Javafree.org

Acessando banco de dados em Java (PARTE 1)

comentários: 24

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

```
view plain print ?
01. 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:

Publicado por
daltoncamargo

Últimos artigos

Trabalhando com arquivos Microsoft Word em Java

Torre de Brahma (Torre de Hanoi) - Simulador em Swing

Exemplo simples de Singleton

NetBeans 6.5, jtree - treeModel e defaultMutableTreeNode

Master Detail baseado em JComboBox

Computação Soberana

Exemplo de aplicação JSF usando o NetBeans 6.5 em 20 passos

Exemplo de CardLayout no netbeans6.5

Pegando o endereço MAC do computador.

Instalando e configurando o Java no Linux Ubuntu

Aplicando Transparencia no panel

Apresentar Resultado de Consulta SQL em JTable

Gerador de Código para JTable

Componente Visual Para Selecionar Datas

Vagas em TI

Blogs em TI

```
view plain print ?
01.    try //A captura de exceções SQLException em Java é obrigatória para usarmos JDBC.
02.    {
03.        // Este é um dos meios para registrar um driver
04.        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").getInstance();
05.
06.        // Registrado o driver, vamos estabelecer uma conexão
07.        Connection con = DriverManager.getConnection("jdbc:odbc:meusCdsDb","conta","senha"
08.    }
09.    catch(SQLException e)
10.    {
11.        // se houve algum erro, uma exceção é gerada para informar o erro
12.        e.printStackTrace(); //vejamos que erro foi gerado e quem o gerou
13.    }
```

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.

```
view plain print ?
01.    // Após estabelecermos a conexão com o banco de dados
02.    // Utilizamos o método createStatement de con para criar o Statement
03.    Statement stm = con.createStatement();
04.
05.    // Vamos executar o seguinte comando SQL :
06.    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:

```
view plain print ?
01.    // Definido o Statement, executamos a query no banco de dados
02.    ResultSet rs = stm.executeQuery(SQL);
03.
04.    // O método next() informa se houve resultados e posiciona o cursor do banco
05.    // na próxima linha disponível para recuperação
06.    // Como esperamos várias linhas utilizamos um laço para recuperar os dados
07.    while(rs.next())
08.    {
09.       // Os métodos getXXX recuperam os dados de acordo com o tipo SQL do dado:
10.       String tit = rs.getString("titulo");
11.       String aut = rs.getString("autor");
12.       int totalFaixas = rs.getInt("total_faixas");
13.
14.       // As variáveis tit, aut e totalFaixas contém os valores retornados
15.       // pela query. Vamos imprimí-los
16.
17.       System.out.println("Titulo: "+tit+" Autor: "+aut+" Tot. Faixas: "+totalFaixas);
18.    }
```

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.

```
view plain print ?
01.    finally
02.    {
03.       try
04.       {
05.          con.close();
06.       }
07.       catch(SQLException onConClose)
08.       {
09.           System.out.println("Houve erro no fechamento da conexão");
10.           onConClose.printStackTrace();
11.       }
12.    }
```

Uma classe para listar uma tabela

Vamos colocar tudo isso em conjunto para termos uma visão em perspectiva:

view plain print ?

```
01.  package wlss.jdbcTutorial;
02.
03.    import java.sql.*;
04.
05.    class Exemplo1
06.    {
07.
08.       public static void main(String args[])
09.       {
10.
11.
12.       // A captura de exceções SQLException em Java é obrigatória para usarmos JDBC.
13.       // Para termos acesso ao objeto con, ele deve ter um escopo mais amplo que o bloco
14.
15.       Connection con = null;
16.
17.       try
18.       {
19.           // Este é um dos meios para registrar um driver
20.           Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").getInstance();
21.
22.           // Registrado o driver, vamos estabelecer uma conexão
23.           con = DriverManager.getConnection("jdbc:odbc:meusCdsDb","conta","senha");
24.
25.           // Após estabelecermos a conexão com o banco de dados
26.           // Utilizamos o método createStatement de con para criar o Statement
27.           Statement stm = con.createStatement();
28.
29.           // Vamos executar o seguinte comando SQL :
30.           String SQL = "Select titulo, autor, total_faixas from MeusCDs";
31.
32.           // Definido o Statement, executamos a query no banco de dados
33.           ResultSet rs = stm.executeQuery(SQL);
34.
35.           // O método next() informa se houve resultados e posiciona o cursor do banco
36.           // na próxima linha disponível para recuperação
37.           // Como esperamos várias linhas utilizamos um laço para recuperar os dados
38.           while(rs.next())
39.           {
40.
41.              // Os métodos getXXX recuperam os dados de acordo com o tipo SQL do dado:
42.              String tit = rs.getString("titulo");
43.              String aut = rs.getString("autor");
44.              int totalFaixas = rs.getInt("total_faixas");
45.
46.              // As variáveis tit, aut e totalFaixas contém os valores retornados
47.              // pela query. Vamos imprimí-los
48.
49.              System.out.println(48:"Titulo: "+tit+" Autor: "+aut+"49:                Tot.
50.           }
51.
52.       }
53.       catch(SQLException e)
54.       {
55.           // se houve algum erro, uma exceção é gerada para informar o erro
56.           e.printStackTrace(); //vejamos que erro foi gerado e quem o gerou
57.       }
58.       finally
59.       {
60.          try
61.          {
62.             con.close();
63.          }
64.          catch(SQLException onConClose)
65.          {
66.              System.out.println("Houve erro no fechamento da conexão");
67.              onConClose.printStackTrace();
68.          }
69.       } // fim do bloco try-catch-finally
70.       } // fim da main
71.
72.    } // 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.

Leia também:
Acessando Banco de Dados em Java (PARTE 2)

Acessando Banco de Dados em Java (PARTE 3)

comentários: 24

O JavaFree.org é uma comunidade java formada pela coolaboração dos desenvolvedores da tecnologia java. A publicação de artigos além de ajudar a comunidade java, ajuda a dar maior visibilidade para o autor. Contribua conosco.

Design: Luka Cvrk · Desenvolvimento: Dalton Camargo