Curso de Java – Módulo III
JDBC

Fábio Mengue – fabio@unicamp.br
Centro de Computação - Unicamp

Conceitos

A tecnologia JDBC foi criada para facilitar o acesso de programas Java a qualquer banco de dados relacional. A idéia era criar uma maneira simples de prover acesso à sintaxe SQL, propiciando ao programador o uso de SELECT's, UPDATE's e outros comandos relevantes na sintaxe.

O JDBC existe desde a versão 1.1 da JDK, e participa da API básica (pacote java.sql). Mas a partir da versão 1.2, existe um pacote adicional de classes para extensão do pacote sql, que provê novas funcionalidades, como suporte a pool de conexões e transação (pacote javax.sql). Neste treinamento, iremos nos ater apenas ao pacote básico.

A seguir, descrevemos o que deve ser instalado e os passos para o acesso a um banco de dados em máquina Windows. Para o acesso a outras plataformas, o procedimento é basicamente o mesmo.

Instalando o ambiente

Claro que para criar um programa Java que faz acesso a banco, deve-se instalar a JDK. Ao fazer isso, já existe na API o acesso ao pacote básico. Após a instalação do Java, devemos instalar um driver que irá permitir a conexão da JVM com o banco de dados em questão. Os drivers JDBC podem ser feitos em código nativo do sistema operacional onde funcionam (como os drivers dos SGBD para Windows) ou podem ser feitos independente de plataforma (como a maioria dos drivers de SGBD Open Souce).

De qualquer maneira, existem instruções de como fazer isso no pacote do SGBD. Nesse material, utilizaremos um banco de dados baseados no Access. As máquinas Windows não possuem driver JDBC gratuito, portanto a Sun criou uma "ponte" que permite o JDBC acessar drivers ODBC (nativo da Microsoft). O Windows normalmente possui no "Painel de Controle" um ícone chamado "Fontes de Dados (ODBC)". Esse aplicativo permite a criação de uma ligação ODBC entre o arquivo .MDB e qualquer aplicativo que tenha a capacidade de ligação com o protocolo. Basta identificar o arquivo .MDB e atribuir um nome que será utilizado para ligar o programa àquela fonte de dados.

Uma das partes interessantes deste tipo de configuração é que não é necessária a instalação do Access para que as coisas funcionem. Basta o arquivo .MDB. Além disso, a identificação da fonte de dados com um nome permite ao programador fazer um programa que se conecte a mais de uma fonte de dados e permite uma flexibilidade grande pois o arquivo da fonte

de dados pode ter seu nome alterado e mesmo ser movido de local; desde que a definição da fonte de dados continue disponível, o programa Java funciona.

Estabelecendo a conexão

Depois de iniciar o programa e importar os pacotes necessários (**import java.sql.*;**), devemos seguir certos passos para que seja possível o acesso ao SGBD. Devemos inicialmente carregar o driver e depois criar a conexão propriamente dita.

A carga do(s) driver(s) é muito simples. Basta invocar o método estático

Class.forName("jdbc.DriverXYZ");

A classe **Class** tem como função instanciar a classe identificada entre parênteses e registrá-la com a JVM, que irá utilizá-la para acessar o SGBD. Em nosso exemplo, utilizaremos a "ponte" ODBC – JDBC:

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

O segundo passo é abrir a conexão. Para isso, normalmente teremos que entrar em contato com o SGBD. No nosso exemplo, estamos conectando com um arquivo estático, mas normalmente estabelecemos conexão com um banco real e existem restrições de segurança. Devemos passar nomes de tabela, e credenciais. A linha mostrada abaixo dá a idéia geral de como é feita a conexão:

Connection con = DriverManager.getConnection(url, "usuario", "senha");

Não é muito complexa. A parte mais "desafiadora" é montar a **url**. Essa parte da informação sempre se inicia com **jdbc:**, mas o resto depende do fabricante do banco. Na seqüência normalmente temos o nome do fabricante (então temos **jdbc:db2, jdbc:oracle, jdbc:psql**, etc.) e por último o nome da fonte de dados (então teremos **jdbc:db2:db, jdbc:oracle:server.unicamp.br:1234/meubanco, jdbc:psql:meubanco)**. Notem que a fonte de dados, para a maioria dos SGBD, representa uma tabela (com as visões dentro dela).

Se eu defini minha fonte de dados no Windows como **bd**, então minha **url** de conexão será **jdbc:odbc:bd**. Obviamente, no lugar de **usuário** e **senha**, devemos utilizar dados válidos constantes nos grants do SGBD para o acesso àquela tabela.

Finalmente, a nossa conexão ficaria definida assim:

String url = "jdbc:odbc:bd";
Connection con = DriverManager.getConnection(url, "", "");

Toda a parte envolvida com a conexão propriamente dita acontece de maneira transparente. O programador não tem necessidade nenhuma de saber o que se passa e, a não ser que ele um dia resolva escrever seu próprio driver para conexão a algum SGBD, tem que se dar por satisfeito da coisa acontecer assim .

A partir de agora, o objeto **com** representa uma conexão ativa e aberta ao banco de dados e o programador pode utiliza-lo para enviar *statements* SQL para o banco.

Criando *statements*

É necessário criar um objeto da classe **Statement** para armazenar o pedido que será enviado ao SGBD. O programador deve simplesmente criar um objeto deste tipo, informar a cadeia de strings que representa seu comando SQL e depois executá-lo, usando métodos especiais dependendo do tipo de comando desejado. Para um SELECT, usa-se **executeQuery**, Para INSERT, UPDATE e DELETE, **executeUpdate**. Por exemplo:

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

Neste ponto, um *statement* foi criado. Mas não existe nenhum comando SQL nele. Podemos executar um comando como no exemplo abaixo:

```
stmt.executeUpdate("CREATE TABLE COFFEES " +
"(COF_NAME VARCHAR(32), SUP_ID INTEGER, PRICE FLOAT, " +
"SALES INTEGER, TOTAL INTEGER)");
```

Note que existe uma estrutura de parênteses, sinais de concatenação a aspas. Esses sinais são importantes, e muitas vezes causa de confusões tremendas em programas. Muito cuidado com eles, principalmente quando houver variáveis do programa Java misturadas ao SQL.

O método mais comum de ser utilizado é o **executeQuery**. Mas ele tem um detalhe: retorna dados. Assim, devemos ter outro tipo de classe que serve exclusivamente para receber esses dados, de maneira que possamos fazer uso deles no processamento.

Essa classe é chamada de **ResultSet**. É necessário instanciar um objeto desta classe que receba o resultado do **executeQuery**. Veja o exemplo:

```
ResultSet rs = stmt.executeQuery("SELECT COF_NAME, PRICE FROM COFFEES");
```

A partir de agora, tenho um objeto chamado **rs** que armazena todos os dados recebidos. Em linhas gerais, os programadores estabelecem um *loop* onde fazem a leitura dos dados baseados em métodos presentes na classe **ResultSet**. Esses métodos permitem “andar” pelos resultados e copiar os valores lidos para outros objetos. Veja o exemplo na próxima página.

```
...
String query = "SELECT COF_NAME, PRICE FROM COFFEES";
ResultSet rs = stmt.executeQuery(query);
while (rs.next()) {
        String s = rs.getString("COF_NAME");
        float n = rs.getFloat("PRICE");
...
```

Para acessar os nomes e preços, corremos cada registro e acessamos os valores de acordo com o tipo. O método **next()** movimenta o cursor para o próximo registro, tornando aquele registro o corrente. Podemos então executar outros métodos. Note que o cursor inicialmente (e convenientemente) fica posicionado acima do primeiro registro, exigindo uma primeira execução do **next()** para que possamos processar o primeiro registro.

Utilizamos métodos cujo nome se iniciam com *get* para obter dados das colunas. Eles devem ser obtidos de acordo com o tipo (o cast pode ser efetuado, quando houver possibilidade), e esses métodos permitem que apontemos as colunas pelo nome ou pelo índice (seu número, na ordem em que foi recebida e não na ordem da tabela original). Assim, o código acima poderia ser:

```
String s = rs.getString(1);
float n = rs.getFloat(2);
```

Essa última abordagem tem uma performance melhor.

As primeiras classes **ResultSet** normalmente implementavam acesso do primeiro registro para o último, e não se pode voltar atrás na leitura. A partir da versão 1.2 da JDK, o programador pode usar instruções do tipo:

```
rs.absolute(5); // move o cursor para o quinto registro
rs.updateString("NAME", "teste");       // altera o valor da coluna NAME do
                                        // quinto registro para “teste”
rs.updateRow(); // atualiza a coluna na tabela
```

Usando Transações

Existem situações onde não se deseja que uma instrução tenha efeito a não ser que outras também tenham. Algumas situações necessitam que dados sejam incluídos em mais de uma tabela para que sejam considerados consistentes. Ou os dados estão presentes em todas as tabelas ou não devem estar presentes em nenhuma.

A técnica utilizada para garantir esse tipo de consistência é o uso de transações. Uma transação nada mais é do que uma série de *statements* que são executados juntos, como uma coisa única: ou todos falham, ou todos são executados.

A primeira providência é desabilitar o *auto-commit*. Quando uma conexão é criada, o padrão é que ela trate cada **executeUpdate** como uma transação separada, que é validada (*comit*) assim que é completada. Assim, devemos utilizar o código

con.setAutoCommit(false);

A partir daqui, nenhum *statement* será validado até que o programador permita. Todos eles serão agrupados e validados como um só. Ao final do processo (caso tudo tenha transcorrido de acordo com o esperado), executa-se:

con.commit();

Caso algo não tenha dado certo, podemos executar:

con.rollback();

E todo *statement* executado será descartado.

É importante lembrar algumas coisas quando lidamos com a transação de maneira manual. Os recursos de banco normalmente ficam presos, esperando um **commit()** ou **rollback()**. Assim, se um programa tratar ou entrar em loop (ou mesmo demorar muito) o acesso ao banco fica prejudicado. Existe ainda a possibilidade de um "dirty read", onde outro programa recupera dados do disco que você está alterando, mas ainda não validou. Esse tipo de comportamento pode ser evitado aumentando o nível de isolamento entre transações. Verifique o método **setTransactionIsolation()** da classe **Connection.**

MetaDados

Ocasionalmente, você irá sentir necessidade de obter mais informações sobre o resultado de um SELECT ou mesmo sobre a tabela em questão. Você pode obter esses dados utilizando a classe **ResultSetMetaData**, que pode ser criada segundo o exemplo:

ResultSet rs = stmt.executeQuery(query);
ResultSetMetaData meta = rs.getMetaData();

A classe **ResultSetMetaData** lhe fornece informações acerca do número de colunas recebidas, o nome e tipo das colunas, se alguma coluna aceita dados nulos, campos de data, se ela é auto incrementável, se pode ser nula, se permite a escrita, etc. Além disso fornece também informações sobre a tabela com que estamos trabalhando.

Dicas e novidades (JDBC 2.0 e 3.0)

A versão 1.4 da JDK já inclui a versão 3.0 do JDBC. Não é a versão em produção hoje em dia (muitos SGBD ainda não tem conformidade com ela). Mas existem as extensões do JDBC 2.0 (representados por javax.sql) e esse pacote possui uma série de serviços interessantes.

Dentre esses serviços, temos a classe **RowSet**. Elas permitem que o programador abra uma conexão, receba os resultado de um SELECT, e a seguir encerre essa conexão, enquanto processa os dados recebidos. Por incrível que pareça, o **ResultSet** não pode fazer isso; ele exige que se mantenha a conexão aberta enquanto os dados são lidos. O **RowSet** permite que se libere o recurso rapidamente, permitindo um melhor aproveitamento dos recursos do SGBD, o que possibilita atender mais usuários.

Outro recurso é a possibilidade de obter conexão ao banco utilizando um serviço de diretório (baseado nas classes JNDI – Java Naming Directory Interface). Não é mais necessário montar a url (com dados de servidores e portas que podem mudar) ou mesmo trabalhar com usernames e senhas dentro do programa (que são voláteis e tem que ser trocadas por questão de segurança). O programa identifica um nome lógico, e a rede provê informação sobre a fonte de dados. Veja um exemplo:

```
import java.sql.*;
import javax.sql.*;
import javax.naming.*;
...
public Conexao() throws Exception {
      Context ctx = new InitialContext();
      DataSource ds = (DataSource)ctx.lookup("java:comp/env/jdbc/DataSource");
      java.sql.Connection c = ds.getConnection();
      ...
   }
...
```

Um dos recursos mais interessantes é a capacidade de trabalhar com pools de conexão. Essas “entidades” nada mais são do que vetores de conexões que ficam permanentemente abertas com o SGBD, diminuindo o “overhead” de abrir e fechar uma conexão (operação que normalmente é custosa). Ao necessitar de conexão, uma conexão já aberta é fornecida ao programa. Quando o programador vai fechá-la, ela é devolvida ao pool, pare ser reutilizada mais tarde.

E por último, o suporte a transação distribuída. A Sun tem uma API especial para o tratamento de transações entre programas. Esse pacote (chamado Java Transaction API – JTA) permite a realização de um *commit* em duas fases, onde a transação começa em um programa e termina em outro. Esse tipo de serviço é essencial em ambientes distribuídos (onde os componentes nem sempre estão na mesma máquina).

Para essas e outras maravilhosas invenções, veja o endereço http://java.sun.com/products/jdbc/ .

Apêndice A – Programa de Exemplo

```
import java.sql.*;

public class Principal {

   public static void main(String[] args) throws Exception {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      Connection con = DriverManager.getConnection("jdbc:odbc:bd");
      Statement stmt = con.createStatement();

      ResultSet rs = stmt.executeQuery("SELECT LOGIN,NOME FROM ATENDENTE");

      while (rs.next()) {
         String login = rs.getString("LOGIN");
         String nome = rs.getString("NOME");
         System.out.println("LOGIN:" + login);
         System.out.println("NOME:" + nome);
      }
   }
}
```

Bibliografia

The Java Tutorial – http://www.sun.com/docs
JDBC Home Page – http://java.sun.com/products/jdbc