，我们在判断当前的连接数有没有超过最大连接数，超过的话，就抛出一个异常(其实这里还可以选择等待其他连接的释放，这个具体实现是很麻烦的)，没有超过的话，就创建连接，并且将其放入池子中。

我们自定义的数据源是实现了 JDBC 中的 DataSource 接口的，这个接口很重要的，后面我们会说到 apache 的数据源都是要实现这个接口的，这个接口统一了数据源的标准，这个接口中有很多实现的，所以看到我们的数据源类中有很多没必要的方法，但是那个方法

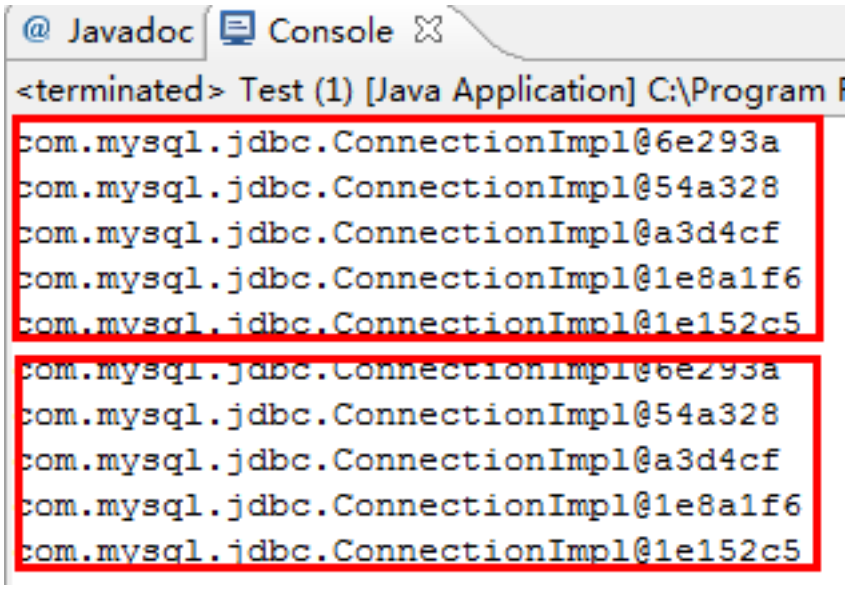
都是要实现的，最重要的就是要实现 getConnection 方法，其他的实现都只需要调用 super.XXX 就可以了。

在 JdbcUtils 类中我们也是需要修改的，首先我们要在静态代码块中初始化我们的数据源，在 getConnection 方法中调用数据源的 getConnection 方法，在 free 方法中调用数据源的 free 方法即可。

看一下测试类：

```
package com.weijia.datasource;

import java.sql.Connection;

import com.weijia.firstdemo.JdbcUtils;
 public class Test {
      public static void main(String[] args) throws Exception{
      for(int i=0;i<10;i++){
         Connection conn = JdbcUtils.getConnection();
         System.out.println(conn);
         JdbcUtils.free(null, null, conn);
      }
   }

}
```

运行结果：



我们可以看到，我们在测试代码中申请了 10 个连接，从结果上可以看出前五个是不同的连接，后五个连接和前五个是一样的，这是因为我们在释放连接的时候就是 free 方法中，是将连接重新放到池子中的，上面显示的是五个，是因为我们初始化的连接数是 5 个，当第一个连接释放的时候这个连接其实已经放到了池子的第六个位置，以此类推。

下面我们继续来看下个问题，我们在上面的数据源中可以看到，我们定义了一个 free 方法来释放连接的，然后在 JdbcUtils 中调用这个方法即可，但是这个貌似不太符合我们的使用习惯，因为之前我们看到我们释放连接的时候都是使用 close 方法的，所以这里面我们在修改一下，至于怎么修改呢？

首先我们知道那个 close 方法是 JDBC 中的 Connection 接口中的，所有自定义的连接都是需要实现这个接口的，那么我们如果我们想让我们 free 中的逻辑放到 close 中的话，就需要实现这个接口了，我们可以看到

```
DriverManager.getConnection(url)
```

通过这种方式获取到的 Connection 也是 mysql 中实现了 Connection 的接口的，那么现在我们可能需要自定一个我们自己的连接，然后实现 Connection 接口，将 free 方法中的逻辑搬到 close 方法中，同时我们还要在连接类中保持一个 mysql 中的连接对象，这里面的逻辑有点不好理解，先看代码：

```
package com.weijia.datasource;

import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.SQLWarning;
import java.sql.Savepoint;
import java.sql.Statement;
import java.util.Map;

public class MyConnection implements Connection{

   //组合方式：静态代理
   private Connection realConnection;
   private MyDataSource2 dataSource;
   //当前连接的使用的次数
   private int maxUseCount = 5;
   private int currentUseCount = 0;

   public MyConnection(Connection conn,MyDataSource2 dataSource){
      this.realConnection = conn;
      this.dataSource = dataSource;
   }

   public void close() throws SQLException {
      this.currentUseCount++;
      if(this.currentUseCount < this.maxUseCount){
```

```
            this.dataSource.free(this);
        }else{
            dataSource.currentCount--;
            this.realConnection.close();
        }
    }

    public void clearWarnings() throws SQLException {
        this.realConnection.clearWarnings();
    }

    public void commit() throws SQLException {
        this.realConnection.commit();
    }

    public Statement createStatement() throws SQLException {
        return this.realConnection.createStatement();
    }

    public Statement createStatement(int resultSetType, int resultSetConcurrency)throws SQLException {
        return this.realConnection.createStatement(resultSetType, resultSetConcurrency);
    }

    public Statement createStatement(int resultSetType,int resultSetConcurrency, int resultSetHoldability)throws S
QLException {
        return this.realConnection.createStatement(resultSetType, resultSetConcurrency, resultSetHoldability);
    }

    public boolean getAutoCommit() throws SQLException {
        return this.realConnection.getAutoCommit();
    }

    public String getCatalog() throws SQLException {
        return this.realConnection.getCatalog();
    }

    public int getHoldability() throws SQLException {
        return this.realConnection.getHoldability();
    }

    public DatabaseMetaData getMetaData() throws SQLException {
        // TODO Auto-generated method stub
        return null;
    }
```

```
public int getTransactionIsolation() throws SQLException {
    // TODO Auto-generated method stub
    return 0;
}

public Map<String, Class<?>> getTypeMap() throws SQLException {
    // TODO Auto-generated method stub
    return null;
}

public SQLWarning getWarnings() throws SQLException {
    // TODO Auto-generated method stub
    return null;
}

public boolean isClosed() throws SQLException {
    // TODO Auto-generated method stub
    return false;
}

public boolean isReadOnly() throws SQLException {
    // TODO Auto-generated method stub
    return false;
}

public String nativeSQL(String sql) throws SQLException {
    // TODO Auto-generated method stub
    return null;
}

public CallableStatement prepareCall(String sql) throws SQLException {
    // TODO Auto-generated method stub
    return null;
}

public CallableStatement prepareCall(String sql, int resultSetType,
        int resultSetConcurrency) throws SQLException {
    // TODO Auto-generated method stub
    return null;
}

public CallableStatement prepareCall(String sql, int resultSetType,
        int resultSetConcurrency, int resultSetHoldability)
```

```
        throws SQLException {
    // TODO Auto-generated method stub
    return null;
}

public PreparedStatement prepareStatement(String sql) throws SQLException {
    // TODO Auto-generated method stub
    return null;
}

public PreparedStatement prepareStatement(String sql, int autoGeneratedKeys)
        throws SQLException {
    // TODO Auto-generated method stub
    return null;
}

public PreparedStatement prepareStatement(String sql, int[] columnIndexes)
        throws SQLException {
    // TODO Auto-generated method stub
    return null;
}

public PreparedStatement prepareStatement(String sql, String[] columnNames)
        throws SQLException {
    // TODO Auto-generated method stub
    return null;
}

public PreparedStatement prepareStatement(String sql, int resultSetType,
        int resultSetConcurrency) throws SQLException {
    // TODO Auto-generated method stub
    return null;
}

public PreparedStatement prepareStatement(String sql, int resultSetType,
        int resultSetConcurrency, int resultSetHoldability)
        throws SQLException {
    // TODO Auto-generated method stub
    return null;
}

public void releaseSavepoint(Savepoint savepoint) throws SQLException {
    // TODO Auto-generated method stub
```

```
}

public void rollback() throws SQLException {
    // TODO Auto-generated method stub

}

public void rollback(Savepoint savepoint) throws SQLException {
    // TODO Auto-generated method stub

}

public void setAutoCommit(boolean autoCommit) throws SQLException {
    // TODO Auto-generated method stub

}

public void setCatalog(String catalog) throws SQLException {
    // TODO Auto-generated method stub

}

public void setHoldability(int holdability) throws SQLException {
    // TODO Auto-generated method stub

}

public void setReadOnly(boolean readOnly) throws SQLException {
    // TODO Auto-generated method stub

}

public Savepoint setSavepoint() throws SQLException {
    // TODO Auto-generated method stub
    return null;
}

public Savepoint setSavepoint(String name) throws SQLException {
    // TODO Auto-generated method stub
    return null;
}

public void setTransactionIsolation(int level) throws SQLException {
    // TODO Auto-generated method stub
```

```
    }

    public void setTypeMap(Map<String, Class<?>> map) throws SQLException {
        // TODO Auto-generated method stub

    }

}
```

首先看到了这个类中有很多恶心的代码，那些方法都是 Connection 接口中的，我们这里只需要实现 close 方法就可以了，其他的方法中可以添加：

```
return this.realConnection.XXX
```

我们看到会在类中保留一个 Connection 对象，这个对象就是真实的连接对象，即我们使用

```
DriverManager.getConnection(url)
```

这种方法获取到的，因为我们要在 close 方法中使用到这个真实的连接

我们看一下 close 方法吧：

```
public void close() throws SQLException {
        this.currentUseCount++;
        if(this.currentUseCount < this.maxUseCount){
            this.dataSource.free(this);
        }else{
            dataSource.currentCount--;
            this.realConnection.close();
        }
    }
```

首先看到我们定义了一个类变量：currentUseCount 用来表示当前连接的使用次数，同时还有一个类变量就是 maxUseCount 表示当前连接的最大使用次数，我们看一下 close 方法的逻辑：

首先当用户调用 close 方法的时候当前连接的使用数就加一，这里有些同学可能不能理解，我们想想上面还记得我们释放连接的时候是怎么做的，是将这个连接重新放到池子中，所以这个连接又被用了一次，所以这里面是加一，当这个连接的当前使用次数没有超过他的最大使用次数的话，就还把他放到池子中(就是数据源中的 free 方法，这个方法中传递的参数是我们自定义的连接对象，因为我们不是真的需要关闭这个连接的)，如果使用次数超过了最大使用次数的话，我们就将这个连接真正的释放关闭了，同时需要将数据源中当前的连接数减去一，这里我们是调用真实连接的关闭方法的，所以我们需要在我们自定义的连接中保持一个真实连接的对象，其实我们采用的是组合的方法，在一个要想调用另外类中的方法，我们需要在本类中维持一个他的对象，然后进行调用他特定的方法，这种方式也是一种设计模式叫做：静态代理，相当于我们本类是另外一个类的代理。

同时我们需要在构造函数中传递一个数据源对象进来的，当然我们这时候需要在之前的数据源中修改一下，这里修改很简单的，只需要修改数据源中的 createConnection 方法就可以了：

```
private Connection createConnection() throws SQLException{
    Connection realConn = DriverManager.getConnection(url);
    MyConnection myConnection = new MyConnection(realConn,this);
    return myConnection;
  }
```

我们返回的其实是我们自己的定义的连接，这个连接其实也是真实连接的一个代理对象。这样我们在 JdbcUtils 中的 free 方法中直接调用：

```
conn.close();
```

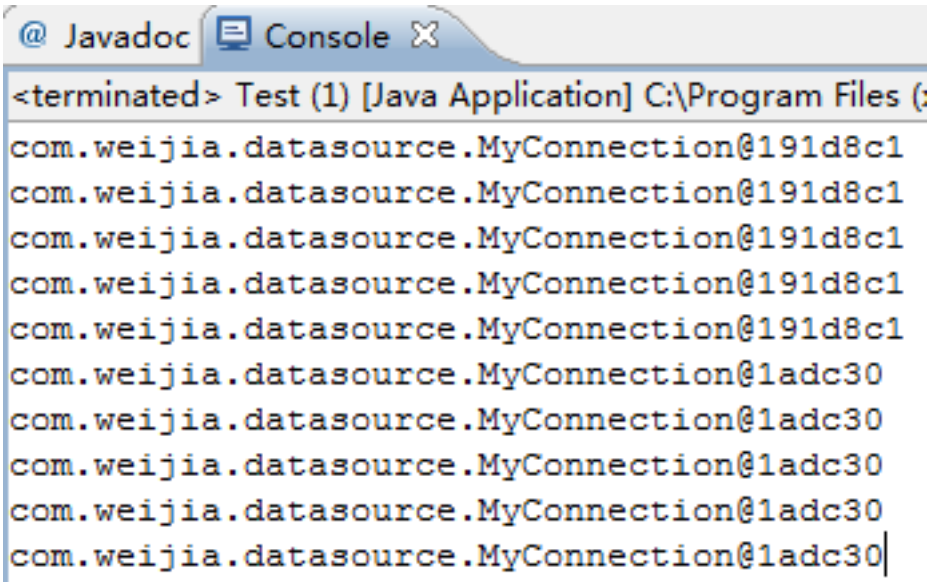
而不需要调用：

```
dataSource.free(conn);
```

这样的释放方式就和我们之前普通连接的释放方式是一样的。其实我们上面做的这么多的操作就是为了这个，想让用户能够还是直接调用 conn.close 方法就可以释放连接，我们运行一下之前的测试类：

```
package com.weijia.datasource;

import java.sql.Connection;

import com.weijia.firstdemo.JdbcUtils;

public class Test {

  public static void main(String[] args) throws Exception{
    for(int i=0;i<10;i++){
      Connection conn = JdbcUtils.getConnection();
      System.out.println(conn);
      JdbcUtils.free(null, null, conn);
    }
  }

}
```

运行结果如下：



我们看到前五个用的是同一个连接对象，这个原因就是我们在我们自定义的连接 MyConnection 类中使用了当前连接的最大使用次数是 5 次

我们看到在定义我们自己的连接类的时候，需要实现 Connection 接口，这个接口中需要实现的方法很多，其实我们只需要一个 close 方法就可以了，这时候我们还可以将我们的代码在修改一下，下面是我们修改之后的自定义连接类：

```
package com.weijia.datasource;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
import java.sql.Connection;

public class MyConnectionHandler implements InvocationHandler{

   private Connection realConnection = null;
   private Connection warpedConnection = null;
   private MyDataSource2 dataSource = null;

   //当前连接的使用的次数
   private int maxUseCount = 5;
   private int currentUseCount = 0;

   public MyConnectionHandler(MyDataSource2 dataSource){
      this.dataSource = dataSource;
   }

   public Connection bind(Connection conn){
      this.realConnection = conn;
```

```
        warpedConnection = (Connection)Proxy.newProxyInstance(this.getClass().getClassLoader(),new Class[]{Co
nnection.class},this);
        return warpedConnection;
    }

    public Object invoke(Object proxy, Method method, Object[] args)throws Throwable {
        if("close".equals(method.getName())){
            this.currentUseCount++;
            if(this.currentUseCount < this.maxUseCount){
                this.dataSource.free(warpedConnection);
            }else{
                dataSource.currentCount--;
                this.realConnection.close();
            }
        }
        return method.invoke(this.realConnection, args);
    }

}
```

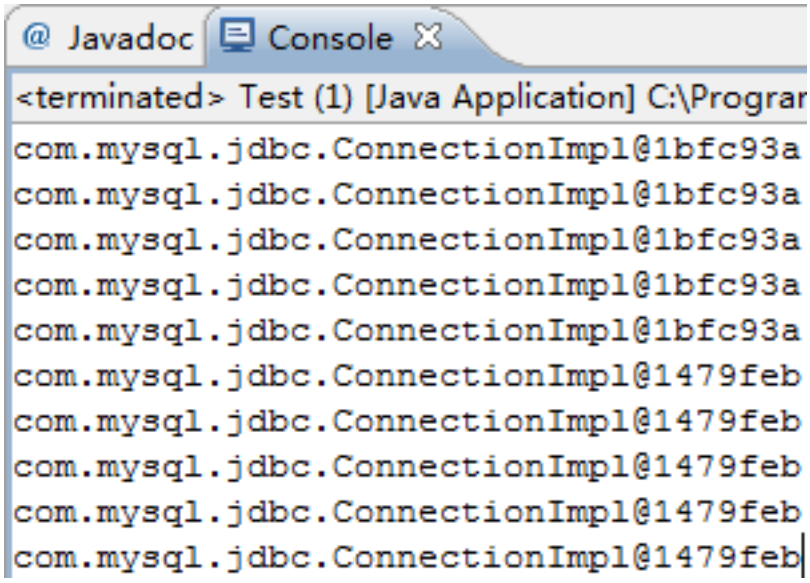
这里我们看到了并没有实现 Connection 接口了，而且代码也是很简洁的，其实这个就是动态代理的模式，我们通过 bind 方法传递进来一个真实的连接对象，然后使用 Proxy 类实例化一个代理对象，newProxyInstance 方法的参数说明：

第一个：需要代理对象的类加载器

第二个：需要代理对象实现的接口

第三个：InvocationHandler 回调接口，我们主要的工具都是实现这个接口中的 invoke 方法

然后我们在 invoke 方法中拦截 close 方法即可，将之前的 close 方法中的逻辑搬到这里就可以了。我们使用上面的测试代码运行如下：



这里我们就看到了使用动态代理很简单的，但是有一个限制，就是代理对象必须要实现一个接口，这里正好是 Connection 接口，他比静态代理优雅了很多的，后面我们在说到 Spring 的时候还会说到这个动态代理模式的

好了，上面我们就可以看到我们自己定义了一个数据源，连接，这样对我们后面的操作优化了很多。

下面我们在来看一下 apache 的数据源 DataSource,其实这个数据源大体上和我们上面设计的以一样的，只是他做了更优化，更好。

首先我们导入需要的 jar 包：

然后我们定义一个 dbcpconfig.properties 文件，用于配置数据源的相关信息：

```
#连接设置
driverClassName=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/test
username=root
password=123456

#<!-- 初始化连接 -->
initialSize=10

#最大连接数量
maxActive=50

#<!-- 最大空闲连接 -->
maxIdle=20

#<!-- 最小空闲连接 -->
minIdle=5

#<!-- 超时等待时间以毫秒为单位 6000 毫秒/1000 等于 60 秒 -->
maxWait=60000



#JDBC 驱动建立连接时附带的连接属性属性的格式必须为这样：[属性名=property;]
```

```
#注意："user" 与 "password" 两个属性会被明确地传递，因此这里不需要包含他们。
connectionProperties=useUnicode=true;characterEncoding=gbk;generateSimpleParameterMetadata=true

#指定由连接池所创建的连接的自动提交（auto-commit）状态。
defaultAutoCommit=true

#driver default 指定由连接池所创建的连接的只读（read-only）状态。
#如果没有设置该值，则“setReadOnly”方法将不被调用。（某些驱动并不支持只读模式，如：
Informix）
defaultReadOnly=

#driver default 指定由连接池所创建的连接的事务级别（TransactionIsolation）。
#可用值为下列之一：（详情可见 javadoc。）
NONE,READ_UNCOMMITTED, READ_COMMITTED, REPEATABLE_READ, SERIALIZABLE
defaultTransactionIsolation=READ_UNCOMMITTED
```

从这些配置上我们可以看到前面的几个参数的含义就是我们自定义数据源中的使用到的，这里还有一个参数是 maxWait 是超时，这个就是我们在获取连接的时候，当连接数超过最大连接数的时候，需要等待的时间，在前面我们自己定义的数据源中我们是采用抛出异常的问题来解决的，这里我们看到 apache 是采用线程等待的方式来解决的。

我们在代码里面修改的东西也是很少的，在 JdbcUtils 中的静态代码块中使用 apache 的数据源即可：

```
//使用 Apache 的 DBCP 数据源
Properties prop = new Properties();
prop.load(JdbcUtils.class.getClassLoader().getResourceAsStream("dbcpconfig.properties"));
dataSource = BasicDataSourceFactory.createDataSource(prop);
```

apache 中的连接也是经过改装的，我们直接调用 conn.close 方法即可，和我们上面实现的是一样的。

3.11 JDBC 中 CRUD 的模板模式

我们从前面的例子中可以看到，我们在操作 CRUD 的时候，返现有很多重复的代码，比如现在一个 UserDao 来操作查询操作，写了一段查询代码，然后有一个 ProductDao 也来操作查询操作，也写了一段查询代码，其实我们会发现这两个查询代码中有很多是重复的，这时候我们就想了，能不能够进行代码的优化，我们想到了模板模式，就是将相同的代码提取出来放到父类中做，不同的代码放到各自的子类中去做，这样重复的代码只会出现一次了，下面来看一下实例，首先我们看一下抽象出来的 Dao 代码：

```
package com.weijia.template;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
```

```
import com.weijia.domain.DaoException;
import com.weijia.firstdemo.JdbcUtils;

public abstract class AbstractDao {

    /**
     * 更新
     */
    protected int update(String sql,Object[] args) {
        //这里需要做判断的,可能 args 为 null
        Connection conn = null;
        PreparedStatement st = null;
        try{
            conn = JdbcUtils.getConnection();
            st = conn.prepareStatement(sql);
            for(int i=0;i<args.length;i++){
                st.setObject(i+1, args[i]);
            }
            int count = 0;
            count = st.executeUpdate();
            System.out.println("更新的记录数:"+count);
            return count;
        }catch(Exception e){
            throw new DaoException(e.getMessage(),e);
        }finally{
            JdbcUtils.free(null, st, conn);
        }
    }

    /**
     * 查询
     * @param sql
     * @param args
     * @return
     */
    protected Object find(String sql,Object[] args){
        Connection conn = null;
        PreparedStatement st = null;
        ResultSet rs = null;
        try{
            conn = JdbcUtils.getConnection();
            st = conn.prepareStatement(sql);
            for(int i=0;i<args.length;i++){
```

```
            st.setObject(i+1, args[i]);
        }
        rs = st.executeQuery();
        Object obj = null;
        while(rs.next()){
            //不同的部分放到子类去做
            obj = rowMapper(rs);
        }
        return obj;
    }catch(Exception e){
        throw new DaoException(e.getMessage(),e);
    }finally{
        JdbcUtils.free(null, st, conn);
    }
}

//子类需要实现的结果集处理方法
protected abstract Object rowMapper(ResultSet rs) throws SQLException;

}
```

看一下 UserDaoImpl 类：

```
package com.weijia.template;

import java.sql.ResultSet;
import java.sql.SQLException;

import com.weijia.domain.User;

public class UserDaoImpl extends AbstractDao{

    /**
     * 更新用户信息
     */
    public int update(User user) {
        String sql = "udpate user set name=?,birthday=?,money=?,where id=?";
        Object[] args = new Object[]{user.getName(),user.getBirthday(),user.getMoney(),user.getId()};
        return super.update(sql, args);//相同的代码调用父类的方法即可
    }

    /**
     * 删除用户
     * @param user
     */
    public void delete(User user){
```

```
        String sql = "delete from user where id=?";
        Object[] args = new Object[]{user.getId()};
        super.update(sql, args);
    }

    /**
     * 查找用户
     * @param loginName
     * @param password
     * @return
     */
    public User findUser(String loginName){
        String sql = "select id,name,money,birthday from user where name=?";
        Object[] args = new Object[]{loginName};
        return (User)super.find(sql, args);
    }

    @Override
    protected Object rowMapper(ResultSet rs) throws SQLException{
        User user = new User();
        user.setId(rs.getInt("id"));
        user.setName(rs.getString("name"));
        user.setMoney(rs.getFloat("money"));
        user.setBirthday(rs.getDate("birthday"));
        return user;
    }

    //如果 insert 的时候不需要获取主键的话，也可以使用 super.update 方法实现的，这样代码就显得很整洁，相同的代码只需要一份即可(放在父类中)
    //不同的地方放到子类来实现
    //首先要区分哪些是变动的部分，哪些是不变的部分即可

}
```

ProductDaoImpl 类：

```
package com.weijia.template;

import java.sql.ResultSet;

public class ProductDaoImpl extends AbstractDao{

    public int update(){
        String sql = "update product set pname=?,price=? where pid=?";
        Object[] args = new Object[]{"drug",11,1};
        return super.update(sql, args);
```

```
    }

    @Override
    protected Object rowMapper(ResultSet rs) {
        return null;
    }

}
```

看到了，这样来实现的话，代码就很简洁了，这里的 ProductDaoImpl 类中没有写完，就是大概是那个意思。这里体现出了一个设计模式就是：模板模式

接着看，现在有一个问题，就是查询，其实 update 的方式很简单的，完全可以统一化的，因为查询需要处理查询之后的结果集，所以很纠结的，上面的例子中我们看到，我们查询的是一个 User 对象，假如现在我只是想查询一个用户的 name，那么我们只能在写一个 findUserName 方法了，同时还需要在 AbstractDao 父类中添加一个抽象方法的行映射器，这种方式就很纠结了，假如我们还有其他的查询需要的话，重复的代码又开始多了，这里我们将采用策略模式进行解决，我们只需要定义行映射器的接口：

```
package com.weijia.strategy;
import java.sql.ResultSet;
import java.sql.SQLException;
public interface RowMapper {
    public Object mapRow(ResultSet rs) throws SQLException;
}
```

在父类中只需要修改一下查询的方法：

```
/**
     * 查找用户
     * @param sql
     * @param args
     * @param rowMapper
     * @return
     */
    protected Object find(String sql,Object[] args,RowMapper rowMapper){
        Connection conn = null;
        PreparedStatement st = null;
        ResultSet rs = null;
        try{
            conn = JdbcUtils.getConnection();
            st = conn.prepareStatement(sql);
            for(int i=0;i<args.length;i++){
```

```
            st.setObject(i+1, args[i]);
        }
        rs = st.executeQuery();
        Object obj = null;
        while(rs.next()){
            obj = rowMapper.mapRow(rs);
        }
        return obj;
    }catch(Exception e){
        throw new DaoException(e.getMessage(),e);
    }finally{
        JdbcUtils.free(null, st, conn);
    }
}
```

添加了一个 RowMapper 接口变量

然后在子类中实现这个接口即可：

```
/**
 * 查询名称
 * @param id
 * @return
 */
public String findUserName(int id){
    String sql = "select name from user where id=?";
    Object[] args = new Object[]{id};
    Object user = super.find(sql, args,new RowMapper(){
        public Object mapRow(ResultSet rs) throws SQLException {
            return rs.getObject("name");
        }
    });
    return ((User)user).getName();
}

/**
 * 采用策略模式：传递不同的行为：C++中可以使用函数指针来实现，Java 中可以使用接口的回调来实现
 * @param loginName
 * @param password
 * @return
 */
public User findUser(String loginName){
    String sql = "select id,name,money,birthday from user where name=?";
    Object[] args = new Object[]{loginName};
    return (User)super.find(sql, args,new RowMapper(){
```

```
        public Object mapRow(ResultSet rs) throws SQLException {
            User user = new User();
            user.setId(rs.getInt("id"));
            user.setName(rs.getString("name"));
            user.setMoney(rs.getFloat("money"));
            user.setBirthday(rs.getDate("birthday"));
            return user;
        }
    });
}
```

我们可以看到这两个查询的方法就很优雅了，这样做了之后，我们只需要在指定的子类中添加指定的方法即可，其实策略模式很简单的，就是相当于回调机制，就是想执行指定的方法，但是 Java 中没有函数指针，C++中其实可以的，所以只能通过回调来实现了。

通过上面的 CRUD 优化之后，我们在进行操作的时候，代码编写是很方便和简洁的。

第 4 章 OR 映射和 Hibernate 框架

4.1 Hibernate 概述

4.1.1 什么是 Hibernate

Hibernate 是用于简化数据库操作的第三方开源工具。它实现 ORM(Object-Relational Mapping，对象关系映射)，在具体的操作业务对象时，就不需要和复杂的 SQL 语句打交道，只要像平时操作对象一样进行业务处理。如果不使用 Hibernate，就会出现很多重复的代码，例如：在 DAO 中，假设存在 createUser()、updateUser()、removeUser()等方法，则在每个方法里，都需要使用 JDBC 连接数据库，而且对于每个方法都要编写相应的 SQL 语句。使用 Hibernate 则可以有效地解决这个问题。

Hibernate 的目的是将 JDBC 进行轻量级的封装，用 Hibernate API 代替 JDBC API 实现数据库的相关操作，并将对数据库表的操作转换为对对象以及对象集合的操作。可以这样理解：Hibernate 封装了 Session 类用于替代 JDBC 中的 Connection 类；封装了 Query 类用于替代原有的 Statement 类（包括 PreparedStatement 类）；直接用集合对象替代原有的 ResultSet 类。其架构如下图所示。

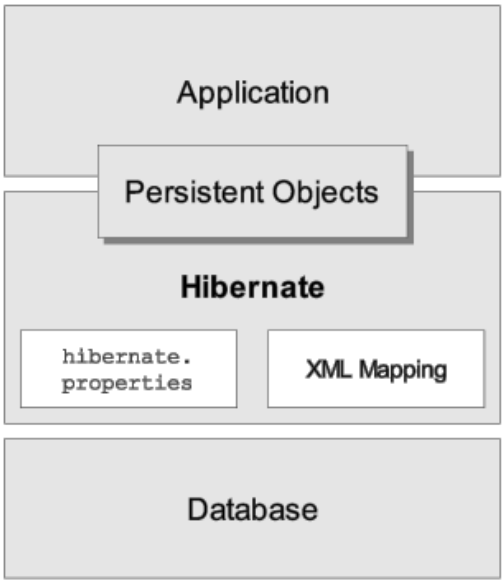


图 Hibernate 体系结构

在上图中整个框架主要有三层：应用层，数据持久层和数据库层。在应用层主要是对对象进行操作；在数据层主要针对的是关系型数据表。而在对象范例和关系范例存在着“阻抗不匹配”，使对象与关系表间的直接数据操作存在一定的困难。在基于 Hibernate 的应用系统中“阻抗不匹配”问题就由中间的 Hibernate 层的 O/R Mapping 来解决，

Hibernate 在对象范例和关系范例中建立映射关系，将应用层中对对象的操作直接作用于关系数据库中的表，使程序员不用再去关心数据库的操作问题。

4.1.2 Hibernate 如何运行

应用程序先调用 Hibernate API 中的 Configuration 类中的方法读取 Hibernate 配置文件及映射文件中的信息，并用这些信息生成一个 SessionFactory 对象（可以将该过程和 JDBC 中的 DriverManager 类进行对比）；然后通过 SessionFactory 对象获取一个 Session 对象（类似于 JDBC 中通过 DriverManage 类获取 Connection 对象）；用 Session 对象生成一个 Transaction 对象（启动事务）；可通过 Session 对象的 get()，load()，save()，update()，delete() 和 saveOrUpdate()等方法对持久化对象进行加载、保存、更新、删除等操作；在查询的情况下，可通过 Session 对象生成一个 Query 对象，然后利用 Query 对象执行查询操作（查询语言可以是 SQL 语句，也可以是 Hibernate 定义的 HQL 语句）；如果没有异常，Transaction 对象将提交这些操作结果到数据库中。Hibernate 的运行过程如下图所示。

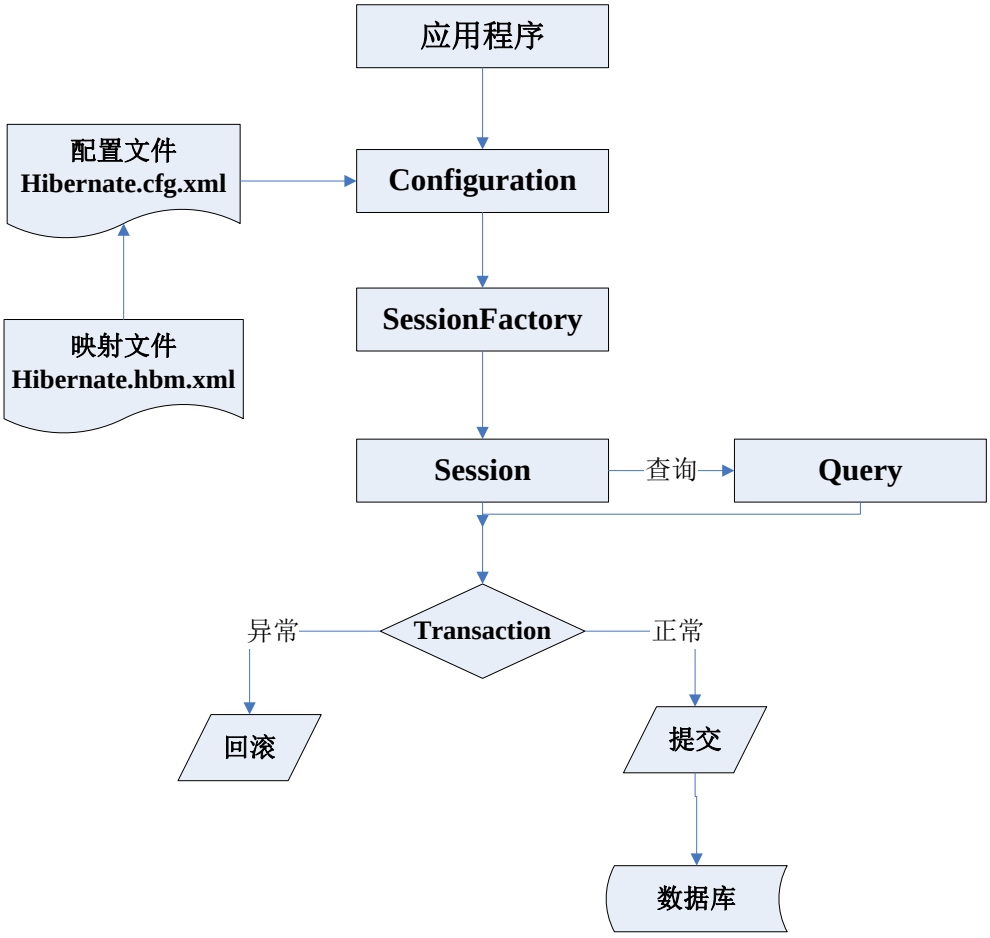


图 Hibernate 各组件之间的关系

4.2 第一个 Hibernate 应用实例

Hibernate 的开发需要一组类库文件和一些配置文件，这里以 Hibernate3 为例介绍开发环境的配置和简单应用的开发，数据库采用 SQL Server，并以其中的例子数据库 pubs 为例。

4.2.1 配置 Hibernate 开发框架环境

首先，获取 Hibernate 类库文件，Hibernate3.1 类库如图所示，其核心为 Hibernate3.jar，通常还包括一些第三方类库。同时，获取数据库的驱动程序文件，图所示为 SQL Server 的一个第三方 JDBC 驱动程序库。

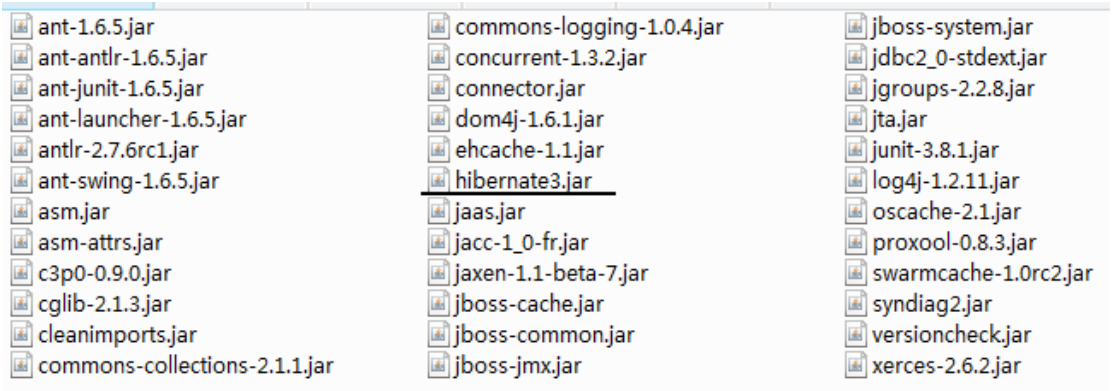


图 Hibernate3.1 类库

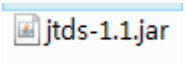


图 SQL Server 的一个第三方 JDBC 驱动程序库

第二步：启动 Eclipse，并运行菜单 windows→preferences。并对 Java →Build Path→User Libraries 进行配置，如下图所示。

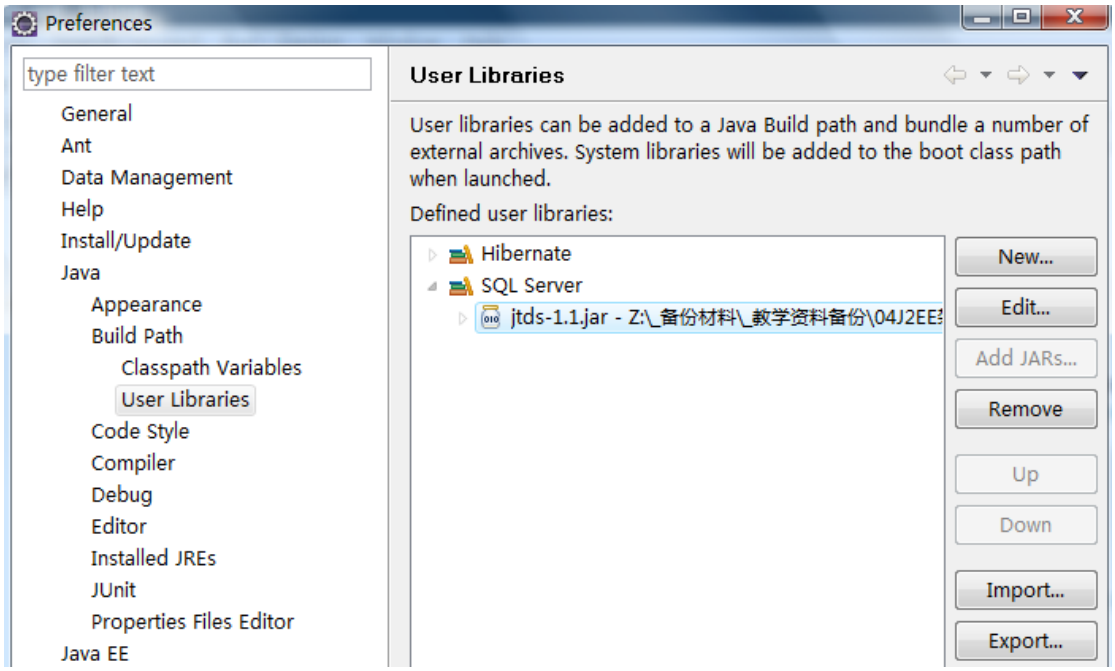


图 Hibernate 及 JDBC 驱动程序类库配置

4.2.2 应用开发

第一步：在当前 workspace 下建立一个 Java 工程，并命名为 hibernateTest，其它选项采用默认值。设置工程属性 Java Build Path，如下图 8.6 所示。其中 Hibernate 和 SQL Server 为前面步骤设定的用户类库，可通过 Add Library 按钮添加。

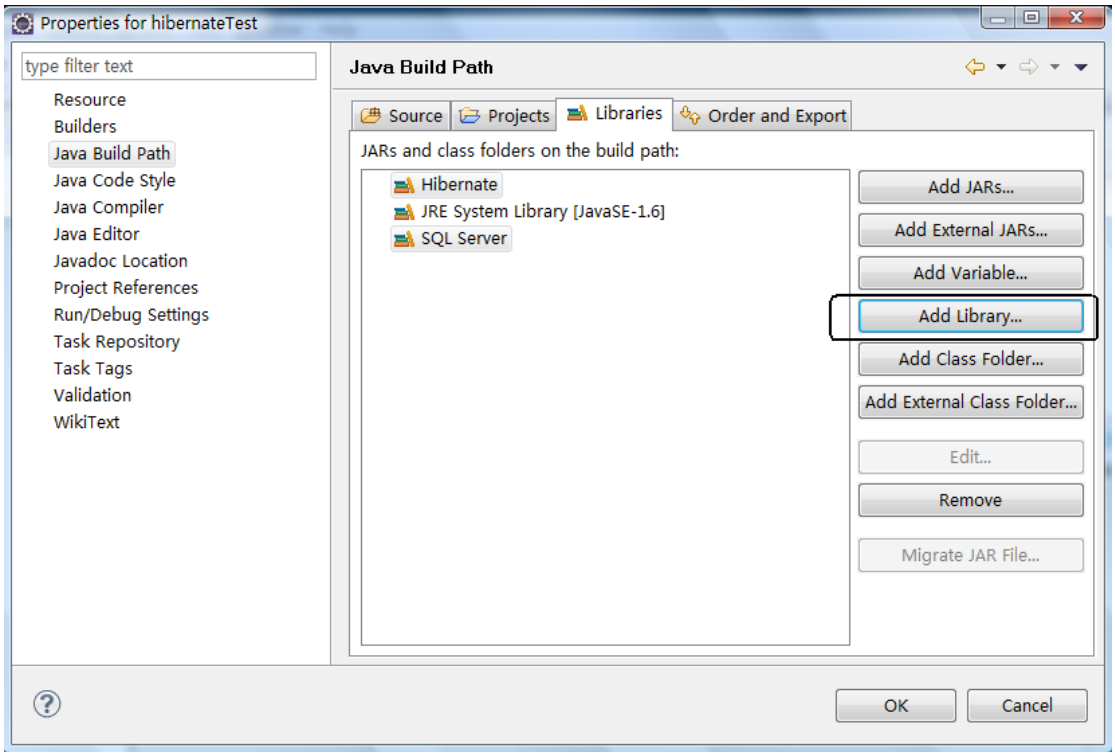


图 工程类库设置

第二步：在工程的 src 目录下，建立一个 hibernate.cfg.xml 文件，文件内容如下。该文件为 Hibernate 的主配置文件，指明了目标数据库的参数（包括驱动程序的指定、JDBC URL 的指定、用户名密码的设定等）。

代码 Hibernate 主配置文件 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>
        <property
name="connection.driver_class">net.sourceforge.jtds.jdbc.Driver</property>
        <property
name="connection.url">jdbc:jtds:sqlserver://127.0.0.1:1433/pubs</property>
        <property name="connection.username">sa</property>
        <property name="connection.password">sa</property>
        <property name="connection.pool_size">50</property>
        <property name="dialect">org.hibernate.dialect.SQLServerDialect</property>
        <property name="current_session_context_class">thread</property>
        <property
name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>
        <property name="show_sql">true</property>
    </session-factory>
</hibernate-configuration>
```

第三步：建立测试类 HibernateTest，测试 Hibernate 是否能够成功连接到数据库。以 Java Application 方式运行该测试类可进行数据库连接测试。

代码 8.2 HibernateTest.java

```
package com.firsthibernate;
import java.sql.SQLException;
import org.hibernate.*;
import org.hibernate.cfg.Configuration;
public class HibernateTest {
    private static SessionFactory sf=null;
    private static Session getSession(){
        if(sf==null){
            Configuration config=new Configuration();
            config.configure();//读配置文件
            sf=config.buildSessionFactory();//得到工厂
        }
        return sf.openSession();
    }
    public static void main(String[] args) {
```

```
            Session session=HibernateTest.getSession();
            try {
                if(!session.connection().isClosed())
                    System.out.println("数据库连接成功");
                else
                    System.out.println("数据库连接失败");
            } catch (Exception e) {
                e.printStackTrace();
            }
            session.close();//关闭资源
        }
    }
```

第四步：建立表示 publishers 表中数据的 JavaBean。其中 publishers 表的结构如下图所示。

	列名	数据类型	长度	允许空
🔑	pub_id	char	4	
	pub_name	varchar	40	✓
	city	varchar	20	✓
	state	char	2	✓
	country	varchar	30	✓

图　publishers 表结构

代码 Publisher.java

```
package com.firsthibernate.publisher;
public class Publisher {
    private String  pubId;
    private String pubName;
    private String city;
    private String state;
    private String country;
    public String getPubId() {          return pubId;  }
    public void setPubId(String pubId) {          this.pubId = pubId; }
    public String getPubName() {       return pubName;   }
    public void setPubName(String pubName) {       this.pubName = pubName;    }
    public String getCity() {         return city;     }
    public void setCity(String city) {          this.city = city; }
    public String getState() {       return state;   }
    public void setState(String state) {       this.state = state;   }
    public String getCountry() {         return country;      }
    public void setCountry(String country) {       this.country = country;   }
}
```

第五步：建立 Publisher 类和 Publishers 表的映射关系。在上述类的包下建立一个 Publisher.hbm.xml 文件，文件内容如下所示。

代码 Publisher.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
      "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
       <class name="com.firsthibernate.publisher.Publisher" table="Publishers">
           <id name="pubId" column="pub_id" type="java.lang.String" unsaved-
value="null">
                <generator class="assigned"/>
           </id>
           <property name="pubName" column="pub_Name" type="java.lang.String" />
           <property name="city" column="city" type="java.lang.String" />
           <property name="state" column="state" type="java.lang.String" />
           <property name="country" column="country" type="java.lang.String" />
      </class>
</hibernate-mapping>
```

第六步：改写 Hibernate.cfg.xml 文件，将上述 Hbm.xml 加入 Mapping 中。

```
           …
           <property name="show_sql">true</property>
           <mapping resource="com/firsthibernate/publisher/Publisher.hbm.xml"/>
           …
```

第七步：改写 HibernateTest 类，加入获取 Publishers 表中所有数据并输入的代码，并运行查看结果。

```
      …
           String hql="from Publisher";
           Query qry=session.createQuery(hql);
           List<Publisher> result=qry.list();
           Iterator<Publisher> it=result.iterator();
           while(it.hasNext()){
                Publisher p=it.next();
                System.out.println(p.getPubName());
           }
           session.close();//关闭资源
           …
```

综上所述，应用 Hibernate 框架进行程序设计的主要工作包括：

（1）定制主配置文件，使之能连接到相应数据库；

（2）根据数据库表结构建立对应的 JavaBean，并编写映射文件配置 JavaBean 和表的关系，包括表字段和 JavaBean 属性的关系；

（3）在主配置文件中添加对映射文件的引用；

（4）利用 Hibernate 中相关 API，特别是 Session、Query 等类进行相关数据操作。

4.3 Hibernate 主配置文件及 SessionFactory 类

Hibernate 的主配置文件用于设置连接数据库的信息和其它全局信息，可以采用两种形式进行相关配置：Hibernate.Properties 文件或 Hibernate.cfg.xml 文件。这里只介绍后一种形式，前一种形式读者可以参考相关文献。在 Hibernate.cfg.xml 文件中，根节点为<hibernate-configuration>标签，一般只需要设置其<session-factory>子标签的内容，各配置项都以<property>标签表达，对于每一个配置项，都有 name 属性和值，其一般形式如下：

```
<property
name="connection.driver_class">net.sourceforge.jtds.jdbc.Driver</property>
```

上述代码配置了数据库连接的 JDBC 驱动程序。这里介绍一下几个主要的配置项：

- connection.driver_class：JDBC 驱动程序类
- connection.url：JDBC URL
- connection.username：数据库用户名
- connection.password：数据库密码
- connection.pool_size：数据库连接池大小
- dialect：HQL→SQL 转换类(SQL 方言)，用于实现 HQL 到 SQL 语句的转换，以保证数据库独立性，也就是 HQL 语句在不同的数据库中，其转换出来的 SQL 语句会有差别。其中 SQL Server 的转换类为：org.hibernate.dialect.SQLServerDialect。
- show_sql：SQL 显示开关，用户控制是否显示转换出来的 SQL 语句，便于调试。

主配置文件的最重要目标就是根据配置文件建立 SessionFactory 对象，为 Session 对象的获取建立基础。

```
Configuration config=new Configuration();
config.configure();//读配置文件
sf=config.buildSessionFactory();//得到工厂
```

通常情况下，会对 SessionFactory 对象进行封装，以构造一个工具类，用于获取 Session 对象。下述代码实现了一个 HibernateUtil 类，通过该类的两个静态方法可以获取一个 Session 对象（该代码可以保证在同一个线程中获取到的 Session 对象为同一个），释放当前线程占用的 Session 对象。

代码 HibernateUtil.java

```
package com.firsthibernate.util;
import org.hibernate.*;
import org.hibernate.cfg.Configuration;
public class HibernateUtil {
    private static SessionFactory sf=null;
    private static ThreadLocal<Session> threadLocal=new ThreadLocal<Session>();
    public static  Session getSession(){
        Session session=threadLocal.get();
        if(session!=null && session.isOpen()) return session;
        if(sf==null){
            Configuration config=new Configuration();
            config.configure();//读配置文件
            sf=config.buildSessionFactory();//得到工厂
        }
        session= sf.openSession();
        threadLocal.set(session);
        return session;
    }
    public static void closeSession(){
        Session session=threadLocal.get();
        threadLocal.set(null);
        if(session!=null) session.close();
    }
}
```

4.4 单表映射

4.4.1 hbm.xml 文件概述

Hbm.xml 文件是配置 JavaBean 和数据表映射的主要文件，其主要目标是建立 JavaBean 和数据库表的对应关系，以及数据库表字段和 JavaBean 属性间的关系、属性的数据类型等。Hbm.xml 文件的根节点是<hibernate-mapping>标签，在一个 hbm.xml 文件中可以配置多个 JavaBean 的映射。对于每一个 JavaBean 的映射，通过<class>标签描述。在 class 标签中必须指明 JavaBean 的类名（类名中应包括包名）和数据库表名。对于每一个映射，都必须指定主码，主码的指定通过<id>标签，其中包括主码对应的 JavaBean 属性、数据库字段名以及主码产生的规则。其它属性通过<property>标签指明，必须指定 JavaBean 属性名、数据库字段名以及数据类型。具体配置实例请参考前述例子。

4.4.2 映射主码

每一个映射的 JavaBean 都必须指定主码（可以是一个字段，也可以是多个字段），这个主码通过<id>标签进行定义。

- 单一字段主码映射

```
<id name="pubId" column="pub_id" type="java.lang.String" unsaved-value="null">
    <generator class="assigned"/>
</id>
```

  - name（可选）：JavaBean 中表示主码的属性，默认值为 id。
  - column（可选）：数据库表中的字段名，默认值是 name 属性值。
  - type（可选）：字段类型，默认值是 name 属性指定的 JavaBean 的属性的类型。
  - <generator>子标签（可选）：用来指定主码值生成的策略，由 class 属性指定。如果不指定<generator>子标签，则 class 属性的默认值是 assigned。

表　主码生成规则

class 值	主码参数规则
Assigned	在代码中赋值产生。
Uuid	32 字节的 UUID 字符串，可以保证唯一。
Increment	对 long、short 或 int 的数据列生成自动增长主键。 主键按数值顺序递增。此方式的实现机制为在当前应用实例中维持一个变量，以保存着当前的最大值，之后每次需要生成主键的时候将此值加 1 作为主键。这种方式可能产生的问题是：如果当前有多个实例访问同一个数据库，那么由于各个实例各自维护主键状态，不同实例可能生成同样的主键，从而造成主键重复异常。因此，如果同一数据库有多个实例访问，此方式必须避免使用。
Identity	对如 SQL Server、MySQL 等支持自动增长列的数据库，如果数据列的类型是 long、short 或 int，可使用主键生成器生成自动增长主键。
Seqhilo	对如 Oracle、DB2 等支持 Sequence 的数据库，如果数据列的类型是 long、short 或 int，可使用该主键生成器生成自动增长主键。
Native	由 Hibernate 根据底层数据库自行判断采用 identity、hilo、sequence 其中一种作为主键生成方式。

- 复合主码映射

复合主码通过<composite-id>标签进行映射，例如：

```
<composite-id name=”pk” class=” com.firsthibernate.compositeid”>
<key-property name=”key1”    column=”key1” />
<key-property name=”key1”    column=”key1” />
</composite-id>
```

- ◆ Name 属性：JavaBean 中表示主码的属性名
- ◆ Class 属性：JavaBean 中表示主码的属性的数据类型
- ◆ <key-property>子标签：主码对象中属性与数据库表字段的对应关系

代码 PrimaryKey.java

```
package com.firsthibernate.compositeid;
public class PrimaryKey {
        private String key1;
        private String key2;
        public String getKey1() {
                return key1;
        }
        public void setKey1(String key1) {
                this.key1 = key1;
        }
        public String getKey2() {
                return key2;
        }
        public void setKey2(String key2) {
                this.key2 = key2;
        }
}
```

4.4.3 映射普通属性

JavaBean 的其它需要持久化的属性也需要进行配置，这些配置通过<property>标签进行。例如：

```
<property name="pubName" column="pub_Name" type="java.lang.String" />
```

其中 name 属性表示 JavaBean 的属性名，column 表示数据库表的字段名，type 为 JavaBean 中属性的数据类型。

4.5 Hibernate 基础操作

在完成 Hibernate 主配置文件和映射文件的配置后，可利用 Hibernate 定义的一系列 API 进行相关数据库操作，包括数据库查询和数据的增删改。

4.5.1 Hibernate 基础 API

Hibernate 的 API 基础体系结构如下图所示。其中持久层即为 Hibernate 的主要 API，在前面的内容中，已经介绍了 Configuration 和 SessionFactory，这里将主要介绍 Session、Transaction 和 Query。

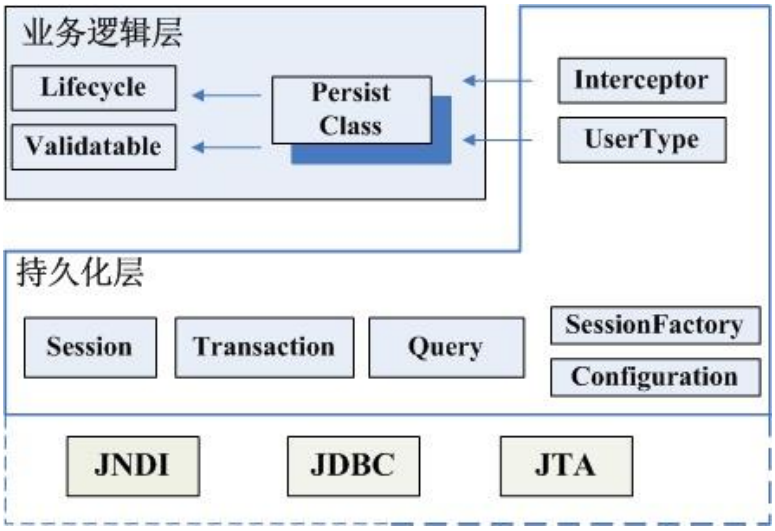


图 Hibernate 基础 API 体系结构

- Session 对象可以理解为一次数据库连接和处理过程，通过 SessionFactory 获取一个 Session 对象即建立起数据库连接。事实上，通过 session.connection()方法即可获得一个 JDBC 的 Connection 对象。同时，Hibernate 还在 Session 对象中封装了很多对持久化对象的操作。因此，还可以将 Session 对象理解为持久化对象的容器，可以从容器中提取指定主码及类型的持久化对象，可以将持久化对象放入 Session 中等操作。
- Transaction 对象是 Hibernate 对数据库事务的封装，可以完成业务代码的事务控制。
- Query 对象是用于执行查询语句的对象，通过该对象可进行各种数据库查询。

4.5.2 基本数据查询

Hibernate 从数据库中查询数据可以分为两种情况：通过主码提取对象；通过查询语句提取对象。

通过主码提取对象

在 Hibernate 中，可以通过 Session 对象的 get 方法获取指定主码的对象，如果对应主码的对象不存在，则获得一个空对象。从下面的代码中可以看出，get 方法的参数需要两个，分别用于指定需要获取的持久化对象类型和主码值。

```
Publisher one=(Publisher) session.get(Publisher.class, "0736");
if(one!=null)
    System.out.println(one.getPubName());
```

通过查询语句提取对象

在 Hibernate 中，通过 Query 对象利用 HQL 语句进行数据查询，HQL 语句的具体语法见后续章节。这里按查询结果的形式介绍几种典型的查询方式。Query 对象类似于 JDBC 中的 Statement 对象，它由 session 对象创建。

```
String hql="from Publisher where pubId='0736' ";
Session session=HibernateTest.getSession();
Query qry=session.createQuery(hql);
```

- 通过 HQL 提取单个对象

根据 HQL 语句创建了 Query 对象后，如果该 HQL 语句将返回一个唯一对象，则可通过 uniqueResult 方法获取该对象。

注：如果返回多个对象，则抛出 org.hibernate.NonUniqueResultException。

代码 8.7 通过 HQL 提取单个对象

```
private static void testPickOneObjectByHQL(){
        String hql="from Publisher where pubId='0736' ";
        Session session=HibernateTest.getSession();
        Query qry=session.createQuery(hql);
        Publisher one=(Publisher)qry.uniqueResult();
        if(one!=null)
            System.out.println(one.getPubName());
        session.close();
    }
```

- 通过 HQL 提取对象列表

通过 HQL 语句创建 Query 对象后，可通过 Query 对象的 list 方法获取一个对象的 List。

代码 8.8 通过 HQL 提取对象列表

```
private static void testPickObjectListByHQL(){
        System.out.println("-----testPickObjectListByHQL-----");
        String hql="from Publisher where pubName like '%oo%' ";
        Session session=HibernateTest.getSession();
        Query qry=session.createQuery(hql);
        List<Publisher> result=qry.list();
        for(int i=0;i<result.size();i++)
            System.out.println(result.get(i).getPubName());
        session.close();
    }
```

- 通过 HQL 进行分页查询

Hibernate 支持在 HQL 层面的分页查询，即在查询前通过设置 Query 对象的 firstResult 属性和 maxResults 属性来确定需要从数据库中查询数据的位置。其中 firstResult 属性表示起始记录，maxResults 表示返回的最大记录数。下面的例子演示了一个分页查询的实现，该查询提取第二页数据，每页三条，即提取第四条到第六条数据。

代码 8.9 通过 HQL 分页查询的实现

```
private static void testPickObjectPageByHQL(){
        System.out.println("-----testPickObjectPageByHQL-----");
        String hql="from Publisher";
        Session session=HibernateTest.getSession();
        Query qry=session.createQuery(hql);
        int pagesize=3;
        int currentPage=2;
        qry.setFirstResult((currentPage-1)*pagesize);
        qry.setMaxResults(pagesize);
        List<Publisher> result=qry.list();
        for(int i=0;i<result.size();i++)
            System.out.println(result.get(i).getPubName());
        session.close();
    }
```

- 通过 HQL 查询对象的部分属性

通过 Hibernate 还可以查询对象的部分属性，这时查询结果为一个 Object[]的 List 对象，即每一条结果记录将被封装成一个 Object 数组。

代码 8.10 通过 HQL 查询对象的部分属性

```
private static void testPickObjectsSomeProperyByHQL(){
        System.out.println("-----testPickObjectsSomeProperyByHQL-----");
        String hql="select pubId,pubName from Publisher";
        Session session=HibernateTest.getSession();
        Query qry=session.createQuery(hql);
        List<Object[]> result=qry.list();
        for(int i=0;i<result.size();i++)
            System.out.println(result.get(i)[0]+"--"+result.get(i)[1]);
        session.close();
    }
```

4.5.3 基本数据操作

对于数据库表的操作无外乎增、删、改操作，在 Hibernate 中，这些操作体现为对 JavaBean 的增、删、改操作。这里以 pubs 数据库中的 jobs 表的操作为例来学习 Hibernate 的基本数据操作功能。

	列名	数据类型	长度	允许空
🔑	job_id	smallint	2	
	job_desc	varchar	50	
	min_lvl	tinyint	1	
	max_lvl	tinyint	1	

图 jobs 表结构

根据 jobs 表的结构，定义 JobBean 用于表达内存数据，并配置 JobBean.hbm.xml 文件关联数据库表和 JavaBean 的映射关系。

代码 JobBean.java

```
package com.firsthibernate.job;
public class JobBean {
    private int jobId;
    private String jobDesc;
    private int minLvl;
    private int maxLvl;
    public int getJobId() {          return jobId;    }
    public void setJobId(int jobId) {      this.jobId = jobId;    }
    public String getJobDesc() {          return jobDesc;}
    public void setJobDesc(String jobDesc) {       this.jobDesc = jobDesc;    }
    public int getMinLvl() {          return minLvl;  }
    public void setMinLvl(int minLvl) {         this.minLvl = minLvl;}
```

```
        public int getMaxLvl() {            return maxLvl;  }
        public void setMaxLvl(int maxLvl) {          this.maxLvl = maxLvl;       }
    }
```

代码 8.12 JobBean.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
      "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
       <class name="com.firsthibernate.job.JobBean" table="jobs">
           <id name="jobId" column="Job_id" type="int" unsaved-value="null">
                <generator class="identity"/>
           </id>
           <property name="jobDesc" column="job_desc" type="java.lang.String" />
           <property name="minLvl" column="min_lvl" type="int" />
           <property name="maxLvl" column="max_lvl" type="int" />
      </class>
</hibernate-mapping>
```

注：由于 job_id 字段在数据库中为标识字段，这里将主码产生的类型定为 identity。

最后，需要在主配置文件 Hibernate.cfg.xml 中增加对该映射文件的引用。

```
<mapping resource="com/firsthibernate/job/JobBean.hbm.xml"/>
```

事务控制和对象状态

一般情况下，进行数据修改时，将修改的代码置于事务中，在 Hibernate 中可用 Tranaction 类进行事务管理，下述代码为典型的基于事务的编程模式。

代码事务控制

```
    private static void aFunction(){
          Session s=com.firsthibernate.util.HibernateUtil.getSession();
          Transaction tx=s.beginTransaction();
          try{
                //编写业务代码
                tx.commit();
          }
          catch(Exception ex){
                ex.printStackTrace();
                tx.rollback();
          }
```

```
        finally{
                HibernateUtil.closeSession();
            }
        }
```

在 Hibernate 中，对象一般存在三种状态：瞬时、持久、脱管。

- 瞬时（Transient）：由 new 操作符创建，且尚未与 Hibernate Session 关联的对象被认定为瞬时的。瞬时对象不会被持久化到数据库中，也不会被赋予持久化标识(identifier)。 如果程序中没有保持对瞬时对象的引用，它会被垃圾回收器销毁。使用 Hibernate Session 可以将其变为持久(Persistent)状态。
- 持久（Persistent）：持久的实例在数据库中有对应的记录，并拥有一个持久化标识（identifier）。持久的实例可能是刚被保存的，或刚被加载的，无论哪一种，按定义对象都仅在相关联的 Session 生命周期内保持这种状态。Hibernate 会检测到处于持久状态的对象的任何改动，在当前操作单元执行完毕时将对象数据与数据库同步。开发者不需要手动执行 UPDATE。
- 脱管（Detached）：与持久对象关联的 Session 被关闭后，对象就变为脱管的。对脱管对象的引用依然有效，对象可继续被修改。脱管对象如果重新关联到某个新的 Session 上，会再次转变为持久的（其间的改动将被持久化到数据库）。这个功能使得一种编程模型，即中间会给用户思考时间的长时间运行的操作单元的编程模型成为可能。

增加数据

在 Hibernate 中，增加数据就是将瞬时对象存入数据库表中，并将该对象与 Session 关联，转换为持久对象。下述代码即为一个典型的增加数据代码。

代码 增加数据

```
private static void addJob(){
    Session s=HibernateUtil.getSession();
    Transaction tx=s.beginTransaction();
    try{
        JobBean aJob=new JobBean();
        aJob.setJobDesc("编辑工作");
        aJob.setMaxLvl(10);
        aJob.setMinLvl(10);
        s.save(aJob);
        tx.commit();
```

```
            }
            catch(Exception ex){
                ex.printStackTrace();
                tx.rollback();
            }
            finally{
                HibernateUtil.closeSession();
            }
        }
```

修改数据

- 通过瞬时对象修改数据

代码 通过外部 JavaBean 修改数据

```
    private static void modifyJob1(){
            Session s=HibernateUtil.getSession();
            Transaction tx=s.beginTransaction();
            try{
                JobBean aJob=new JobBean();
                aJob.setJobId(17);
              //从数据库中查出需要修改的 job 的主码，笔者的数据库中为 17
                aJob.setJobDesc("编辑工作 1");
                aJob.setMaxLvl(10);
                aJob.setMinLvl(10);
                s.update(aJob);
                tx.commit();
            }
            catch(Exception ex){
                ex.printStackTrace();
                tx.rollback();
            }
            finally{
                HibernateUtil.closeSession();
            }
        }
```

在上例中，aJob 对象在执行 s.update(aJob)前是一个新对象，和 Session 对象没有关联；执行 s.update(aJob)后，将用 aJob 对象替换数据库中相同主码的记录。

- 通过持久对象修改数据

代码 8.16 通过关联的 JavaBean 修改数据

```
    private static void modifyJob2(){
            Session s=HibernateUtil.getSession();
```

```
        Transaction tx=s.beginTransaction();
        try{
            JobBean aJob=(JobBean)s.get(JobBean.class, 17);
            aJob.setJobDesc("编辑工作 2");
            tx.commit();
        }
        catch(Exception ex){
            ex.printStackTrace();
            tx.rollback();
        }
        finally{
            HibernateUtil.closeSession();
        }
    }
```

在上例中，aJob 对象在执行 aJob=s.get(JobBean.class,17)后就是一个和 Session 对象关联的 JavaBean，因此对该对象的修改将在递交事务时自动对数据库数据进行修改。如果希望这个修改不递交到数据库中，则需要通过 Session 的 evict 方法将 JavaBean 与 Session 对象的关联解除。下述代码不会修改数据库数据。

代码 8.17 evict 用法

```
private static void modifyJob3(){
        Session s=HibernateUtil.getSession();
        Transaction tx=s.beginTransaction();
        try{
            JobBean aJob=(JobBean)s.get(JobBean.class, 17);
            s.evict(aJob);
            aJob.setJobDesc("编辑工作 3");//这个修改不会反应到数据库
            tx.commit();
        }
        catch(Exception ex){
            ex.printStackTrace();
            tx.rollback();
        }
        finally{
            HibernateUtil.closeSession();
        }
    }
```

删除数据

在 Hibernate 中，可以通过 Session 对象的 delete 方法从数据库中删除数据。

代码 8.18 删除数据

```
private static void deleteJob(){
        Session s=HibernateUtil.getSession();
        Transaction tx=s.beginTransaction();
        try{
            JobBean aJob=(JobBean)s.get(JobBean.class, 17);
            s.delete(aJob);
            tx.commit();
        }
        catch(Exception ex){
            ex.printStackTrace();
            tx.rollback();
        }
        finally{
            HibernateUtil.closeSession();
        }
    }
```

4.6 多表映射

前面介绍了单表数据的查询、增、删、改等操作的基本方法。另外，在数据库中经常通过外码关联表，以建立数据之间的复杂关系。通过 Hibernate，可以将这种表之间的关联映射为 JavaBean 之间的关联。

4.6.1 关联的种类

在数据库的表中，一般存在一对一、一对多、多对多关联。同样，在 JavaBean 之间也可以存在这样的关联，而且这些关联又可以根据关联的方向进行细分。

- 一对一单向关联：在定义相关的两个 JavaBean 时，可以为其中一个 JavaBean 添加一个类型为另一个 JavaBean 的属性，这样通过这个 JavaBean 对象可以直接访问相关的另一个 JavaBean 对象。
- 一对一双向关联：在定义相关的两个 JavaBean 时，可以为两个 JavaBean 都添加一个类型为另一个 JavaBean 的属性，这样通过任何一个 JavaBean 对象可以直接访问相关的另一个 JavaBean 对象。

- 一对多单向关联：在定义相关的两个 JavaBean 时，可以为其中“一”一方的 JavaBean 添加一个类型为另一个 JavaBean 集合的属性，这样通过这个 JavaBean 对象可以直接访问相关的另一个 JavaBean 对象集合。
- 一对多双向关联：在定义相关的两个 JavaBean 时，可以为其中“一”一方的 JavaBean 添加一个类型为另一个 JavaBean 集合的属性，这样通过这个 JavaBean 对象可以直接访问相关的另一个 JavaBean 对象集合。同时为“多”一方的 JavaBean 添加一个类型为另一个 JavaBean 的属性，这样通过这个 JavaBean 对象可以直接访问相关的另一个 JavaBean 对象。
- 多对一单向关联：在定义两个相关的 JavaBean 时，为“多”一方的 JavaBean 添加一个类型为另一个 JavaBean 的属性，这样通过这个 JavaBean 对象可以直接访问相关的另一个 JavaBean 对象。
- 多对多双向关联：在定义相关的两个 JavaBean 时，可以为两个 JavaBean 都添加一个类型为另一个 JavaBean 集合的属性，这样通过这个 JavaBean 对象可以直接访问相关的另一个 JavaBean 对象集合。

4.6.2 关联的映射

这里，介绍比较常用的多对一单向关联（包括一对一单向关联）、一对多单向关联。其它关联方式请参阅 Hibernate 文档。

一对一单向关联和多对一单向关联

数据库中存在一对一关联的两章表 Person 和 Card，其表结构如下图 8.10 所示。其中 Person 表的 CardId 字段为外码。一对一单向关联其实是多对一单向关联的一个特例，这里采用相同的方式进行介绍。

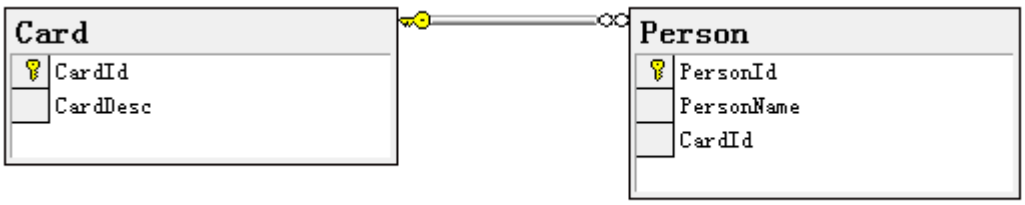


图 8.10 Person 和 Card 表

为此，可以设计两个 JavaBean：Person 和 Card 分别表示两张表的数据，并建立一对一单向关联。

代码 8.19 Card.java 和 Person.java

```
package com.firsthibernate.relation.one2one;
public class Card {
    private String cardId;
    private String cardDesc;
    public String getCardId() {     return cardId;  }
    public void setCardId(String cardId) {          this.cardId = cardId; }
    public String getCardDesc() {       return cardDesc;    }
    public void setCardDesc(String cardDesc) {          this.cardDesc = cardDesc; }
}

package com.firsthibernate.relation.one2one;
public class Person {
    private String id;
    private String name;
    private Card card;
    public String getId() {         return id; }
    public void setId(String id) {          this.id = id;   }
    public String getName() {       return name;    }
    public void setName(String name) {      this.name = name;   }
    public Card getCard() {         return card;    }
    public void setCard(Card card) {            this.card = card;s    }
}
```

从上面 JavaBean 的设计中可以看出，Person 类中没有 CardId 对应的属性，而用 Card 类型的对象属性对应，需要通过映射配置将 CardID 这个外码对应的关系体现为 JavaBean 的关联，其映射文件配置如下。

代码 8.20 Card 和 Person 的映射文件

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
     <class name="com.firsthibernate.relation.one2one.Card" table="Card">
        <id name="cardId" column="cardId" type="java.lang.String" unsaved-value="null">
            <generator class="assigned"/>
        </id>
        <property name="cardDesc" column="cardDesc" type="java.lang.String" />
    </class>
     <class name="com.firsthibernate.relation.one2one.Person" table="Person">
```

```
            <id name="id" column="PersonId" type="java.lang.String" unsaved-value="null">
                <generator class="assigned"/>
            </id>
            <property name="name" column="PersonName" type="java.lang.String" />
            <many-to-one name="card" column="cardId"
             class="com.firsthibernate.relation.one2one.Card"
             cascade="all" unique="true"/>
        </class>
    </hibernate-mapping>
```

该应用配置的关键是<many-to-one>标签，这里对该标签的配置进行简单解释，关于其详细内容，请参阅 Hibernate 文档。

```
    <many-to-one
            name="propertyName"                                    (1)
            column="column_name"                                   (2)
            class="ClassName"                                      (3)
            cascade="cascade_style"                                (4)
            fetch="join|select"                                    (5)
            update="true|false"                                    (6)
            insert="true|false"                                    (7)
            property-ref="propertyNameFromAssociatedClass"         (8)
            access="field|property|ClassName"                      (9)
            unique="true|false"                                    (10)
            not-null="true|false"                                  (11)
            optimistic-lock="true|false"                           (12)
            lazy="true|proxy|false"                                (13)
            not-found="ignore|exception"                           (14)
            entity-name="EntityName"                               (15)
            node="element-name|@attribute-name|element/@attribute|."
            embed-xml="true|false"
    />
```

(1) name：属性名。

(2) column（可选）：外键字段名。它也可以通过嵌套的<column>元素指定。

(3) class（可选，默认是通过反射得到属性类型）：关联的类的名字。

(4) cascade（级联）（可选）：指明哪些操作会从父对象级联到关联的对象。

cascade 属性设置为除了 none 以外任何有意义的值，它将把特定的操作传播到关联对象中。这个值就代表着 Hibernate 基本操作的名称， persist、merge、delete、save-update、evict、replicate、lock、refresh，以及特别的值 delete-orphan 和 all，并且可以用逗号分隔符来合并这些操作，例如，cascade="persist,merge,evict"或

cascade="all,delete-orphan"。通常在<many-to-one>或<many-to-many>关系中应用级联（cascade）没什么意义。级联通常在 <one-to-one>和<one-to-many>关系中比较有用。

如果子对象的寿命限定在父亲对象的寿命之内，可通过指定 cascade="all,delete-orphan"将其变为自动生命周期管理的对象（lifecycle object）。

其它情况，可根本不需要级联，可以使用 cascade="all"将一个关联关系（无论是对值对象的关联，或者对一个集合的关联）标记为父/子关系的关联。这样对父对象进行 save/update/delete 操作就会导致子对象也进行 save/update/delete 操作。

(5) fetch（可选，默认为 select）：在外连接抓取（outer-join fetching）和序列选择抓取（sequential select fetching）两者中选择其一。

(6) update/insert（可选，默认为 true) ：指定对应的字段是否包含在用于 UPDATE 或 INSERT 的 SQL 语句中。如果二者都是 false，则这是一个纯粹的 “外源性（derived）”关联，它的值是通过映射到同一个（或多个）字段的某些其它属性或者通过 trigger（触发器）或其它程序得到。

(7) property-ref （可选）：指定关联类的一个属性，这个属性将会和本外键相对应。如果没有指定，会使用对方关联类的主键。

(8) access（可选，默认是 property）：Hibernate 用来访问属性的策略。

(9) unique（可选）：使用 DDL 为外键字段生成一个唯一约束。此外，这也可以用作 property-ref 的目标属性。这使关联同时具有一对一的效果。

(10) not-null（可选）：使用 DDL 为外键字段生成一个非空约束。

(11) optimistic-lock（可选，默认为 true）：指定这个属性在做更新时是否需要获得乐观锁定（optimistic lock）。 换句话说，它决定这个属性发生脏数据时版本（version）的值是否增长。

(12) lazy（可选，默认为 proxy）：默认情况下，单点关联是经过代理的。lazy="true"指定此属性在实例变量第一次被访问时应该延迟抓取（fetche lazily）（需要运行时字节码的增强）。lazy="false"指定此关联总是被预先抓取。

(13) not-found（可选，默认为 exception）：指定外键引用的数据不存在时如何处理，ignore 会将数据不存在作为关联到一个空对象（null）处理。

(14) entity-name（可选）：被关联的类的实体名。

- 添加数据

Card 对象和 Person 对象的添加可采用如下模式进行。

代码 8.21 添加对象

```
private    static void addPerson(){
            Person p=new Person();
            p.setId("00001");
            p.setName("test person");
            Card c=new Card();
            c.setCardDesc("a card");
            c.setCardId("card00001");
            p.setCard(c);
            Session s=HibernateUtil.getSession();
            Transaction tx=s.beginTransaction();
            try{
                  s.save(c);
                  s.save(p);
                  tx.commit();
            }
            catch (Exception ex){
                  tx.rollback();
                  ex.printStackTrace();
            }
            HibernateUtil.closeSession();
      }
```

- 提取数据

提取数据时，只要提取 Person 对象，即可同时提取关联的 Card 对象。

代码 8.22 添加对象

```
private static void loadPerson(){
            Session s=HibernateUtil.getSession();
            Person p=(Person)s.get(Person.class, "00001");
            System.out.println(p.getName()+"的卡的描述是："+p.getCard().getCardDesc());
            HibernateUtil.closeSession();
      }
```

- 修改数据

修改数据包括 JavaBean 数据的修改和关联关系的修改，JavaBean 数据的修改和单表映射没有区别。关联关系的修改也可以采用对象方式进行。

代码 8.23 添加数据

```
private static void modifyPerson(){
            Session s=HibernateUtil.getSession();
            Person p=(Person)s.get(Person.class, "00001");
            Transaction tx=s.beginTransaction();
```

```
        try{
            p.setName("新名字");
            p.getCard().setCardDesc("卡的新描述");
            Card newCard=new Card();
            newCard.setCardId("card00002");
            newCard.setCardDesc("新卡");
            s.save(newCard);
            p.setCard(newCard);
            tx.commit();
        }
        catch (Exception ex){
            tx.rollback();
            ex.printStackTrace();
        }
        HibernateUtil.closeSession();
    }
```

- 删除数据

在删除 Person 对象时，会根据 cascade 值配置不同，这里在删除 Person 时会同时删除关联的 Card 对象。

代码 8.24 删除数据

```
private static void deletePerson(){
        Session s=HibernateUtil.getSession();
        Person p=(Person)s.get(Person.class, "00001");
        Transaction tx=s.beginTransaction();
        try{
            s.delete(p);
            tx.commit();
        }
        catch (Exception ex){
            tx.rollback();
            ex.printStackTrace();
        }
        HibernateUtil.closeSession();
    }
```

一对多单向关联

一对多关联是最常见的关联方式，这里介绍一对多单向关联在 Hibernate 中的实现。一对多单向关联一般通过集合映射的方式实现，这里以 Set 映射为例，介绍一种一对多单向关联方式。

图 8.11 产品和零件表

为此，可以设计两个 JavaBean：Product 和 Part 分别表示两张表的数据，并建立一对多单向关联。

代码 8.25 Product.java 和 Part.java

```
package com.firsthibernate.relation.one2many;
public class Part {
	private String partId;
	private String partName;
	public String getPartId() {	return partId;	}
	public void setPartId(String partId) {	this.partId = partId;	}
	public String getPartName() {	return partName;	}
	public void setPartName(String partName) {	this.partName = partName;	}
}
package com.firsthibernate.relation.one2many;
import java.util.Set;
public class Product {
	private String productId;
	private String productName;
	private Set<Part> parts;
	public String getProductId() {	return productId;	}
	public void setProductId(String productId) {	this.productId = productId;	}
	public String getProductName() {	return productName;	}
	public void setProductName(String productName) {this.productName = roductName;	}
	public Set<Part> getParts() {	return parts;	}
	public void setParts(Set<Part> parts) {	this.parts = parts;	}
}
```

从上面 JavaBean 的设计中可以看出，Part 类中没有 ProductId 对应的属性，而在 Product 类中增加了一个 Part 的集合属性，因此需要通过映射配置将这个外码对应的关系体现为 JavaBean 的关联，其映射文件配置如下。

代码 8.26 Product 和 Part 映射

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

```
    <class name="com.firsthibernate.relation.one2many.Part" table="Parts">
        <id name="partId" column="PartId" type="java.lang.String" unsaved-value="null">
            <generator class="assigned"/>
        </id>
        <property name="partName" column="PartName" type="java.lang.String" />
    </class>
    <class name="com.firsthibernate.relation.one2many.Product" table="Products">
        <id name="productId" column="ProductId" type="java.lang.String"
        unsaved-value="null">
            <generator class="assigned"/>
        </id>
        <property name="productName" column="ProductName" type="java.lang.String" />
        <set name="parts" cascade="delete">
        <key column="ProductId" />
        <one-to-many class="com.firsthibernate.relation.one2many.Part"/>
    </set>
    </class>
</hibernate-mapping>
```

在此配置下，可以对 Product 对象进行操作，下述代码演示增加和删除 Product 的基本方法。

代码 8.27 增加和删除 Product

```
private static void addProduct(){
    Session s=HibernateUtil.getSession();
    Transaction tx=s.beginTransaction();
    try{
        Product product=new Product();
        product.setProductId("0001");
        product.setProductName("测试产品");
        Part part1=new Part();
        part1.setPartId("pt001");
        part1.setPartName("部件 1");
        Part part2=new Part();
        part2.setPartId("pt002");
        part2.setPartName("部件 2");
        Set<Part> pts=new HashSet<Part>();
        pts.add(part1);
        pts.add(part2);
        s.save(part1);
        s.save(part2);
        product.setParts(pts);
        s.save(product);
        tx.commit();
    }
```

```
            catch (Exception ex){
                tx.rollback();
                ex.printStackTrace();
            }
            HibernateUtil.closeSession();
        }
    private static void deleteProduct(){
            Session s=HibernateUtil.getSession();
            Transaction tx=s.beginTransaction();
            try{
                Product product=(Product)s.get(Product.class, "0001");
                s.delete(product);
                tx.commit();
            }
            catch (Exception ex){
                tx.rollback();
                ex.printStackTrace();
            }
            HibernateUtil.closeSession();
        }
```

4.7 基本 HQL

Hibernate 配备了一种非常强大的查询语言，这种语言看上去很像 SQL。但是不要被语法结构上的相似所迷惑，HQL 是非常有意识的被设计为完全面向对象的查询，它可以理解如继承、多态和关联之类的概念。除了 Java 类与属性的名称外，查询语句对大小写并不敏感。所以 SeLeCT 与 sELEct 以及 SELECT 是相同的，但是 org.hibernate.eg.FOO 并不等价于 org.hibernate.eg.Foo，并且 foo.barSet 也不等价于 foo.BARSET。下面介绍一些基本的 HQL 语句，关于 HQL 语句的详细介绍请参考 Hibernate 文档。

4.7.1 from 子句

Hibernate 中最简单的查询语句的形式如下：

```
from com.firsthibernate.relation.one2many.Product
```

该子句简单地返回 com.firsthibernate.relation.one2many.Product 类的所有实例。通常不需要使用类的全限定名，因为 auto-import（自动引入）是缺省的情况。 所以可以只使用如下的简单写法：

```
from Product
```

可以为类指定一个别名，以便在查询语句的其它部分引用。

```
from Product as p
```

这个语句把别名 p 指定给类 Product 的实例，这样就可以在随后的查询中使用此别名。其中，关键字 as 是可选的。子句中可以同时出现多个类，其查询结果是产生一个笛卡儿积或产生跨表的连接。

```
from Formula, Parameter
from Formula as form, Parameter as param
```

4.7.2 select 子句

select 子句选择将哪些对象与属性返回到查询结果集中。例如：

代码 8.28 select 查询

```
private static void testSelectHql(){
        String hql="select p.productId,p.productName from Product p";
        Session s=HibernateUtil.getSession();
        Query qry=s.createQuery(hql);
        List list=qry.list();
        for(int i=0;i<list.size();i++){
            Object[] row=(Object[])list.get(i);
            System.out.println(row[0]+"--"+row[1]);
        }
        hql="select p.productName from Product p";
        qry=s.createQuery(hql);
        list=qry.list();
        for(int i=0;i<list.size();i++){
            String row=(String)list.get(i);
            System.out.println(row);
        }
        HibernateUtil.closeSession();
}
```

在本例中，第一个 HQL 语句的 select 后面有两个属性，则查询结构为一个 Object[]的 List；第二个 HQL 语句的 select 后面只有一个属性，则返回值为该属性所对应的类型的 List。

4.7.3 聚集函数

HQL 查询可以返回作用于属性之上的聚集函数的计算结果。

```
select max(p.productId), count(p)    from Product p
```

受支持的聚集函数如下：avg(...)， sum(...)， min(...)，max(...)，count(*)，count(...)，count(distinct ...)，count(all...)。

4.7.4 where 子句

where 子句允许将返回的实例列表的范围缩小，如果没有指定别名，则可以使用属性名来直接引用属性。

```
from Product where productName='产品 1'
```

如果指派了别名，需要使用完整的属性名。

```
from Product as p where p.productName='产品 1'
```

在 HQL 中和 SQL 中一样，可以将几张表进行连接（这里是进行对象连接）。例如：

```
from AuditLog log, Payment payment
where log.item.class = 'Payment' and log.item.id = payment.id
```

4.7.5 表达式

在 where 子句中允许使用的表达式（包括大多数可以在 SQL 使用的表达式种类）：数学运算符+、-、*、/；二进制比较运算符=、>=、<=、<>、!=、like；逻辑运算符 and、or、not；in、not in、between、is null、is not null、is empty、is not empty、member of、not member of；case、case ... when ... then ... else ... end；字符串连接符...||...、concat(...,...)；current_date()、current_time()、current_timestamp()；second(...)、minute(...)、hour(...)、day(...)、month(...)、year(...)，等等。

4.7.6 order by 子句

查询返回的列表(list)可以按照一个返回的类或组件（components)中的任何属性（property）进行排序。

```
from Person person order by person.card.cardId desc
```

可选的 asc 或 desc 关键字指明了按照升序或降序进行排序。

4.7.7 group by 子句

一个返回聚集值的查询可以按照一个返回的类或组件中的任何属性进行分组：

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
```

4.7.8 子查询

对于支持子查询的数据库，Hibernate 支持在查询中使用子查询。一个子查询必须被圆括号包围起来（经常是 SQL 聚集函数的圆括号）。甚至相互关联的子查询（引用到外部查询中的别名的子查询）也是允许的。

```
from Cat as fatcat
where fatcat.weight > (
    select avg(cat.weight) from DomesticCat cat
)
```