JavaFree.com.br

[Tutoriais] - Banco de Dados - Acessando banco de dados em Java (PARTE 1)

aspirante - Qui Out 16, 2003 11:22 pm
Assunto: Acessando banco de dados em Java (PARTE 1)

Uma funcionalidade essencial em qualquer sistema é a habilidade para comunicar-se com um repositório de dados. Podemos definir repositório de dados de várias maneiras, por exemplo, como um pool de objetos de negócio num ORB ou um banco de dados. Bancos de dados constituem o tipo mais comum de repositório. Java dispõe de uma API para acessar repositórios de dados: a Java DataBase Connectivity API ou JDBC API.
A JDBC implementa em Java a funcionalidade definida pelo padrão SQL Call Level Interface ou SQLCLI. Um outro exemplo de API que implementa o SQL Call Level Interface é o popularíssimo ODBC das plataformas Wintel. A maioria dos fornecedores de bancos de dados oferece uma implementação particular de SQLCLI. A vantagem de JDBC é a portabilidade da aplicação cliente, inerente da linguagem Java. A especificação corrente da JDBC API é a 2.1.
A JDBC compreende uma especificação para ambos: os desenvolvedores de drivers JDBC e os desenvolvedores de aplicações clientes que precisem acessar bancos de dados em Java. Estaremos dando uma olhada no desenvolvimento de aplicações em Java, então, é uma boa idéia começar com o suporte de dados.
Existem 4 tipos de diferentes de drivers JDBC (para uma lista de fornecedores por especificação e tipo, vide http://www.javasoft.com/products/jdbc/drivers.html):
- Uma vez que ODBC é uma especificação padrão do mundo Wintel, o tipo 1 é um driver de ponte entre Java e ODBC. O driver de ponte mais conhecido é o fornecido pela Sun o JDBC-ODBC bridge. Este tipo de driver não é portável, pois depende de chamadas a funções de ODBC implementadas em linguagem C e compiladas para Wintel, ou outra plataforma ODBC compatível, as chamadas funções nativas.
- O driver tipo 2 é implementado parcialmente em Java e parcialmente através de funções nativas que implementam alguma API específica do fornecedor de banco de dados. Este tipo faz o que se chama de wrap-out, ou seja, provê uma interface Java para uma API nativa não-Java.
- O tipo 3 é um driver totalmente Java que se comunica com algum tipo de middleware que então se comunica com o banco de dados
- O tipo 4 é um driver totalmente Java que vai diretamente ao banco de dados.
Numa próxima parte veremos ainda um driver gratuito que permite acessar bancos de dados que ofereçam suporte apenas ao Bridge (tipo 1) via rede. Veremos a seguir como acessar um banco de dados através de JDBC. Nosso cenário básico é uma pequena aplicação de controle dos meus CDs (clássica !) implementada em algum xBase compatível. Em próximos exemplos iremos utilizar outros bancos de dados.
Para utilizarmos a JDBC num programa em Java, precisamos declarar o pacote que contém a JDBC API:

Acessando bancos de dados em JDBC

Código:

```
import java.sql.*;
```

A primeira coisa a fazer é estabelecer uma conexão com o banco de dados. Fazemos isso em dois passos: primeiro carregamos o driver para o banco de dados na JVM da aplicação (1). Uma vez carregado, o driver se registra para o DriverManager e está disponível para a aplicação. Utilizamos então a classe DriverManager para abrir uma conexão com o banco de dados (2). A interface Connection designa um objeto, no caso con, para receber a conexão estabelecida:

Código:

```
try //A captura de exceções SQLException em Java é obrigatória para usarmos JDBC.
{
   // Este é um dos meios para registrar um driver
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").getInstance();

   // Registrado o driver, vamos estabelecer uma conexão
   Connection con = DriverManager.getConnection("jdbc:odbc:meusCdsDb","conta","senha");
```

```
}
catch(SQLException e)
{
   // se houve algum erro, uma exceção é gerada para informar o erro
   e.printStackTrace(); //vejamos que erro foi gerado e quem o gerou
}
```

Estabelecida a conexão, podemos executar comandos SQL para o banco de dados. Vejamos como realizar uma consulta sobre o título, numero de faixas e o artista de cada CD no banco de dados. Podemos usar 3 interfaces para executar comandos SQL no banco de dados. A primeira delas é a interface Statement, que permite a execução dos comandos fundamentais de SQL (**SELECT**, **INSERT**, **UPDATE** ou **DELETE**). A interface PreparedStatement nos permite usufruir de SQL armazenado ou pré-compilado no banco, quando o banco de dados suportar este recurso. A terceira interface é CallableStatement, e permite executar procedimentos e funções armazenados no banco quando o banco suportar este recurso. Vejamos como utilizar a interface Statement. Nos próximos artigos sobre JDBC iremos investigar as outras.

Código:

```
// Após estabelecermos a conexão com o banco de dados
// Utilizamos o método createStatement de con para criar o Statement
Statement stm = con.createStatement();

// Vamos executar o seguinte comando SQL :
String SQL = "Select titulo, autor, total_faixas from MeusCDs";
```

A interface ResultSet permite colher os resultados da execução de nossa query no banco de dados. Esta interface apresenta uma série de métodos para prover o acesso aos dados:

Código:

```
// Definido o Statement, executamos a query no banco de dados
ResultSet rs = stm.executeQuery(SQL);

// O método next() informa se houve resultados e posiciona o cursor do banco
// na próxima linha disponível para recuperação
// Como esperamos várias linhas utilizamos um laço para recuperar os dados
while(rs.next())
{
  // Os métodos getXXX recuperam os dados de acordo com o tipo SQL do dado:
  String tit = rs.getString("titulo");
  String aut = rs.getString("autor");
  int totalFaixas = rs.getInt("total_faixas");

  // As variáveis tit, aut e totalFaixas contém os valores retornados
  // pela query. Vamos imprimí-los

  System.out.println("Titulo: "+tit+" Autor: "+aut+" Tot. Faixas: "+totalFaixas);
}
```

E nosso acesso está terminado. O importante agora é liberar os recursos alocados pelo banco de dados para a execução deste código. Podemos fazer isso fechando o Statement, que libera os recursos associados à execução desta consulta mas deixa a conexão aberta para a execução de uma próxima consulta, ou fechando diretamente a conexão, que encerra a comunicação com o banco de dados. Para termos certeza de que vamos encerrar esta conexão mesmo que uma exceção ocorra, reservamos o fechamento para a cláusula finally() do tratamento de exceções.

Código:

```
finally
{
  try
```

```
    {
      con.close();
    }
    catch(SQLException onConClose)
    {
       System.out.println("Houve erro no fechamento da conexão");
       onConClose.printStackTrace();
    }
  }
```

Uma classe para listar uma tabela
Vamos colocar tudo isso em conjunto para termos uma visão em perspectiva:

Código:

```
package wlss.jdbcTutorial;

 import java.sql.*;

 class Exemplo1
 {

   public static void main(String args[])
   {


   // A captura de exceções SQLException em Java é obrigatória para usarmos JDBC.
   // Para termos acesso ao objeto con, ele deve ter um escopo mais amplo que o bloco try

   Connection con = null;

   try
   {
      // Este é um dos meios para registrar um driver
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").getInstance();

      // Registrado o driver, vamos estabelecer uma conexão
      con = DriverManager.getConnection("jdbc:odbc:meusCdsDb","conta","senha");

      // Após estabelecermos a conexão com o banco de dados
      // Utilizamos o método createStatement de con para criar o Statement
      Statement stm = con.createStatement();

      // Vamos executar o seguinte comando SQL :
      String SQL = "Select titulo, autor, total_faixas from MeusCDs";

      // Definido o Statement, executamos a query no banco de dados
      ResultSet rs = stm.executeQuery(SQL);

      // O método next() informa se houve resultados e posiciona o cursor do banco
      // na próxima linha disponível para recuperação
      // Como esperamos várias linhas utilizamos um laço para recuperar os dados
      while(rs.next())
      {

        // Os métodos getXXX recuperam os dados de acordo com o tipo SQL do dado:
        String tit = rs.getString("titulo");
        String aut = rs.getString("autor");
        int totalFaixas = rs.getInt("total_faixas");

        // As variáveis tit, aut e totalFaixas contém os valores retornados
        // pela query. Vamos imprimí-los

        System.out.println(48:"Titulo: "+tit+" Autor: "+aut+"49:            Tot. Faixas:
"+totalFaixas);
      }

   }
   catch(SQLException e)
   {
```

```
        // se houve algum erro, uma exceção é gerada para informar o erro
        e.printStackTrace(); //vejamos que erro foi gerado e quem o gerou
    }
    finally
    {
      try
      {
        con.close();
      }
      catch(SQLException onConClose)
      {
         System.out.println("Houve erro no fechamento da conexão");
         onConClose.printStackTrace();
      }
    } // fim do bloco try-catch-finally
    } // fim da main

  } // fim de nosso primeiro exemplo !
```

Na próxima parte deste artigo iremos analisar as extensões introduzidas pela API 2.1 e as interfaces PreparedStatement e CallableStatement.

Powered by JavaFree
www.javafree.com.br