Persistência com JDBC e JPA

Aula 1

Marcos Alberto Lopes da Silva

(malopes21@gmail.com)

Pós-Java turma 2017-2018!!!

Sumário

- Introdução do curso
- Foco em dados – JDBC na prática
- API JDBC
- Comandos em lote (batch)
- Invocando Stored Procedures
- Lendo Metadados
- Dicas para melhoria JDBC

Siglas e acrônimos

- BD = banco de dados;
- POO = Programação Orientada a Objetos;
- SGBDR = Sistema de gerenciamento de bancos de dados relacionais;
- DAO = Data Acess Object;
- JDBC = Java Database Connectivity;
- JPA = Java Persistence API;
- ORM = object-relational mapping ou mapeamento objeto-relacional;
- DDL – data definition language;
- DML – data manipulation language

O que a disciplina abrange

- Persistência de dados (JDBC);
- Mapeamento Objeto Relacional;
- Persistência de objetos (JPA/engine);
- Persistência em arquivos (arquivos tipados, serialização, xml);
- Boas práticas.

O que a disciplina não abrange

- Recursos específicos de BDs;
- Frameworks e especificações fora da camada de persistência (EJBs, JSF, Swing, etc..)
- Novas tendências noSql como Db4o;

Foco em dados

- JDBC ”na unha”
- Performance

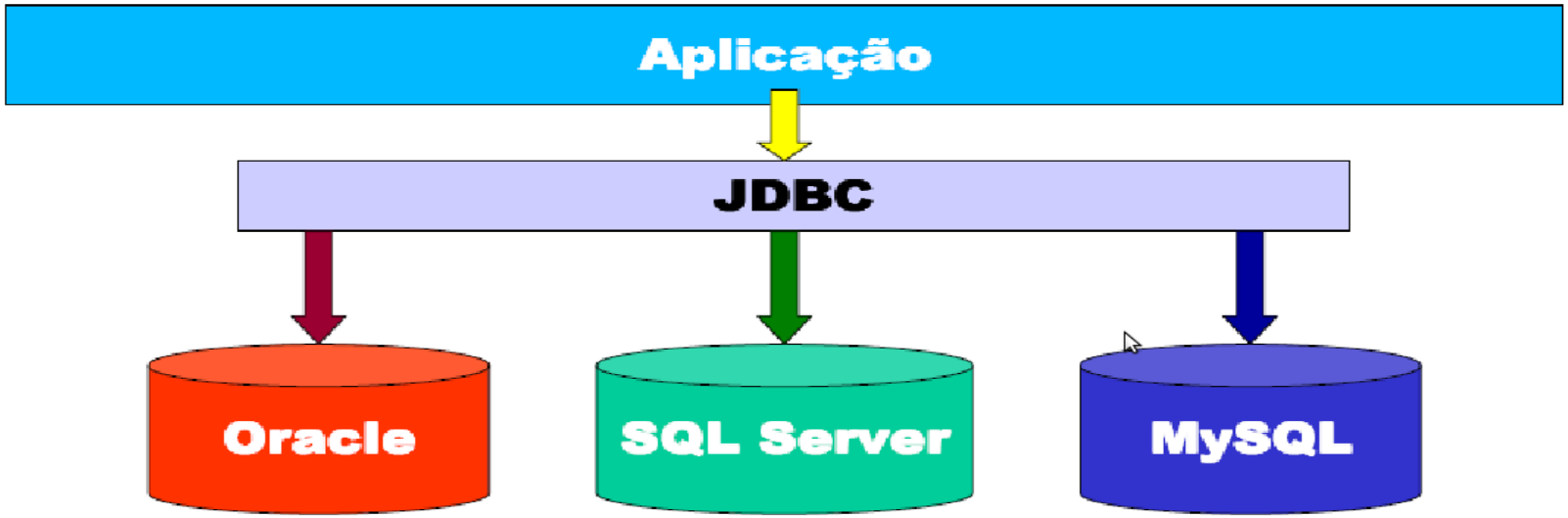
Foco em dados

- Péssima produtividade;

- Organização do acesso a dados na mão

JDBC

- API de baixo nível, modelada segundo o padrão ODBC da Microsoft;



Por que JDBC é uma API?

A API JDBC

- Provê um conjunto de interfaces (java.sql.*) para acesso ao BD;
- Uma implementação em particular destas interfaces é chamada de **d*river***;
- Cada BD possui um driver JDBC específico

- Padronização do acesso!

A API JDBC

- Conecta em qualquer BD;
- Submete comandos Sql recuperando resultados;
- Permite acesso aos metadados do banco;
- É a API base para vários frameworks realizarem comunicação com o BD;
- Não necessita de nenhuma configuração prévia ou instalação de cliente nativo do banco

Drivers JDBC

- A Api necessita, além da JVM, de um driver JDBC para o BD escolhido;
- São bibliotecas Java independentes de SO;
- As bibliotecas devem estar no classpath da aplicação para que esta seja executada (não precisa para ser compilada)

http://dev.mysql.com/downloads/connector/j/

http://www.oracle.com/technetwork/database/features/jdbc/index-091264

http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=2

Exercício 1

- Criar um BD qualquer (sugestões: JavaDB, Mysql, Oracle, ...) com uma tabela Conta (id:int, nome:varchar, agencia:int, senha:int)
- Conectar nesse BD usando recursos da IDE NetBeans e/ou Eclipse
- Emitir comandos de DDL e DML para o banco

- http://www.vogella.com/tutorials/ApacheDerby/article.html

JDBC - Carregar o driver

- Mysql

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

- Oracle

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

- O classloader da JVM irá carregar essa classe em tempo de execução

Abrir conexão com o BD

- Mysql

  Connection conexao =
  DriverManager.getConnection(”jdbc:mysql://localhost/dbartigos”, ”root”, ”unitri”);

- *Oracle*

  Connection conexao =
  DriverManager.getConnection(”jdbc:oracle:thin:@96.1.1.3:1100:ERPTST”, ”root”, ”unitri”);

- DriverManager perguntará para cada driver registrado se este aceita a String de conexão

A interface Statement

- Permite executar um comando qualquer (DDL, DML)

  Statement stm = conexao.createStatement();

- Queries: método *executeQuery()*

  ResultSet rs = stm.executeQuery("SELECT coluna1, coluna2, coluna3 FROM tabela");

- Inserts, updates e deletes: método *executeUpdate()*

  int registrosAfetados = stm.executeUpdate(”Update tabela set coluna1 = valor where coluna2 = valor”)

A interface Statement

- *ExecuteQuery() retorna um ResultSet, um cursor que permite ler, linha a linha os resultados da consulta*

```
while (rs.next()) {
    String coluna1 = rs.getString("coluna1");
    int coluna2 = rs.getInt("coluna2");
    Date coluna3 = rs.getDate("coluna3");
}
```

- *Pode ser utilizado a posição ou o nome da coluna. Em bancos como mysql, a posição resulta em um pequeno ganho de performance*

ResultSet é um cursor

- ResultSet é um cursor, ou seja, não ”sabe” a quantidade de linhas em um resultado até que este seja inteiramente percorrido

Fechar conexão

- *Statement, ResultSet e Connection devem ser fechadas com o método close();*
- *Estes objetos representam recursos do Sistema Operacional e do BD;*
- O Garbage Collection não fará a coleta automática;
- Todo código que inclui chamadas JDBC para fechar conexões devem estar dentro de um bloco **try-finally**

Fechar conexão

```
public void fazConsulta() {
    Connection conexao = ConexaoJDBC.getConexao();
    Statement comando = null;
    ResultSet resultado = null;
    try {
        //faz query
    } catch (SQLException e) {
        System.err.println("Erro de banco de dados "+e.ge
    } catch (Exception e) {
        System.err.println("Erro inesperado "+e.getMessag
    } finally {
        try {
            if (resultado != null ) {
                resultado.close();
            }
            if (comando != null) {
                comando.close();
            }
            if (conexao != null) {
                conexao.close();
            }
        } catch(SQLException e) {
            //descarta a excessão
        }
    }
}

}
```

Tipos de dados Sql X Java

- Number ou int32 → Integer;
- Char ou varchar → String;
- Tipos numéricos (decimal) → java.math.BigDecimal;
- Timestamp – java.util.Date;
- Date – java.sql.Date;
- Time - java.sql.Time.

Transações

- Para evitar que uma aplicação Java possa controlar quando suas transações iniciam e terminam, deve-se desligar o auto commit

  conexao.setAutoCommit(false);

- Caso a aplicação ”esqueça” de sinalizar o final de uma transação, o BD irá realizar um **roolback** ao fim da conexão ou após um timeout;

Transações

```
public void fazAtualizacaoImportante() {
    Connection conexao = null;
    Statement comando = null;

    try {
        conexao = getConexao();
        conexao.setAutoCommit(false);

        comando = conexao.createStatement();
        int linhas = comando.executeUpdate("update desconto set percentual = percentual - 5 where estado = 'SP'");

        if (linhas == 0) {
            throw new RuntimeException("Oh meu Deus! Não atualizou!!");
        }

        comando.executeUpdate("update promocao set meta = 100 where estado = 'SP'");

        conexao.commit();

    } catch (Exception e) {
        //avalia excecoes
    } finally {
        //fecha conexoes
    }
}
```

Transações

- Caso a aplicação mantenha a conexão ativa mesmo em erros, deverá chamar explicitamente roolback() em blocos catch() para evitar ”efeitos colaterais”;
- Caso a conexão seja encerrada em finally, não é necessário o roolback

```
String sql = "insert into Conta (id, nome, agencia, senha) values (?, ?, ?, ?)";
try {
    con = ConnectionFactory.getConnection();
    stmt = con.prepareStatement(sql);
    stmt.setInt(1, 2);    stmt.setString(2, "Marcos Lopes Silva");
    stmt.setInt(3, 230); stmt.setString(4, "123456");
    int n = stmt.executeUpdate();
    con.commit();
} catch (Exception ex) {
    logger.log(Level.SEVERE, "Erro: {0}.", ex);
    if(con != null) { con.rollback(); }   //não necessário aqui!!!
} finally {
    ConnectionFactory.closeConnection(con, stmt, null);
}
```

Statements otimizados e simplificados

- Comandos SQL com concatenação de Strings?

Statements otimizados e simplificados

- PreparedStatement – Sqls parametrizados

```
public void getProdutos(int codigoProduto) {
    Connection conexao = null;
    PreparedStatement comando = null;
    ResultSet resultado = null;

    try {
        conexao = getConexao();

        comando = conexao.prepareStatement("select pro.despro from tbProduto pro where pro.codpro = ?");
        comando.setInt(1, codigoProduto);

        resultado = comando.executeQuery();

        while (resultado.next()) {
            System.out.println(resultado.getString("despro"));
        }

    } catch (Exception e) {
        try {
            conexao.rollback();
        } catch (SQLException e2) {

        }
        //avalia excecoes
    } finally {
```

Statements otimizados e simplificados

- Análise sintática do comando SQL, incluindo verificações de nomes de tabelas e de colunas referenciadas pelo comando;
- Construção do plano de acesso → compilação e otimização do comando SQL para sua execução pelo BD;
- Execução do plano de acesso, modificando ou recuperando registros

Statements otimizados e simplificados

- PreparedStatement executa as 2 primeiras etapas apenas 1 vez;
- Parâmetros de PreparedStatement são como argumentos para métodos Java;
- Não é possível dar nomes aos parâmetros. Se o mesmo valor repetir várias vezes, deve ser passado várias vezes

Exercício 2

- Implementar um projeto Java stand-alone para acessar um BD (sugestão: usar o JavaDB ou o MySql) e construir testes para as operações CRUD de uma entidade Conta (id:int, nome:varchar, agencia:int, senha:int)
- Não é necessário montar o projeto com base no padrão DAO
- Não é necessário montar interface gráfica

Statements ou PreparedStatements?

- PreparedStatements resolve 3 problemas:

  1) Simplifica a representação dos parâmetros;

  2) Aumenta o desempenho;

  3) Suporta updates em batch

- Caches do BD e do driver são de longo prazo. PreparedStatement é mais rápido para muitos inserts, Statement é mais rápido para poucos inserts;

- Planos de execução podem ser impactados com o uso de Statement ou PreparedStatement.

Statements ou PreparedStatements?

- Statements são recomendados para consultas dinâmicas. Ex: aplicação que permite ao usuário digitar comandos sql arbitrários;
- Consultas parcialmente dinâmicas:

  3 parâmtros opcionais? 8 combinações

  16 parâmetros opcionais? 65536 combinações

Operações em Batch

- Ler ou atualizar um grande volume de dados de uma só vez;
- Gera operações semelhantes, que variam somente quanto aos argumentos, e depois são despachadas de uma só vez ao banco;
- Em alguns drivers JDBC, pode existir limite de número de chamadas addBatch();
- Deve ser evitado a acumulação excessiva de itens para um único executeBatch()

Operações em Batch

```
public void atualizarProdutos(int codigoFornecedor,List<Produto> produtos) {
    Connection conexao = null;
    PreparedStatement comando = null;

    try {
        conexao = getConexao();
        comando = conexao.prepareStatement("update tbProduto set codfrn = ? where codpro = ?");
        comando.setInt(1,codigoFornecedor);
        for (Produto produto: produtos) {
            comando.setInt(2, produto.getCodigo());
            comando.addBatch();
        }
        comando.executeBatch();
        conexao.commit();
    } catch (Exception e) {
        try {
            conexao.rollback();
        } catch (SQLException e2) {

        }
        //avalia excecoes
    } finally {
        //fecha conexoes
    }
}
```

- O código do fornecedor pode ser colocado fora do loop (minimiza o overhead da JVM).

Exercício 3

- Implementar o exemplo de atualização em batch citado no slide anterior;
- A lista de produtos pode ser criada em memória;
- Tente medir o ganho de desempenho aplicando uma atualização em lote (batch) de cerca de 30 produtos comparando com a mesma funcionalidade usando Statements separados para cada operação.

Metadados de uma consulta

- Necessidade de descobrir dinamicamente quais são os campos retornados na consulta;

```
ResultSetMetaData metaDados =
   resultado.getMetaData();

for (int i = 0; i < metaDados.getColumnCount();i++) {

    System.out.println(getColumnName(i)
   +":"+resultado.getString());

}
```

- Não é necessário fechar os objetos de ResultSetMetaData

Metadados do BD

- Interface **DatabaseMetaData**, chamada pelo método **getMetaData()** de **Connection**;

  DatabaseMetaData meta = conexao.getMetaData();

  ResultSet resultado = meta.getColumns(null,null,”tbProduto,”%”);

- **getColumns()** funciona como uma máscara para nomes de objetos.

Metadados do BD

```
While (resultado.next()) {
   String nome = resultado.getString("COLUMN_NAME");
   String tipo = resultado.getString("COLUMN_TYPE");
   Boolean notNull = (resultado.getInt("NULLABLE") ==
    DatabaseMetaData.columnNotNulls) ? true:false;
}
```

Exercício 4

- Implementar o exemplo de ler medados de um BD qualquer com pelo menos 2 tabelas

Stored Procedures

```
CallableStatement stmt = con.prepareCall("{call
    nomeDaProcedure(?,?,?)}";
stmt.setString("teste");
…
stmt.execute();
```

- http://www.devmedia.com.br/stored-procedures-no-mysql/29030
- http://www.javacodegeeks.com/2013/09/java-stored-procedures-in-java-db.html
- http://carminedimascio.com/2013/07/java-stored-procedures-with-derby/

Exercício 5

- Criar e implantar um Stored Procedure num BD qualquer (Mysql, Derby, Oracle, SQLServer, etc)
- Implementar uma classe de teste para invocar o Procedimento Armazenado criado
-

```
DELIMITER $$

CREATE PROCEDURE Selecionar_Alunos(IN quantidade INT)
BEGIN
        SELECT * FROM aluno
        LIMIT quantidade;
END $$
DELIMITER ;
```

Dicas para melhorar o desempenho com JDBC

1) Ao executar selects que retornam **milhares de registros**, métodos getXxx() indexados pelo nome da coluna devem ser evitados. É melhor utilizar índices numéricos;

2) Manter driver JDBC atualizado. Ex: alguns BDs possuem drivers otimizados para versões mais recentes do Java, utilizando java.nio para transferência mais eficiente de dados pela rede;

3) Desista da independência de Bds. O máximo que se pode fazer é criar DAOs para isolar o local de alterações. Ex: consultas envolvendo paginações

Dicas para melhorar o desempenho com JDBC

4) Opções de isolamento de transações (ACID)

- Escolher transação mais leve que não afete integridade;

conexao.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED)

- Só evita dirty reads (dados de transações que ainda não fizeram commit)

- http://docs.oracle.com/javase/tutorial/jdbc/basics/transactions.html

Dicas para melhorar o desempenho com JDBC

4) Opções de isolamento de transações (ACID)

*conexao.setTransactionIsolation(Connection.TRANSACTION_REPEATABLE_READ) *** default*

- Adiciona proteção contra non-repeatable reads;

Dicas para melhorar o desempenho com JDBC

4) Opções de isolamento de transações (ACID)

conexao.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE)

- Adiciona proteção contra phantom reads – garante o ACID;

Dicas para melhorar o desempenho com JDBC

5) Caso não vá efetuar alterações na base, setReadOnly(true) para Connection fará otimizações como não pré-alocar caches para escrita;

- 6) Opções de ResultSet

rs.setFetchDirection(ResultSet.FETCH_FORWARD)

Ordem normal

rs.setFetchDirection(ResultSet.FETCH_REVERSE)

Ordem reversa

rs.setFetchDirection(ResultSet.FETCH_UNKNOWN)

Ordem aleatória

Dicas para melhorar o desempenho com JDBC

7) Opções de ResultSet para statement

CONCUR_UPDTABLE – liberará updateXXX() e deleteRow(), mas fará locks no banco;

- TYPE_FORWARD_ONLY – limite o resultSet com next;
- TYPE_SCROLL_INSENSITIVE e TYPE_SCROLL_SENSITIVE – utilização de métodos absolute(),first() e last()

Dicas para melhorar o desempenho com JDBC

```
PreparedStatement stmt =
    conn.prepareStatement("Select * from funcionario
    where nome = ? for
    update",ResultSet.TYPE_FORWARD_ONLY,ResultSet
    .CONCUR_UPDATABLE);

stmt.setString(1,"Carlos");

ResultSet rs = stmt.executeQuery();

rs.next();

double salario = rs.getDouble("salario");

salario = calcAumento(salario);

rs.updateDouble("salario",salario);

System.out.println("novo salario = "+salario);
```

Dicas para melhorar o desempenho com JDBC

7) CLOSE_CURSORS_AT_COMMIT – determina que o cursor (resultSet) será fechado automaticamente quando a transação for confirmada por Connection.commit();

- HOLD_CURSORS_OVER_COMMIT – não fecha o cursor. Ideal para transações onde precisa do mesmo cursor utilizado antes e após o commit().

Exercício 6

- Implementar uma classe de teste para demonstrar o funcionamento adequado usando transações para operações com mais de um comando no BD.

Referências bibliográficas

[1] ”Dados e mapeamento” - Java Magazine 42;

[2] ”Persistência Turbinada I” - Java Magazine 25;

[3] ”Persistência Turbinada II” - Java Magazine 26;

[4] ”JDBC de ponta a ponta parte 1” - Java Magazine 41;

[5] ”JDBC de ponta a ponta parte 2” - Java Magazine 42;

[6] ”Introdução ao JDBC” http://www.guj.com.br/articles/7