

JSP 学习——全面解析 JDBC

Java 数据库连接体系结构是用于 Java 应用程序连接数据库的标准方法。JDBC 对 Java 程序员而言是 API，对实现与数据库连接的服务提供商而言是接口模型。作为 API，JDBC 为程序开发提供标准的接口，并为数据库厂商及第三方中间件厂商实现与数据库的连接提供了标准方法。JDBC 使用已有的 SQL 标准并支持与其它数据库连接标准，如 ODBC 之间的桥接。JDBC 实现了所有这些面向标准的目标并且具有简单、严格类型定义且高性能实现的接口。

如何选择合适的 JDBC 产品？

有关 JDBC 最新的信息，有兴趣的读者可以查阅 JDBC 的官方网站——即 JavaSoft 的主页，其 URL 为：<http://Java.sun.com/products/jdbc>

1. JavaSoft 框架

JavaSoft 提供三种 JDBC 产品组件，它们是 Java 开发工具包（JDK）的组成部份：JDBC 驱动程序管理器、JDBC 驱动程序测试工具包和 JDBC-ODBC 桥。

JDBC 驱动程序管理器是 JDBC 体系结构的支柱。它实际上很小，也很简单；其主要作用是把 Java 应用程序连接到正确的 JDBC 驱动程序上，然后即退出。

JDBC 驱动程序测试工具包为使 JDBC 驱动程序运行您的程序提供一定的可信度。只有通过 JDBC 驱动程序测试的驱动程序才被认为是符合 JDBC 标准 TM 的。

JDBC-ODBC 桥使 ODBC 驱动程序可被用作 JDBC 驱动程序。它的实现为 JDBC 的快速发展提供了一条途径，其长远目标提供一种访问某些不常见的 DBMS（如果对这些不常见的 DBMS 未实现 JDBC）的方法。

2. JDBC 驱动程序的类型

目前比较常见的 JDBC 驱动程序可分为以下四个种类：

- (1) JDBC-ODBC 桥加 ODBC 驱动程序

JavaSoft 桥产品利用 ODBC 驱动程序提供 JDBC 访问。注意，必须将 ODBC 二进制代码（许多情况下还包括数据库客户机代码）加载到使用该驱动程序的每个客户机上。因此，这种类型的驱动程序最适合于企业网（这种网络上客户机的安装不是主要问题），或者是用 Java 编写的三层结构的应用程序服务器代码。

(2) 本地 API

这种类型的驱动程序把客户机 API 上的 JDBC 调用转换为 Oracle、Sybase、Informix、DB2 或其它 DBMS 的调用。注意，象桥驱动程序一样，这种类型的驱动程序要求将某些二进制代码加载到每台客户机上。

(3) JDBC 网络纯 Java 驱动程序

这种驱动程序将 JDBC 转换为与 DBMS 无关的网络协议，之后这种协议又被某个服务器转换为一种 DBMS 协议。这种网络服务器中间件能够将它的纯 Java 客户机连接到多种不同的数据库上。所用的具体协议取决于提供者。通常，这是最为灵活的 JDBC 驱动程序。有可能所有这种解决方案的提供者都提供适合于 Intranet 用的产品。为了使这些产品也支持 Internet 访问，它们必须处理 Web 所提出的安全性、通过防火墙的访问等方面的额外要求。几家提供者正将 JDBC 驱动程序加到他们现有的数据库中间件产品中。

(4) 本地协议纯 Java 驱动程序

这种类型的驱动程序将 JDBC 调用直接转换为 DBMS 所使用的网络协议。这将允许从客户机机器上直接调用 DBMS 服务器，是 Intranet 访问的一个很实用的解决方法。由于许多这样的协议都是专用的，因此数据库提供者自己将是主要来源，有几家提供者已在着手做这件事了。

据专家预计第 (3)、(4) 类驱动程序将成为从 JDBC 访问数据库的首方法。第 (1)、(2) 类驱动程序在直接的纯 Java 驱动程序还没有上市前会作为过渡方案来使用。对第 (1)、(2) 类驱动程序可能会有一些变种，这些变种要求有连接器，但通常这些是更加不可取的解决方案。第 (3)、(4) 类驱动程序提供了 Java 的所有优点，包括自动安装（例如，通过使用 JDBC 驱动程序的 applet 来下载该驱动程序）。

3. JDBC驱动程序的获取

目前已有几十个 (1) 类的驱动程序, 即可与 Javasoft 桥联合使用的 ODBC 驱动程序的驱动程序。有大约十多个属于种类 (2) 的驱动程序是以 DBMS 的本地 API 为基础编写的。只有几个属于种类 (3) 的驱动程序, 其首批提供者是 SCO、OpenHorizon、Visigenic 和 WebLogic。此外, JavaSoft 和数据库连接的领先提供者 Intersolv 还合作研制了 JDBC-ODBC 桥和 JDBC 驱动程序测试工具包。

如何建立JDBC连接?

Connection 对象代表与数据库的连接。连接过程包括所执行的 SQL 语句和在该连接上所返回的结果。一个应用程序可与单个数据库有一个或多个连接, 或者可与许多数据库有连接。

1. 打开连接

与数据库建立连接的标准方法是调用 DriverManager.getConnection 方法。该方法接受含有某个 URL 的字符串。DriverManager 类 (即所谓的 JDBC 管理层) 将尝试找到可与那个 URL 所代表的数据库进行连接的驱动程序。DriverManager 类存有已注册的 Driver 类的清单。当调用方法 getConnection 时, 它将检查清单中的每个驱动程序, 直到找到可与 URL 中指定的数据库进行连接的驱动程序为止。Driver 的方法 connect 使用这个 URL 来建立实际的连接。

用户可绕过 JDBC 管理层直接调用 Driver 方法。这在以下特殊情况下将很有用: 当两个驱动器可同时连接到数据库中, 而用户需要明确地选用其中特定的驱动器。但一般情况下, 让 DriverManager 类处理打开连接这种事将更为简单。

下述代码显示如何打开一个与位于 URL "jdbc: odbc: wombat" 的数据库的连接。所用的用户标识符为 "freely", 口令为 "ec":

```
String url = "jdbc: odbc: wombat";  
Connection con = DriverManager.getConnection (url, "freely", "ec");
```

2. 一般用法的URL

相信自己-----总会成功

由于 URL 常引起混淆,我们将先对一般 URL 作简单说明,然后再讨论 JDBCURL。URL (统一资源定位符) 提供在 Internet 上定位资源所需的信息。可将它想象为一个地址。URL 的第一部份指定了访问信息所用的协议,后面总是跟着冒号。常用的协议有"ftp" (代表"文件传输协议") 和"http" (代表"超文本传输协议")。如果协议是"file", 表示资源是在某个本地文件系统上而非在 Internet 上(下例用于表示我们所描述的部分; 它并非 URL 的组成部分)。

ftp: //Javasoft.com/docs/JDK-1_apidocs.zip

http: //Java.sun.com/products/jdk/CurrentRelease

file: /home/haroldw/docs/books/tutorial/summary.html

URL 的其余部份(冒号后面的)给出了数据资源所处位置的有关信息。如果协议是 file, 则 URL 的其余部份是文件的路径。对于 ftp 和 http 协议, URL 的其余部份标识了主机并可选地给出某个更详尽的地址路径。例如, 以下是 JavaSoft 主页的 URL。该 URL 只标识了主机: http: //Java.sun.com。从该主页开始浏览, 就可以进到许多其它的网页中, 其中之一就是 JDBC 主页。JDBC 主页的 URL 更为具体, 它具体表示为:

http: //Java.sun.com/products/jdbc

3. JDBC URL

JDBC URL 提供了一种标识数据库的方法, 可以使相应的驱动程序能识别该数据库并与之建立连接。实际上, 驱动程序程序员将决定用什么 JDBC URL 来标识特定的驱动程序。用户不必关心如何来形成 JDBC URL; 他们只须使用与所用的驱动程序一起提供的 URL 即可。JDBC 的作用是提供某些约定, 驱动程序程序员在构造他们的 JDBC URL 时应该遵循这些约定。

由于 JDBC URL 要与各种不同的驱动程序一起使用, 因此这些约定应非常灵活。首先, 它们应允许不同的驱动程序使用不同的方案来命名数据库。例如, odbc 子协议允许(但并不是要求) URL 含有属性值。

其次, JDBC URL 应允许驱动程序程序员将一切所需的信息编入其中。这样就可以让要与给定数据库对话的 applet 打开数据库连接, 而无须要求用户去做任何系统管理工作。

最后, JDBC URL 应允许某种程度的间接性。也就是说, JDBC URL 可指向逻辑主机或数据库名,而这种逻辑主机或数据库名将由网络命名系统动态地转换为实际的名称。这可以使系统管理员不必将特定主机声明为 JDBC 名称的一部份。网络命名服务(例如 DNS、NIS 和 DCE)有多种,而对于使用哪种命名服务并无限制。

JDBC URL 的标准语法如下所示。它由三部分组成,各部分间用冒号分隔:

jdbc: <子协议>:<子名称>:<子名称> JDBC URL 的三个部分可分解如下:

(1) jdbc 协议: JDBC URL 中的协议总是 jdbc。

(2) <子协议>: 驱动程序名或数据库连接机制(这种机制可由一个或多个驱动程序支持)的名称。子协议名的典型示例是"odbc",该名称是为用于指定 ODBC 风格的数据资源名称的 URL 专门保留的。例如,为了通过 JDBC-ODBC 桥来访问某个数据库,可以用如下所示的 URL: jdbc:odbc:book。本例中,子协议为"odbc",子名称"book"是本地 ODBC 数据资源。如果要用网络命名服务(这样 JDBC URL 中的数据库名称不必是实际名称),则命名服务可以作为子协议。例如,可用如下所示的 URL: jdbc:dcenaming:accounts。本例中,该 URL 指定了本地 DCE 命名服务应该将数据库名称"accounts"解析为更为具体的可用于连接真实数据库的名称。

(3) <子名称>: 种标识数据库的方法。子名称可以依不同的子协议而变化。它还可以有子名称的子名称(含有驱动程序程序员所选的任何内部语法)。使用子名称的目的是为定位数据库提供足够的信息。前例中,因为 ODBC 将提供其余部份的信息,因此用"book"就已足够。然而,位于远程服务器上的数据库需要更多的信息。例如,如果数据库是通过 Internet 来访问的,则在 JDBC URL 中应将网络地址作为子名称的一部份包括进去,且必须遵循如下所示的标准 URL 命名约定://主机名:端口/子协议。

假设"dbnet"是个用于将某个主机连接到 Internet 上的协议,则 JDBC URL 应为:jdbc:dbnet://wombat:356/fred。

4. "odbc"子协议

子协议 odbc 是一种特殊情况。它是为用于指定 ODBC 风格的数据资源名称的 URL 而保留的,并具有下列特性:允许在子名称(数据资源名称)后面指定任意多个属性值。odbc 子协议的完整语法为:

相信自己-----总会成功

`jdbc: odbc: <数据资源名称>[; <属性名>=<属性值>]`, 因此, 以下都是合法的 `jdbc:` `odbc` 名称:

```
jdbc: odbc: qeor7
```

```
jdbc: odbc: wombat
```

```
jdbc: odbc: wombat; CacheSize=20; ExtensionCase=LOWER
```

```
jdbc: odbc: qeora; UID=kgh; PWD=foeey
```

5. 注册子协议

驱动程序程序员可保留某个名称以将之用作 JDBC URL 的子协议名。当 `DriverManager` 类将此名称加到已注册的驱动程序清单中时, 为之保留该名称的驱动程序应能识别该名称并与它所标识的数据库建立连接。例如, `odbc` 是为 JDBC-ODBC 桥而保留的。假设有个 `Miracle` 公司, 它可能会将"miracle"注册为连接到其 `Miracle DBMS` 上的 JDBC 驱动程序的子协议, 从而使其他人都无法使用这个名称。

JavaSoft 目前作为非正式代理负责注册 JDBC 子协议名称。要注册某个子协议名称, 请发送电子邮件到下述地址: `jdbc@wombat.eng.sun.com`。

6. 发送 SQL 语句

连接一旦建立, 就可用来向它所涉及的数据库传送 SQL 语句。JDBC 对可被发送的 SQL 语句类型不加任何限制。这就提供了很大的灵活性, 即允许使用特定的数据库语句或甚至于非 SQL 语句。然而, 它要求用户自己负责确保所涉及的数据库可以处理所发送的 SQL 语句, 否则将自食其果。例如, 如果某个应用程序试图向不支持储存程序的 `DBMS` 发送储存程序调用, 就会失败并将抛出异常。JDBC 要求驱动程序应至少能提供 ANSI SQL-2 Entry Level 功能才可算是符合 JDBC 标准 TM 的。这意味着用户至少可信赖这一标准级别的功能。

JDBC 提供了三个类, 用于向数据库发送 SQL 语句。`Connection` 接口中的三个方法可用于创建这些类的实例。下面列出这些类及其创建方法:

(1) **Statement:** 由方法 `createStatement` 所创建。`Statement` 对象用于发送简单的 SQL 语句。

(2) `PreparedStatement`: 由方法 `prepareStatement` 所创建。`PreparedStatement` 对象用于发送带有一个或多个输入参数 (IN 参数) 的 SQL 语句。`PreparedStatement` 拥有一组方法, 用于设置 IN 参数的值。执行语句时, 这些 IN 参数将被送到数据库中。`PreparedStatement` 的实例扩展了 `Statement`, 因此它们都包括了 `Statement` 的方法。`PreparedStatement` 对象有可能比 `Statement` 对象的效率更高, 因为它已被预编译过并存放在那以供将来使用。

(3) `CallableStatement`: 由方法 `prepareCall` 所创建。`CallableStatement` 对象用于执行 SQL 储存程序——一组可通过名称来调用 (就象函数的调用那样) 的 SQL 语句。`CallableStatement` 对象从 `PreparedStatement` 中继承了用于处理 IN 参数的方法, 而且还增加了用于处理 OUT 参数和 INOUT 参数的方法。

不过通常来说 `createStatement` 方法用于简单的 SQL 语句 (不带参数)、`prepareStatement` 方法用于带一个或多个 IN 参数的 SQL 语句或经常被执行的简单 SQL 语句, 而 `prepareCall` 方法用于调用已储存过程。

7. 事务

事务由一个或多个这样的语句组成: 这些语句已被执行、完成并被提交或还原。当调用方法 `commit` 或 `rollback` 时, 当前事务即告就结束, 另一个事务随即开始。缺省情况下, 新连接将处于自动提交模式。也就是说, 当执行完语句后, 将自动对那个语句调用 `commit` 方法。这种情况下, 由于每个语句都是被单独提交的, 因此一个事务只由一个语句组成。如果禁用自动提交模式, 事务将要等到 `commit` 或 `rollback` 方法被显式调用时才结束, 因此它将包括上一次调用 `commit` 或 `rollback` 方法以来所有执行过的语句。对于第二种情况, 事务中的所有语句将作为组来提交或还原。

方法 `commit` 使 SQL 语句对数据库所做的任何更改成为永久性的, 它还将释放事务持有的全部锁。而方法 `rollback` 将弃去那些更改。有时用户在另一个更改生效前不想让此更改生效。这可通过禁用自动提交并将两个更新组合在一个事务中来达到。如果两个更新都是成功, 则调用 `commit` 方法, 从而使两个更新结果成为永久性的; 如果其中之一或两个更新都失败了, 则调用 `rollback` 方法, 以将值恢复为进行更新之前的值。

大多数 JDBC 驱动程序都支持事务。事实上，符合 JDBC 的驱动程序必须支持事务。DatabaseMetaData 给出的信息描述 DBMS 所提供的事务支持水平。

8. 事务隔离级别

如果 DBMS 支持事务处理，它必须有某种途径来管理两个事务同时对一个数据库进行操作时可能发生的冲突。用户可指定事务隔离级别，以指明 DBMS 应该花多大精力来解决潜在冲突。例如，当事务更改了某个值而第二个事务却在更改被提交或还原前读取该值时该怎么办。

假设第一个事务被还原后，第二个事务所读取的更改值将是无效的，那么是否可允许这种冲突？JDBC 用户可用以下代码来指示 DBMS 允许在值被提交前读取该值（"dirty 读取"），其中 con 是当前连接：

```
con.setTransactionIsolation ( TRANSACTION_READ_UNCOMMITTED ) ;
```

事务隔离级别越高，为避免冲突所花的精力也就越多。Connection 接口定义了五级，其中最低级别指定了根本就不支持事务，而最高级别则指定当事务在对某个数据库进行操作时，任何其它事务不得对那个事务正在读取的数据进行任何更改。通常，隔离级别越高，应用程序执行的速度也就越慢（由于用于锁定的资源耗费增加了，而用户间的并发操作减少了）。在决定采用什么隔离级别时，开发人员必须在性能需求和数据一致性需求之间进行权衡。当然，实际所能支持的级别取决于所涉及的 DBMS 的功能。

当创建 Connection 对象时，其事务隔离级别取决于驱动程序，但通常是所涉及的数据库的缺省值。用户可通过调用 setIsolationLevel 方法来更改事务隔离级别。新的级别将在该连接过程的剩余时间内生效。要想只改变一个事务的事务隔离级别，必须在该事务开始前进行设置，并在该事务结束后进行复位。我们不提倡在事务的中途对事务隔离级别进行更改，因为这将立即触发 commit 方法的调用，使在此之前所作的任何更改变成永久性的。

JDBC驱动管理内幕是怎么样的？

DriverManager 类是 JDBC 的管理层，作用于用户和驱动程序之间。它跟踪可用的驱动程序，并在数据库和相应驱动程序之间建立连接。另外，DriverManager 类也处理诸如驱动程序登录时间限制及登录和跟踪消息的显示等事务。

相信自己-----总会成功

对于简单的应用程序，一般程序员需要在此类中直接使用的唯一方法是 `DriverManager.getConnection`。正如名称所示，该方法将建立与数据库的连接。JDBC 允许用户调用 `DriverManager` 的方法 `getDriver`、`getDrivers` 和 `registerDriver` 及 `Driver` 的方法 `connect`。但多数情况下，让 `DriverManager` 类管理建立连接的细节为上策。

1. 跟踪可用驱动程序

`DriverManager` 类包含一系列 `Driver` 类，它们已通过调用方法 `DriverManager.registerDriver` 对自己进行了注册。所有 `Driver` 类都必须包含有一个静态部分。它创建该类的实例，然后在加载该实例时 `DriverManager` 类进行注册。这样，用户正常情况下将不会直接调用 `DriverManager.registerDriver`；而是在加载驱动程序时由驱动程序自动调用。加载 `Driver` 类，然后自动在 `DriverManager` 中注册的方式有两种：

(1) 调用方法 `Class.forName`

这将显式地加载驱动程序类。由于这与外部设置无关，因此推荐使用这种加载驱动程序的方法。以下代码加载类 `acme.db.Driver`：`Class.forName("acme.db.Driver")`。

如果将 `acme.db.Driver` 编写为加载时创建实例，并调用以该实例为参数的 `DriverManager.registerDriver`（本该如此），则它在 `DriverManager` 的驱动程序列表中，并可用于创建连接。

(2) 将驱动程序添加到 `Java.lang.System` 的属性 `jdbc.drivers` 中

这是一个由 `DriverManager` 类加载的驱动程序类名的列表，由冒号分隔：初始化 `DriverManager` 类时，它搜索系统属性 `jdbc.drivers`，如果用户已输入了一个或多个驱动程序，则 `DriverManager` 类将试图加载它们。以下代码说明程序员如何在 `~/.hotJava/properties` 中输入三个驱动程序类（启动时，`HotJava` 将把它加载到系统属性列表中）：

```
jdbc.drivers=foo.bah.Driver: wombat.sql.Driver: bad.test.ourDriver;
```

对 `DriverManager` 方法的第一次调用将自动加载这些驱动程序类。注意：加载驱动程序的第二种方法需要持久的预设环境。如果对这一点不能保证，则调用方法 `Class.forName` 显式地加载

相信自己-----总会成功

每个驱动程序就显得更为安全。这也是引入特定驱动程序的方法，因为一旦 `DriverManager` 类被初始化，它将不再检查 `jdbc.drivers` 属性列表。

在以上两种情况中，新加载的 `Driver` 类都要通过调用 `DriverManager.registerDriver` 类进行自我注册。如上所述，加载类时将自动执行这一过程。

由于安全方面的原因，JDBC 管理层将跟踪哪个类加载器提供哪个驱动程序。这样，当 `DriverManager` 类打开连接时，它仅使用本地文件系统或与发出连接请求的代码相同的类加载器提供的驱动程序。

2. 建立连接

加载 `Driver` 类并在 `DriverManager` 类中注册后，它们即可用来与数据库建立连接。当调用 `DriverManager.getConnection` 方法发出连接请求时，`DriverManager` 将检查每个驱动程序，查看它是否可以建立连接。

有时可能有多个 JDBC 驱动程序可以与给定的 URL 连接。例如，与给定远程数据库连接时，可以使用 JDBC-ODBC 桥驱动程序、JDBC 到通用网络协议驱动程序或数据库厂商提供的驱动程序。在这种情况下测试驱动程序的顺序至关重要，因为 `DriverManager` 将使用它所找到的第一个可以成功连接到给定 URL 的驱动程序。

首先 `DriverManager` 试图按注册的顺序使用每个驱动程序（`jdbc.drivers` 中列出的驱动程序总是先注册）。它将跳过代码不可信任的驱动程序，除非加载它们的源与试图打开连接的代码的源相同。它通过轮流在每个驱动程序上调用方法 `Driver.connect`，并向它们传递用户开始传递给方法 `DriverManager.getConnection` 的 URL 来对驱动程序进行测试，然后连接第一个认出该 URL 的驱动程序。这种方法初看起来效率不高，但由于不可能同时加载数十个驱动程序，因此每次连接实际只需几个过程调用和字符串比较。

以下代码是通常情况下用驱动程序（例如 JDBC-ODBC 桥驱动程序）建立连接所需所有步骤的示例：

```
Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver"); //加载驱动程序  
String url = "jdbc: odbc: fred";  
DriverManager.getConnection (url, "userID", "passwd");
```

如何利用JDBC发送SQL语句?

Statement 对象用于将 SQL 语句发送到数据库中。实际上有三种 Statement 对象, 它们都作为在给定连接上执行 SQL 语句的容器: Statement、PreparedStatement (它从 Statement 继承而来) 和 CallableStatement (它从 PreparedStatement 继承而来)。它们都专用于发送特定类型的 SQL 语句: Statement 对象用于执行不带参数的简单 SQL 语句; PreparedStatement 对象用于执行带或不带 IN 参数的预编译 SQL 语句; CallableStatement 对象用于执行对数据库已存储过程的调用。

Statement 接口提供了执行语句和获取结果的基本方法; PreparedStatement 接口添加了处理 IN 参数的方法; 而 CallableStatement 添加了处理 OUT 参数的方法。

1. 创建Statement对象

建立了到特定数据库的连接之后, 就可用该连接发送 SQL 语句。Statement 对象用 Connection 的方法 createStatement 创建, 如下列代码段中所示:

```
Connection con = DriverManager.getConnection (url, "sunny", "");  
Statement stmt = con.createStatement ();
```

为了执行 Statement 对象, 被发送到数据库的 SQL 语句将被作为参数提供给 Statement 的方法:

```
ResultSet rs = stmt.executeQuery ("SELECT a, b, c FROM Table2");
```

2. 使用Statement对象执行语句

Statement 接口提供了三种执行 SQL 语句的方法: executeQuery、executeUpdate 和 execute。使用哪一个方法由 SQL 语句所产生的内容决定。

相信自己-----总会成功

方法 `executeQuery` 用于产生单个结果集的语句，例如 `SELECT` 语句。方法 `executeUpdate` 用于执行 `INSERT`、`UPDATE` 或 `DELETE` 语句以及 SQL DDL（数据定义语言）语句，例如 `CREATE TABLE` 和 `DROP TABLE`。`INSERT`、`UPDATE` 或 `DELETE` 语句的效果是修改表中零行或多行中的一列或多列。`executeUpdate` 的返回值是一个整数，指示受影响的行数（即更新计数）。对于 `CREATE TABLE` 或 `DROP TABLE` 等不操作行的语句，`executeUpdate` 的返回值总为零。

执行语句的所有方法都将关闭所调用的 `Statement` 对象的当前打开结果集（如果存在）。这意味着在重新执行 `Statement` 对象之前，需要完成对当前 `ResultSet` 对象的处理。应注意，继承了 `Statement` 接口中所有方法的 `PreparedStatement` 接口都有自己的 `executeQuery`、`executeUpdate` 和 `execute` 方法。`Statement` 对象本身不包含 SQL 语句，因而必须给 `Statement.execute` 方法提供 SQL 语句作为参数。`PreparedStatement` 对象并不需要 SQL 语句作为参数提供给这些方法，因为它们已经包含预编译 SQL 语句。

`CallableStatement` 对象继承这些方法的 `PreparedStatement` 形式。对于这些方法的 `PreparedStatement` 或 `CallableStatement` 版本，使用查询参数将抛出 `SQLException`。

3. 语句完成

当连接处于自动提交模式时，其中所执行的语句在完成时将自动提交或还原。语句在已执行且所有结果返回时，即认为已完成。对于返回一个结果集的 `executeQuery` 方法，在检索完 `ResultSet` 对象的所有行时该语句完成。对于方法 `executeUpdate`，当它执行时语句即完成。但在少数调用方法 `execute` 的情况中，在检索所有结果集或它生成的更新计数之后语句才完成。

有些 DBMS 将已存储过程中的每条语句视为独立的语句；而另外一些则将整个过程视为一个复合语句。在启用自动提交时，这种差别就变得非常重要，因为它影响什么时候调用 `commit` 方法。在前一种情况中，每条语句单独提交；在后一种情况中，所有语句同时提交。

4. 关闭Statement对象

`Statement` 对象将由 Java 垃圾收集程序自动关闭。而作为一种好的编程风格，应在不需要 `Statement` 对象时显式地关闭它们。这将立即释放 DBMS 资源，有助于避免潜在的内存问题。

5. 使用方法execute

相信自己-----总会成功

`execute` 方法应该仅在语句能返回多个 `ResultSet` 对象、多个更新计数或 `ResultSet` 对象与更新计数的组合时使用。当执行某个已存储过程或动态执行未知 SQL 字符串（即应用程序程序员在编译时未知）时，有可能出现多个结果的情况，尽管这种情况很少见。例如，用户可能执行一个已存储过程，并且该已存储过程可执行更新，然后执行选择，再进行更新，再进行选择，等等。通常使用已存储过程的人应知道它所返回的内容。

因为方法 `execute` 处理非常规情况，所以获取其结果需要一些特殊处理并不足为怪。例如，假定已知某个过程返回两个结果集，则在使用方法 `execute` 执行该过程后，必须调用方法 `getResultSet` 获得第一个结果集，然后调用适当的 `getXXX` 方法获取其中的值。要获得第二个结果集，需要先调用 `getMoreResults` 方法，然后再调用 `getResultSet` 方法。如果已知某个过程返回两个更新计数，则首先调用方法 `getUpdateCount`，然后调用 `getMoreResults`，并再次调用 `getUpdateCount`。

对于不知道返回内容，则情况更为复杂。如果结果是 `ResultSet` 对象，则方法 `execute` 返回 `true`；如果结果是 `Javaint`，则返回 `false`。如果返回 `int`，则意味着结果是更新计数或执行的语句是 DL 命令。在调用方法 `execute` 之后要做的第一件事情是调用 `getResultSet` 或 `getUpdateCount`。调用方法 `getResultSet` 可以获得两个或多个 `ResultSet` 对象中第一个对象；或调用方法 `getUpdateCount` 可以获得两个或多个更新计数中第一个更新计数的内容。

当 SQL 语句的结果不是结果集时，则方法 `getResultSet` 将返回 `null`。这可能意味着结果是一个更新计数或没有其它结果。在这种情况下，判断 `null` 真正含义的唯一方法是调用方法 `getUpdateCount`，它将返回一个整数。这个整数为调用语句所影响的行数；如果为 -1 则表示结果是结果集或没有结果。如果方法 `getResultSet` 已返回 `null`（表示结果不是 `ResultSet` 对象），则返回值 -1 表示没有其它结果。也就是说，当下列条件为真时表示没有结果（或没有其它结果）：

```
( (stmt.getResultSet () == null) && (stmt.getUpdateCount () == -1) )
```

如果已经调用方法 `getResultSet` 并处理了它返回的 `ResultSet` 对象，则有必要调用方法 `getMoreResults` 以确定是否有其它结果集或更新计数。如果 `getMoreResults` 返回 `true`，则需要再次调用 `getResultSet` 来检索下一个结果集。如上所述，如果 `getResultSet` 返回 `null`，则需要调用 `getUpdateCount` 来检查 `null` 是表示结果为更新计数还是表示没有其它结果。

当 `getMoreResults` 返回 `false` 时，它表示该 SQL 语句返回一个更新计数或没有其它结果。因此需要调用方法 `getUpdateCount` 来检查它是哪一种情况。在这种情况下，当下列条件为真时表示没有其它结果：

```
( (stmt.getMoreResults () ==false) && (stmt.getUpdateCount () ==-1) )
```

下面的代码演示了一种方法用来确认已访问调用方法 `execute` 所产生的全部结果集和更新计数：

```
stmt.execute (queryStringWithUnknownResults) ;
while (true) {
introwCount=stmt.getUpdateCount () ;
if (rowCount>0) {//它是更新计数
System.out.println ("Rows changed="+count) ;
stmt.getMoreResults () ;
continue;
}
if (rowCount==0) {//DDL 命令或 0 个更新
System.out.println ("No rows changed or statement was DDL command") ;
stmt.getMoreResults () ;
continue;
}
//执行到这里，证明有一个结果集
//或没有其它结果
ResultSet rs=stmt.getResultSet () ;
if (rs!=null) {
...//使用元数据获得关于结果集列的信息
while (rs.next () ) {
...//处理结果
stmt.getMoreResults () ;
```

相信自己-----总会成功

```
continue;  
}
```

如何获得SQL语句的执行结果?

ResultSet 包含符合 SQL 语句中条件的所有行，并且它通过一套 get 方法（这些 get 方法可以访问当前行中的不同列）提供了对这些行中数据的访问。ResultSet.next 方法用于移动到 ResultSet 中的下一行，使下一行成为当前行。

下面的代码段是执行 SQL 语句的示例。该 SQL 语句将返回行集合，其中列 1 为 int，列 2 为 String，而列 3 则为字节数组：

```
Java.sql.Statement stmt=conn.createStatement ( ) ;  
ResultSet r=stmt.executeQuery ("SELECT a, b, c FROM Table1") ;  
while (r.next ( ) ) {  
//打印当前行的值。  
Int i=r.getInt ("a") ;  
String s=r.getString ("b") ;  
Float f=r.getFloat ("c") ;  
System.out.println ("ROW="+i+" "+s+" "+f) ;  
}
```

1. 行和光标

ResultSet 维护指向其当前数据行的光标。每调用一次 next 方法，光标向下移动一行。

最初它位于第一行之前，因此第一次调用 next 将把光标置于第一行上，使它成为当前行。随着每次调用 next 导致光标向下移动一行，按照从上至下的次序获取 ResultSet 行。

在 ResultSet 对象或其父辈 Statement 对象关闭之前，光标一直保持有效。在 SQL 中，结果表的光标是有名字的。如果数据库允许定位更新或定位删除，则需要将光标的名字作为参数提供给更新或删除命令。可通过调用方法 getCursorName 获得光标名。

DatabaseMetaData.supportsPositionedDelete 和 supportsPositionedUpdate 方法来检查特定连接是否支持这些操作。当 DBMS 支持定位更新和删除操作时，DBMS/驱动程序必须确保适当锁定选定行，以使定位更新不会导致更新异常或其它并发问题。

2. 列

方法 getXXX 提供了获取当前行中某列值的途径。在每一行内，可按任何次序获取列值。但为了保证可移植性，应该从左至右获取列值，并且一次性地读取列值。

列名或列号可用于标识要从中获取数据的列。例如，如果 ResultSet 对象 rs 的第二列名为 "title"，并将值存储为字符串，则下列任一代码将获取存储在该列中的值：

```
String s=rs.getString ("title") ;
```

```
String s=rs.getString (2) ;
```

注意列是从左至右编号的，并且从列 1 开始。同时，用作 getXXX 方法的输入的列名不区分大小写。

提供使用列名这个选项的目的是为了让在查询中指定列名的用户可使用相同的名字作为 getXXX 方法的参数。另一方面，如果 select 语句未指定列名（例如在 "select * from table1" 中或是导出的时），则应该使用列号。这些情况下，用户将无法确切知道列名。

有些情况下，SQL 查询返回的结果集中可能有多个列具有相同的名字。如果列名用作 getXXX 方法的参数，则 getXXX 将返回第一个匹配列名的值。因而，如果多个列具有相同的名字，则需要使用列索引来确保检索了正确的列值。这时，使用列号效率要稍微高一些。

关于 ResultSet 中列的信息，可通过调用方法 ResultSet.getMetaData 得到。返回的 ResultSetMetaData 对象将给出其 ResultSet 对象各列的编号、类型和属性。

如果列名已知，但不知其索引，则可用方法 findColumn 得到其列号。

3. 数据类型和转换

对于 getXXX 方法，JDBC 驱动程序试图将基本数据转换成指定 Java 类型，

相信自己-----总会成功

然后返回适合的 Java 值。例如，如果 getXXX 方法为 getString，而基本数据库中数据类型为 VARCHAR，则 JDBC 驱动程序将把 VARCHAR 转换成 JavaString。getString 的返回值将为 JavaString 对象。

4. 对非常大的行值使用流

ResultSet 可以获取任意大的 LONGVARBINARY 或 LONGVARCHAR 数据。方法 getBytes 和 getString 将数据返回为大的块（最大为 Statement.getMaxFieldSize 的返回值）。但是，以较小的固定块获取非常大的数据可能会更方便，而这可通过让 ResultSet 类返回 Java.io.InputStream 来完成。从该流中可分块读取数据。注意：必须立即访问这些流，因为在下一次对 ResultSet 调用 getXXX 时它们将自动关闭（这是由于基本实现对大块数据访问有限制）。

JDBC API 具有三个获取流的方法，分别具有不同的返回值：

- getBinaryStream：返回只提供数据库原字节而不进行任何转换的流。
- getAsciiStream 返回提供单字节 ASCII 字符的流。
- getUnicodeStream 返回提供双字节 Unicode 字符的流。

注意：它不同于 Java 流，后者返回无类型字节并可（例如）通用于 ASCII 和 Unicode 字符。

下列代码演示了 getAsciiStream 的用法：

```
Java.sql.Statement stmt=con.createStatement ( ) ;
ResultSet r=stmt.executeQuery ("SELECT x FROM Table2") ;
//现在以 4K 块大小获取列 1 结果:
byte buff=new byte[4096];
while ( r.next ( ) ) {
Java.io.InputStream fin=r.getAsciiStream (1) ;
for ( ; ; ) {
int size=fin.read ( buff ) ;
if ( size== -1 ) { //到达流末尾
break;
}
```

相信自己-----总会成功

```
}  
//将新填充的缓冲区发送到 ASCII 输出流:  
output.write (buff, 0, size) ;  
}  
}
```

5. NULL结果值

要确定给定结果值是否是 JDBC NULL，必须先读取该列，然后使用 `ResultSet.wasNull` 方法检查该次读取是否返回 JDBC NULL。

当使用 `ResultSet.getXXX` 方法读取 JDBC NULL 时，方法 `wasNull` 将返回下列值之一：

(1) Javanull 值

对于返回 Java 对象的 `getXXX` 方法（例如 `getString`、`getBigDecimal`、`getBytes`、`getDate`、`getTime`、`getTimestamp`、`getAsciiStream`、`getUnicodeStream`、`getBinaryStream`、`getObject` 等）。

(2) 零值：对于 `getByte`、`getShort`、`getInt`、`getLong`、`getFloat` 和 `getDouble`。

(3) false 值：对于 `getBoolean`。

6. 可选结果集或多结果集

通常使用 `executeQuery`（它返回单个 `ResultSet`）或 `executeUpdate`（它可用于任何数据库修改语句，并返回更新行数）可执行 SQL 语句。但有些情况下，应用程序在执行语句之前不知道该语句是否返回结果集。此外，有些已存储过程可能返回几个不同的结果集和/或更新计数。

为了适应这些情况，JDBC 提供了一种机制，允许应用程序执行语句，然后处理由结果集和更新计数组成的任意集合。这种机制的原理是首先调用一个完全通用的 `execute` 方法，然后调用另外三个方法，`getResultSet`、`getUpdateCount` 和 `getMoreResults`。这些方法允许应用程序一个一个地研究语句结果，并确定给定结果是 `ResultSet` 还是更新计数。

用户不必关闭 `ResultSet`；当产生它的 `Statement` 关闭、重新执行或用于从多结果序列中获取下一个结果时，该 `ResultSet` 将被 `Statement` 自动关闭。

基于JDBC有哪些数据库通用访问方法？

1. 通用数据库Bean设计

本实例中对数据库连接和执行 SQL 语句等通用数据库操作进行了封装，通过实现 `DBConnBean` 和 `DBQueryBean` 两个 `JavaBean` 来完成上述功能。其中 `DBConnBean` 负责 Java 应用程序和数据库的连接；`DBQueryBean` 提供了一组执行标准 SQL 的功能，可以实现标准 SQL 完成的所有功能。其功能代码分别如下所示：

① `DBConnBean.java` 的源代码如下所示：

```
package dbaccess;

import java.sql.*;
import java.util.*;
import java.io.*;

public class DBConnBean
implements Serializable{

private String DBDriver = "sun.jdbc.odbc.JdbcOdbcDriver";
private String DBHost = "127.0.0.1";
private String DBName = "demo";
private String conp = "jdbc: odbc: db_demo";
private String username = "";
private String password = "";
private boolean xdebug = true;

public Connection con = null;

public String sql = null;
```

相信自己-----总会成功

```
Statement stmt = null;
public ResultSet result = null;
private int affectedRows = 0;

public DBConnBean ()
{
xdebug = true;
con = null;
sql = null;
}

public Connection Connect ()
throws Exception
{
String msg = null;
try
{
Class.forName (DBDriver) .newInstance ();
}
catch (Exception e)
{
msg = "加载数据库驱动失败";
if (xdebug) msg += " (驱动"+DBDriver+" ) ";
throw new Exception (msg) ;
}
try
{
String conStr = conp;
con = DriverManager.getConnection (conStr, username, password) ;
}
catch (SQLException e)
```

相信自己-----总会成功

```
{
msg = "!!数据库连接失败";
if (xdebug)
{
msg += " (错误信息=" + e.getMessage () + " SQL状态值=" + e.getSQLState () + " 错误
代码=" + e.getErrorCode () + ") ";
}
throw new Exception (msg) ;
}
return con;
}
protected void finalize ()
throws Throwable
{
super.finalize () ;
if (stmt != null) stmt.close () ;
if (result != null) result.close () ;
}
//最近一次对数据库查询受影响的行数
public int getAffectedRows ()
{
return affectedRows;
}
public Connection getCon ()
{
return con;
}
public String getComp ()
{
return comp;
}
```

相信自己-----总会成功

```
}  
public String getDBDriver ()  
{  
return DBDriver;  
}  
public String getDBName ()  
{  
return DBName;  
}  
public boolean getDebug ()  
{  
return xdebug;  
}  
public String getPassword ()  
{  
return password;  
}  
public ResultSet getResult ()  
{  
return result;  
}  
public String getSql ()  
{  
return sql;  
}  
public String getUsername ()  
{  
return username;  
}  
public void over ()
```

相信自己-----总会成功

```
throws Throwable
{
finalize ( ) ;
}
public ResultSet query ( )
throws Exception
{
result = null;
affectedRows = 0;
if ( con == null )
Connect ( ) ;
if ( stmt == null )
stmt = con.createStatement ( ) ;
if ( sql.substring ( 0, 6 ) .equalsIgnoreCase ( "select" ) )
{
result = stmt.executeQuery ( sql ) ;
}
else
{
affectedRows = stmt.executeUpdate ( sql ) ;
}
return result;
}
public ResultSet query ( String s )
throws Exception
{
sql = s;
return query ( ) ;
}
public void setDBDriver ( String s )
```

相信自己-----总会成功

```
{
DBDriver = s;
}
public void setDebug (boolean b)
{
xdebug = b;
}
public void setgetComp (String s)
{
comp = s;
}
public void setgetDBName (String s)
{
DBName = s;
}
public void setgetUsername (String s)
{
username = s;
}
public void setPassword (String s)
{
password = s;
}
public void setSql (String s)
{
sql = s;
}
}
```

② **DBQueryBean.Java**的源代码如下所示:

```
package dbaccess;

import java.sql.*;
import java.util.*;
import java.io.*;
import java.lang.reflect.*;

public class DBQueryBean
implements Serializable
{
    DBConnBean dbc;
    String sql = null;
    int rowcount = 0;
    int colcount = 0;
    // int limitcount = 0;
    Vector result = null;
    public String _WATCH = "";

    public DBQueryBean ()
    {
        dbc = new DBConnBean ();
        try {
            dbc.Connect ();
        } catch (Exception e) {
            handleException (e);
        }
    }

    protected void finalize ()
    throws Throwable
    {
```

相信自己-----总会成功

```
super.finalize ( ) ;
if ( dbc != null ) dbc.over ( ) ;
if ( result != null ) result.removeAllElements ( ) ;
}
public String get ( int row, int col )
{
if ( result==null || row >= result.size ( ) ) return null;
String r[] = (String[]) result.elementAt ( row ) ;
if ( col >= Java.lang.reflect.Array.getLength ( r ) ) return null;
return r[col];
}
public int getAffRows ( ) { return dbc.getAffectedRows ( ) ; }
public int getColumncount ( ) {
return colcount;
}
public String[] getRow ( int row )
{
if ( result==null || row >= result.size ( ) ) return null;
return (String []) result.elementAt ( row ) ;
/*String ret[] = new String[colcount];
Vector r = (Vector) result.elementAt ( row ) ;
for (int i=0; i<colcount; i++)
ret[i] = (String) r.elementAt ( i ) ;
return ret; */
}
public int getRowcount ( ) {
return rowcount;
}
public void handleException ( Exception e )
{
相信自己-----总会成功
```

```
_WATCH = e.getMessage ( ) ;  
}  
public void init ( )  
{  
rowcount = 0;  
colcount = 0;  
// limitcount = 0;  
result = null;  
}  
public void over ( )  
throws Throwable  
{  
finalize ( ) ;  
}  
public int query (String sql)  
{  
result = new Vector ( ) ;  
int ret = 0;  
try {  
ResultSet rs = dbc.query (sql) ;  
if (rs == null)  
{  
ret = dbc.getAffectedRows ( ) ;  
}  
else  
{  
ResultSetMetaData rm = rs.getMetaData ( ) ;  
colcount = rm.getColumnCount ( ) ;  
while (rs.next ( ) )  
{  
相信自己-----总会成功
```

```
String row[] = new String[colcount];
for (int i=0; i<colcount; i++)
row[i] = rs.getString (i+1) ;
result.addElement (row) ;
rowcount++;
}
rs.close () ; // to release the resource.
ret = result.size () ;
}
}
catch (Exception e)
{
handleException (e) ;
return -1;
}

return ret;
}
}
```

2. 数据库表结构

本实例中主要出现了三个数据库表，表名和字段分别如下所示：

计划采购表: **jhcg_table**

字段名称 中文名称 类型 长度

Goods_no 物品编号 varchar 10

Goods_name 物品名称 Vchar 50

Amount 采购数量 Int

相信自己-----总会成功

Price 采购单价 float

Gold 币种 Vchar 15

Units 单位 Vchar 10

Date 时间 Date

Remark 备注 vchar 100

库存统计表: kctj_table

字段名称 中文名称 类型 长度

Goods_no 物品编号 Vchar 10

Goods_name 物品名称 Vchar 50

amount 库存数量 Int

Date 时间 Date

remark 备注 Vchar 100

实际采购表: sjcg_table

字段名称 中文名称 类型 长度

Goods_no 物品编号 Vchar 10

Goods_name 物品名称 Vchar 50

Amount 采购数量 Int

Price Price 采购单价 Float

Gold 币种 Vchar 15

Units 采购单位 Vchar 10

Date 时间 Date

Remark 备注 vchar 100

其中业务逻辑非常简单，即根据计划采购表和库存统计表生成实际采购表。同时，对各表完成数据库的增、删、改、查等通用操作。

3. JSP设计

① 插入操作

完成对数据库表的记录插入功能，其中计划采购表的插入主页面（insert_jhcg.htm）为：

insert_jhcg.htm 将用户输入传送给 demo_insert_jhcg.jsp，完成插入操作。改 jsp 文件的功能代码如下：

```
<html>
<body>
<jsp: useBean id="DBConn" class="dbaccess.DBConnBean" scope="page"/>
<jsp: useBean id="DBBean" class="dbaccess.DBQueryBean" scope="page"/>
<hr>
<!--test JavaBean-->
<%
if (DBConn == null||DBBean == null) {
out.println ("JavaBean not found!");
return;
}
%>

<!--try db_demo connection-->
<%
try{
DBConn.Connect ();
}catch (Exception e) {
out.println (e.getMessage ());
}
%>
```

相信自己-----总会成功

```
<!--execute sql statement-->
<%
String insGoodno = request.getParameter ("ed_jhcg_no") ;
String insGoodname = request.getParameter ("ed_jhcg_name") ;
int insAmount = ( Integer.valueOf (request.getParameter ("ed_jhcg_amount")) ) .intValue
() ;
float insPrice = ( Float.valueOf (request.getParameter ("ed_jhcg_price")) ) .floatValue () ;
String insGold = request.getParameter ("ed_jhcg_gold") ;
String insUnit = request.getParameter ("ed_jhcg_unit") ;
String insRemark = request.getParameter ("ed_jhcg_remark") ;
String sqlStatement = "insert into jhcg_table (good_no, good_name, amount,
price, gold, unit, remark) values (" + "" + insGoodno + "" + ", " + "" + insGoodname + "" + ",
" + insAmount + ", " + insPrice + ", " + "" + insGold + "" + ", " + "" + insUnit + "" + ", " + "" +
insRemark + "" + ") ";
try{
DBBean.query (sqlStatement) ;
}catch (Exception e) {
out.println (e.getMessage () ) ;
}
%>
<a href="demo_main.htm">Records inserted...Click here to return</a> </p>
</body>
</html>
```

② 查询操作

该查询主页面主要提供对三个数据库表的条件查询功能,

query.htm 将用户选择查询的数据库表和查询条件发送给 demo_query.jsp, 由 jsp 文件完成

数据库查询操作和查询结果集的返回及显示，其功能代码如下：

```
<html>
<body>
<%
String sqlStatement;
String sqlField = "";
String whichTable = "";
String whereClause = "";
String queryNo = "";
String queryName = "";
%>
<jsp: useBean id="DBConn" class="dbaccess.DBConnBean" scope="page"/>
<jsp: useBean id="DBBean" class="dbaccess.DBQueryBean" scope="page"/>
<hr>
<!--test JavaBean-->
<%
if (DBConn == null||DBBean == null) {
out.println ("JavaBean not found!");
return;
}
%>
<!--try db_demo connection-->
<%
try{
DBConn.Connect ();
}catch (Exception e) {
out.println (e.getMessage ());
}
%>
```

相信自己-----总会成功

```
<!--prepare sql statement-->
<%
String queryRequest = request.getParameter ("rb_request") ;
//out.println ("queryRequest: "+queryRequest) ;
String whichCB = "";
if (queryRequest.equals ("1") ) {
whichCB = "ck_jhcg";
whichTable = "jhcg_table";
queryNo = request.getParameter ("ed_jhcg_no") ;
queryName = request.getParameter ("ed_jhcg_name") ;
if (!queryNo.equals ("") )
whereClause = " where good_no="+""+queryNo+"";
if (!queryName.equals ("") ) {
if (!queryNo.equals ("") )
whereClause += " and good_name="+""+queryName+"";
else whereClause = " where good_name="+""+queryName+"";
}
}
if (queryRequest.equals ("2") ) {
whichCB = "ck_kctj";
whichTable = "kctj_table";
queryNo = request.getParameter ("ed_kctj_no") ;
queryName = request.getParameter ("ed_kctj_name") ;
if (!queryNo.equals ("") )
whereClause = " where good_no="+""+queryNo+"";
if (!queryName.equals ("") ) {
if (!queryNo.equals ("") )
whereClause += " and good_name="+""+queryName+"";
else whereClause = " where good_name="+""+queryName+"";
}
}
}
```

相信自己-----总会成功

```
}
if (queryRequest.equals ("3")) {
whichCB = "ck_sjcg";
whichTable = "sjcg_table";
queryNo = request.getParameter ("ed_sjcg_no");
queryName = request.getParameter ("ed_sjcg_name");
if (!queryNo.equals ("")) {
whereClause = " where good_no=" + "" + queryNo + "";
if (!queryName.equals ("")) {
if (!queryNo.equals ("")) {
whereClause += " and good_name=" + "" + queryName + "";
else whereClause = " where good_name=" + "" + queryName + "";
}
}
String[] printTitle = request.getParameterValues (whichCB);

%>
<!--create query sql statement-->
<%
sqlStatement = "select ";
for (int i = 0; i<printTitle.length; i++) {
sqlField += printTitle[i] + ", ";
}
sqlStatement += sqlField.substring (0, sqlField.length () -1) + " from " + whichTable;
if (!whereClause.equals ("")) {
sqlStatement += whereClause;
}
%>
<!--show query response-->
<%
```

```
try{
DBBean.query (sqlStatement) ;
}catch (Exception e) {
out.println ("Database Error!") ;
}

int rows = DBBean.getRowcount () ;
int cols = DBBean.getColumncount () ;

%>
<Table align="center" width="80%" border=1>
<tr align=center>
<%
for (int i = 0; i < printTitle.length; i++) {
out.println ("<td><b>");
out.println (printTitle[i]) ;
out.println ("</b></td>");
}
%>
</tr>
<%
for (int i = 0; i < rows; i++) {
out.println ("<tr>");
for (int j = 0; j < cols; j++)
out.println ("<td>"+DBBean.get (i, j) +"</td>");
out.println ("</tr>");
}
%>
</Table>
<br>
<hr>
<a href="demo_main.htm">Click here to return</a> </p>
```

相信自己-----总会成功

```
</body>
```

```
</html>
```

③ 生成实际采购表

生成数据库表是一个隐式操作,程序根据计划采购表和库存统计表的相应字段生成实际采购表,不需要用户的任何输入,其功能代码如下(demo_create.jsp):

```
<%@page import="Java.util.*"%>
```

```
<html>
```

```
<body>
```

```
<jsp: useBean id="DBConn" class="dbaccess.DBConnBean" scope="page"/>
```

```
<jsp: useBean id="DBBean" class="dbaccess.DBQueryBean" scope="page"/>
```

```
<hr>
```

```
<!--test JavaBean-->
```

```
<%
```

```
if (DBConn == null||DBBean == null) {
```

```
out.println ("JavaBean not found!");
```

```
return;
```

```
}
```

```
%>
```

```
<!--try db_demo connection-->
```

```
<%
```

```
try{
```

```
DBConn.Connect ();
```

```
}catch (Exception e) {
```

```
out.println (e.getMessage ());
```

相信自己-----总会成功

```
}
%>

<!--prepare sql statement-->
<%
int amount_jhcg, amount_kctj;
Vector updateRs = new Vector ( ) ;
DBBean.query ("delete * from sjcg_table") ; //delete all old records in sjcg_table
DBBean.query ("select jhcg_table.good_no, jhcg_table.good_name, jhcg_table.amount,
kctj_table.amount, jhcg_table.unit from jhcg_table left join kctj_table on
kctj_table.good_no=jhcg_table.good_no") ;
int rows = DBBean.getRowcount ( ) ;
int cols = DBBean.getColumncount ( ) ;
for (int i = 0; i < rows; i++) {
String record[] = new String[4];
record[0] = DBBean.get (i, 0) ;
record[1] = DBBean.get (i, 1) ;
amount_jhcg = (Integer.valueOf (DBBean.get (i, 2) ) ) .intValue ( ) ;
if (DBBean.get (i, 3) == null) amount_kctj = 0;
else amount_kctj = (Integer.valueOf (DBBean.get (i, 3) ) ) .intValue ( ) ;
record[2] = Integer.toString (amount_jhcg - amount_kctj) ;
record[3] = DBBean.get (i, 4) ;
updateRs.addElement (record) ;
}
for (int i = 0; i < rows; i++) {
String insRecord[] = (String []) updateRs.elementAt (i) ;
String insGoodno, insGoodname, insUnit, insAmount;
insGoodno = insRecord[0];
insGoodname = insRecord[1];
insAmount = insRecord[2];
```

相信自己-----总会成功

```
insUnit = insRecord[3];
String sqlStatement = "insert into sjcg_table (good_no, good_name, amount, unit)
values?quot;+""+insGoodno+""+", "+""+insGoodname+""+", "+insAmount+",
"+""+insUnit+""+" ) ";
DBBean.query (sqlStatement) ;
DBBean.query ("delete * from sjcg_table where amount<=0" ) ;
}
%>
<a href="demo_main.htm">Database created...Click here to return</a></p>
</body>
</html>
```

上述的开发工具综合应用介绍了基于 Java 开发电子商务应用系统的全过程，包括应用开发平台搭建、业务流程分析、JavaBean 封装和 JSP 开发等内容，其中 JSP 开发中涉及到了通用 SQL（查询和插入数据库表）和游标操作（生成实际采购表），基本可以完成任何网络数据库应用的需求。本实例基本上可以将前面介绍的基于 Java 的电子商务开发技术串接起来，指导读者进行电子商务应用开发。

如何在JSP中实现分页显示？

分页显示是 Web 数据库应用中经常需要遇到的问题，当用户的数据库查询结果远远超过了计算机屏幕的显示能力的时候，我们该如何合理的将数据呈现给用户呢？答案就是数据库分页显示，可以完美的解决上述问题。下面是一个数据库分页操作的通用实例，对任何数据库平台上的分页功能都有很好的借鉴意义。

```
<%@ page contentType="text/html;charset=8859_1" %>
```

```
<%
```

```
//变量声明
```

```
Java.sql.Connection sqlCon; //数据库连接对象
```

相信自己-----总会成功

```
Java.sql.Statement sqlStmt; //SQL 语句对象
Java.sql.ResultSet sqlRst; //结果集对象

Java.lang.String strCon; //数据库连接字符串
Java.lang.String strSQL; //SQL 语句

int intPageSize; //一页显示的记录数
int intRowCount; //记录总数
int intPageCount; //总页数
int intPage; //待显示页码
Java.lang.String strPage;

int i;

//设置一页显示的记录数
intPageSize = 2;

//取得待显示页码
strPage = request.getParameter("page");
if(strPage==null){//表明在 QueryString 中没有 page 这一个参数, 此时显示第一页数据
intPage = 1;
}
else{//将字符串转换成整型
intPage = Java.lang.Integer.parseInt(strPage);
if(intPage<1) intPage = 1;
}

//装载 JDBC 驱动程序
Java.sql.DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());

//设置数据库连接字符串
strCon = "jdbc:oracle:thin:@linux:1521:ora4cweb";
```

相信自己-----总会成功

```
//连接数据库
sqlCon = Java.sql.DriverManager.getConnection(strCon,"hzq","hzq");

//创建一个可以滚动的只读的 SQL 语句对象
sqlStmt =
sqlCon.createStatement(Java.sql.ResultSet.TYPE_SCROLL_INSENSITIVE,Java.sql.ResultSet.C
ONCUR_READ_ONLY);

//准备 SQL 语句
strSQL = "select name,age from test";

//执行 SQL 语句并获取结果集
sqlRst = sqlStmt.executeQuery(strSQL);

//获取记录总数
sqlRst.last();
intRowCount = sqlRst.getRow();

//计算总页数
intPageCount = (intRowCount+intPageSize-1) / intPageSize;

//调整待显示的页码
if(intPage>intPageCount) intPage = intPageCount;
%>

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>JSP 数据库操作例程 - 数据分页显示 - JDBC 2.0 - Oracle</title>
</head>

<body>
```

```
<table border="1" cellspacing="0" cellpadding="0">
<tr>
<th>姓名</th>
<th>年龄</th>
</tr>

<%
if(intPageCount>0){
//将记录指针定位到待显示页的第一条记录上
sqlRst.absolute((intPage-1) * intPageSize + 1);

//显示数据
i = 0;
while(i<intPageSize && !sqlRst.isAfterLast()){
%>
<tr>
<td><%=sqlRst.getString(1)%></td>
<td><%=sqlRst.getString(2)%></td>
</tr>
<%
sqlRst.next();
i++;
}
}
%>

</table>

第<%=intPage%>页 共<%=intPageCount%>页
  <%if(intPage<intPageCount){%><a href="jdbc20-oracle.jsp?page=<%=intPage+1%>
">下一页
```

相信自己-----总会成功

```
</a><%}%> <%if(intPage>1){%><a href="jdbc20-oracle.jsp?page=<
%=intPage-1%>">上一页</a><%}%>
```

```
</body>
```

```
</html>
```

```
<%
```

```
//关闭结果集
```

```
sqlRst.close();
```

```
//关闭 SQL 语句对象
```

```
sqlStmt.close();
```

```
//关闭数据库
```

```
sqlCon.close();
```

```
%>
```

JDBC TM技术解析

1. JDBCTM的定义

JDBCTM 是一种用于执行 SQL 语句的 JavaTM API，它由一组用 Java 编程语言编写的类和接口组成。JDBC 为工具/数据库开发人员提供了一个标准的API，使他们能够用纯 Java API 来编写数据库应用程序。

有了 JDBC，向各种关系数据库发送 SQL 语句就是一件很容易的事。换言之，有了 JDBC API，就不必为访问 Sybase 数据库专门写一个程序，为访问 Oracle 数据库又专门写一个程序，为访问 Informix 数据库又写另一个程序，等等。您只需用 JDBC API 写一个程序就够了它可向相应数据库发送 SQL 语句。而且，使用 Java 编程语言编写的应用程序，就无须去忧虑要为不同的平台编写不同的应用程序。将 Java 和 JDBC 结合起来将使程序员只须写一遍程序就可让它在任何平台上运行。

相信自己-----总会成功

Java 具有坚固、安全、易于使用、易于理解和可从网络上自动下载等特性，是编写数据库应用程序的杰出语言。所需要的只是 Java 应用程序与各种不同数据库之间进行对话的方法。而 JDBC 正是作为此种用途的机制。

JDBC 扩展了 Java 的功能。例如，用 Java 和 JDBC API 可以发布含有 applet 的网页，而该 applet 使用的信息可能来自远程数据库企业也可以用 JDBC 通过 Intranet 将所有职员连到一个或多个内部数据库中（即使这些职员所用的计算机有 Windows、Macintosh 和 UNIX 等各种不同的操作系统）。随着越来越多的程序员开始使用 Java 编程语言，对从 Java 中便捷地访问数据库的要求也在日益增加。

MIS 管理员们都喜欢 Java 和 JDBC 的结合，因为它使信息传播变得容易和经济。企业可继续使用它们安装好的数据库，并能便捷地存取信息，即使这些信息是储存在不同数据库管理系统上。新程序的开发期很短。安装和版本控制将大为简化。程序员可只编写一遍应用程序或只更新一次，然后将它放到服务器上，随后任何人就都可得到最新版本的应用程序。对于商务上的销售信息服务，Java 和 JDBC 可为外部客户提供获取信息更新的更好方法。

2. JDBC 的用途

简单地说，JDBC 可做三件事：与数据库建立连接、发送 SQL 语句并处理结果。下列代码段给出了以上三步的基本示例：

```
Connection con = DriverManager.getConnection ("jdbc: odbc: wombat", "login",
"password") ;
Statement stmt = con.createStatement () ;
ResultSet rs = stmt.executeQuery ("SELECT a, b, c FROM Table1") ;
while (rs.next () ) {
int x = rs.getInt ("a") ;
String s = rs.getString ("b") ;
float f = rs.getFloat ("c") ;
}
```

相信自己-----总会成功

上述代码对基于 JDBC 的数据库访问做了经典的总结，当然，在本小节的后续部分会对它做详尽的分析讲解。

3. JDBC API

JDBC 是个"低级"接口，也就是说，它用于直接调用 SQL 命令。在这方面它的功能极佳，并比其它的数据库连接 API 易于使用，但它同时也被设计为一种基础接口，在它之上可以建立高级接口和工具。高级接口是"对用户友好的"接口，它使用的是一种更易理解和更为方便的 API，这种API在幕后被转换为诸如 JDBC 这样的低级接口。

在关系数据库的"对象/关系"映射中，表中的每行对应于类的一个实例，而每列的值对应于该实例的一个属性。于是，程序员可直接对 Java 对象进行操作；存取数据所需的 SQL 调用将在"掩盖下"自动生成。此外还可提供更复杂的映射，例如将多个表中的行结合进一个 Java 类中。

随着人们对 JDBC 的兴趣日益增涨，越来越多的开发人员一直在使用基于 JDBC 的工具，以使程序的编写更加容易。程序员也一直在编写力图使最终用户对数据库的访问变得更为简单的应用程序。例如应用程序可提供一个选择数据库任务的菜单。任务被选定后，应用程序将给出提示及空白供填写执行选定任务所需的信息。所需信息输入应用程序将自动调用所需的 SQL 命令。在这样一种程序的协助下，即使用户根本不懂 SQL 的语法，也可以执行数据库任务。

4. JDBC与ODBC和其它API的比较

目前，Microsoft 的 ODBC API 可能是使用最广的、用于访问关系数据库的编程接口。它能在几乎所有平台上连接几乎所有的数据库。为什么 Java 不使用 ODBC? 对这个问题的回答是：Java 可以使用 ODBC，但最好是在 JDBC 的帮助下以 JDBC-ODBC 桥的形式使用，这一点我们稍后再说。现在的问题已变成："为什么需要 JDBC"? 答案是显然的：ODBC 不适合直接在 Java 中使用，因为它使用 C 语言接口。从Java 调用本地 C 代码在安全性、实现、坚固性和程序的自动移植性方面都有许多缺点。从 ODBC C API 到 Java API 的字面翻译是不可取的。例如，Java 没有指针，而 ODBC 却对指针用得很广泛（包括很容易出错的指针"void *"）。您可以将 JDBC 想象成被转换为面向对象接口的 ODBC，而面向对象的接口对 Java 程序员来说较易于接收。

ODBC 很难学。它把简单和高级功能混在一起,而且即使对于简单的查询,其选项也极为复杂。相反,JDBC 尽量保证简单功能的简便性,而同时在必要时允许使用高级功能。启用"纯 Java "机制需要象 JDBC 这样的 Java API。如果使用 ODBC,就必须手动地将 ODBC 驱动程序管理器和驱动程序安装在每台客户机上。如果完全用 Java 编写 JDBC 驱动程序则 JDBC 代码在所有 Java 平台上(从网络计算机到大型机)都可以自动安装、移植并保证安全性。

总之,JDBC API 对于基本的 SQL 抽象和概念是一种自然的 Java 接口。它建立在 ODBC 上而不是从零开始。因此,熟悉 ODBC 的程序员将发现 JDBC 很容易使用。JDBC 保留了 ODBC 的基本设计特征;事实上,两种接口都基于 X/Open SQL CLI(调用级接口)。它们之间最大的区别在于:JDBC 以 Java 风格与优点为基础并进行优化,因此更加易于使用。

目前,Microsoft 又引进了 ODBC 之外的新 API: RDO、ADO 和 OLE DB。这些设计在许多方面与 JDBC 是相同的,即它们都是面向对象的数据库接口且基于可在 ODBC 上实现的类。但在这些接口中,我们未看见有特别的功能使我们要转而选择它们来替代 ODBC,尤其是在 ODBC 驱动程序已建立起较为完善的市场的环境下。它们最多也就是在 ODBC 上加了一种装饰而已。

5. JDBC对B/S和C/S模式的支持

JDBC API 既支持数据库访问的两层模型(C/S),同时也支持三层模型(B/S)。在两层模型中,Java applet或应用程序将直接与数据库进行对话。这将需要一个JDBC驱动程序来与所访问的特定数据库管理系统进行通讯。用户的SQL语句被送往数据库中,而其结果将被送回给用户。数据库可以位于另一台计算机上,用户通过网络连接到上面。这就叫做客户机/服务器配置,其中用户的计算机为客户机,提供数据库的计算机为服务器。网络可以是 Intranet(它可将公司职员连接起来),也可以是 Internet。

在三层模型中,命令先是被发送到服务的"中间层",然后由它将 SQL 语句发送给数据库。数据库对 SQL 语句进行处理并将结果送回到中间层,中间层再将结果送回给用户。MIS 主管们都发现三层模型很吸引人,因为可用中间层来控制对公司数据的访问和可作的的更新的种类。中间层的另一个好处是,用户可以利用易于使用的高级 API,而中间层将把它转换为相应的低级调用。最后,许多情况下三层结构可提供一些性能上的好处。

到目前为止,中间层通常都用 C 或 C++ 这类语言来编写,这些语言执行速度较快。然而,随着最优化编译器(它把 Java 字节代码转换为高效的特定于机器的代码)的引入,用 Java 来实现中间层将变得越来越实际。这将是一个很大的进步,它使人们可以充分利用 Java 的诸多优点(如坚固、多线程和安全等特征)。JDBC 对于从 Java 的中间层来访问数据库非常重要。

6. SQL 的一致性

结构化查询语言 (SQL) 是访问关系数据库的标准语言。困难之处在于:虽然大多数的 DBMS (数据库管理系统)对其基本功能都使用了标准形式的 SQL,但它们却不符合最近为更高级的功能定义的标准 SQL 语法或语义。例如,并非所有的数据库都支持储存程序或外部连接,那些支持这一功能的数据库又相互不一致。人们希望 SQL 中真正标准的那部份能够进行扩展以包括越来越多的功能。但同时 JDBC API 又必须支持现有的 SQL。

JDBC API 解决这个问题的一种方法是允许将任何查询字符串一直传到所涉及的 DBMS 驱动程序上。这意味着应用程序可以使用任意多的 SQL 功能,但它必须冒这样的风险:有可能在某些 DBMS 上出错。事实上,应用程序查询甚至不一定要是 SQL,或者说它可以是个为特定的 DBMS 设计的 SQL 的专用派生物(例如,文档或图象查询)。

JDBC 处理 SQL 一致性问题的第二种方法是提供 ODBC 风格的转义子句,这将在后续部分中讨论。转义语法为几个常见的 SQL 分歧提供了一种标准的 JDBC 语法。例如,对日期文字和已储存过程的调用都有转义语法。

对于复杂的应用程序,JDBC 用第三种方法来处理 SQL 的一致性它利用 DatabaseMetaData 接口来提供关于 DBMS 的描述性信息,从而使应用程序能适应每个 DBMS 的要求和功能。

由于 JDBC API 将用作开发高级数据库访问工具和 API 的基础 API,因此它还必须注意其所有上层建筑的一致性。"符合 JDBC 标准 TM" 代表用户可依赖的 JDBC 功能的标准级别。要使用这一说明,驱动程序至少必须支持 ANSI SQL-2 Entry Level (ANSI SQL-2 代表美国国家标准局 1992 年所采用的标准。Entry Level 代表 SQL 功能的特定清单)。驱动程序开发人员可用 JDBC API 所带的测试工具包来确定他们的驱动程序是否符合这些标准。

"符合 JDBC 标准 TM" 表示提供者的 JDBC 实现已经通过了 JavaSoft 提供的一致性测试。这些一致性测试将检查 JDBC API 中定义的所有类和方法是否都存在,并尽可能地检查程序是否具有 SQL Entry Level 功能。当然,这些测试并不完全,而且 JavaSoft 目前也无意对各提供者的实现进行标级。但这种一致性定义的确可对 JDBC 实现提供一定的可信度。随着越来越多的数据库提供者、连接提供者、Internet 提供者和应用程序程序员对 JDBC API 的接受,JDBC 也正迅速成为 Java 数据库访问的标准。