



Edição 49

JDBC além do básico
Conheça recursos para melhorar
o desempenho de seus sistemas

Hibernate 4 na prática
Aprenda a persistir os dados de sua aplicação



PROGRAME COM BOAS PRÁTICAS

Estruturas de Dados e Padrões de Projeto



DESENVOLVA PARA CORPORAÇÕES OU PARA O DIA A DIA COM JAVA

FORMAÇÃO DESENVOLVEDOR JAVA

DOMINE O DESENVOLVIMENTO DE APLICAÇÕES ORIENTADAS A OBJETO COM **UML** E APRENDA TÓPICOS AVANÇADOS DE PROGRAMAÇÃO COMO O USO DE **SERVLETS, JSP, FRAMEWORKS WEB DENTRO DO PADRÃO MVC.**

➤ A linguagem Java está presente em complexos sistemas como em aplicativos mobile e desktop, servindo a milhares de pessoas. Inicie sua carreira e certifique-se no instituto que valoriza seu currículo. Faça Infnet!

CERTIFICAÇÕES:

➤ Oracle Certified Professional, Java SE 6 Programmer

➤ Oracle Certified Professional, Java EE 5 Web Component Developer



Sumário

Conteúdo sobre Boas Práticas

06 - JDBC além do básico

[Luís Fernando Orleans]

Conteúdo sobre Boas Práticas

15 - Programando com estruturas de dados e padrões de projeto

[Miguel Diogenes Matrakas]

Artigo no estilo Solução Completa

24 - Persistência de dados com Hibernate 4

[Carlos Alberto Silva e Eduardo Augusto Silvestre]

CURSOS ONLINE

A Revista Java Magazine oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.



CONHEÇA ALGUNS DOS CURSOS:

- Curso de noSQL (Redis) com Java
- Curso Básico de JDBC
- Java Básico: Aplicações Desktop
- JSF com Primefaces
- Conhecendo o Apache Struts

Para mais informações :

www.devmedia.com.br/curso/javamagazine

(21) 3382-5038



Edição 49 • 2015 • ISSN 2173625-4



Assine agora e tenha acesso a todo o conteúdo da DevMedia:
www.devmedia.com.br/mvp

EXPEDIENTE

Editor

Eduardo Spínola (eduspinola@gmail.com)

Consultor Técnico Diogo Souza (diagosouzac@gmail.com)

Produção

Jornalista Responsável Kaline Dolabella - JP24185

Capa e Diagramação Romulo Araujo

Atendimento ao leitor

A DevMedia possui uma Central de Atendimento on-line, onde você pode tirar suas dúvidas sobre serviços, enviar críticas e sugestões e falar com um de nossos atendentes. Através da nossa central também é possível alterar dados cadastrais, consultar o status de assinaturas e conferir a data de envio de suas revistas. Acesse www.devmedia.com.br/central, ou se preferir entre em contato conosco através do telefone 21 3382-5038.

Publicidade

publicidade@devmedia.com.br – 21 3382-5038

Anúncios – Anunciando nas publicações e nos sites do Grupo DevMedia, você divulga sua marca ou produto para mais de 100 mil desenvolvedores de todo o Brasil, em mais de 200 cidades. Solicite nossos Media Kits, com detalhes sobre preços e formatos de anúncios.

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique à vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site Easy Java Magazine, entre em contato com o editor, informando o título e mini-resumo do tema que você gostaria de publicar:



EDUARDO OLIVEIRA SPÍNOLA

eduspinola.wordpress.com

@eduspinola / @Java_Magazine

CURSOS ONLINE

A Revista SQL Magazine oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.

Feita para Desenvolvedores de Software e DBAs



CONHEÇA OS CURSOS MAIS RECENTES:

- **Cursos: Curso de noSQL (Redis) com Java**
- **Desenvolvimento para SQL Server com .NET**
- **Curso PostgreSQL - Treinamento de banco de dados (Curso Básico)**

Para mais informações :

<http://www.devmedia.com.br/cursos/banco-de-dados>
(21) 3382-5038



DEV MEDIA



CRIANDO APLICAÇÕES INOVADORAS QUE VÃO MUDAR O MUNDO?

DESENVOLVA COM AS FERRAMENTAS QUE OS MELHORES DESENVOLVEDORES
USAM: GIT, REDMINE, JENKINS, SONAR, MAVEN E MUITO MAIS.

A suite ToolsCloud é um conjunto integrado das melhores ferramentas para desenvolvimento de software do mercado.

Um ambiente pronto para usar e com suporte, que os desenvolvedores curtem e os gerentes se surpreendem.

- Aumenta a produtividade;
- Facilita a colaboração e comunicação;
- Maior visibilidade e acompanhamento;
- Melhor planejamento e mais agilidade.

Na suite ToolsCloud você tem a sua disposição:

- As ferramentas mais usadas, integradas e de uso imediato;
- Repositório de código (git e svn), integração contínua (Jenkins/Hudson), Gerência de pendências (Redmine), e mais Maven, Nexus, Sonar;
- Infraestrutura robusta, em ambiente de nuvem, suporte e segurança;
- Facilita práticas de agilidade e lean startups;
- Ambiente isolado, customizável e flexível;
- Colaboração garantida: wiki, fóruns de discussão, chat (IRC).

Acompanhe a Toolscloud:

 toolscloud@toolscloud.com



twitter.com/toolscloud



ToolsCloud
toolscloud.com

JDBC além do básico

Entenda os aspectos importantes para otimizar o desempenho na persistência de dados JDBC

Praticamente todos os sistemas que são desenvolvidos hoje em dia utilizam algum tipo de repositório lógico para armazenar as informações preenchidas pelos usuários. Este passo é necessário, pois torna possível manter o estado do sistema entre suas falhas, reinícios e atualizações de hardware e software. Dentre estes tipos de repositórios, Sistemas Gerenciadores de Bancos de Dados Relacionais (SGB-DR) ainda se destacam como alternativa principal, dada a maturidade da tecnologia e seu formalismo matemático.

No Java, diversas especificações foram criadas para facilitar a transformação de objetos em um formato relacional, composto primariamente de tabelas, linhas e colunas. Os documentos que definiram como utilizar EJB 1.x, EJB 2.x, JDO (agora sob responsabilidade da Apache) e, mais recentemente, JPA, têm em comum tirar do programador a necessidade de aprender dois mundos distintos (Java e SQL), introduzindo camadas intermediárias que fazem o chamado mapeamento objeto-relacional (MOR) de forma transparente. Outra similaridade entre estas tecnologias reside no fato de que elas usam a API da *Java Database Connectivity* (JDBC) para realizar conexões com os SGBDs, enviar consultas, delimitar transações, etc. De um modo geral, as ferramentas de mapeamento objeto-relacional realizam bons trabalhos, simplificando a vida dos desenvolvedores que desejam acessar SGBDs (apesar de fomentar a criação dos indesejados métodos de acesso nos *beans*, os conhecidos `getXXXX()` e `setXXXX(...)`) – entretanto, este é um assunto para outro artigo).

Contudo, existem situações onde as ferramentas de MOR não conseguem obter um bom desempenho. A seguir, são descritos apenas alguns dos problemas mais comuns em projetos onde o JPA é utilizado:

- **Geração de consultas SQL que não utilizam índices:** qualquer SQL minimamente complexo a ser gerado (ex.: sub-consultas, junções externas, etc.) tem alta probabilidade de ser ineficiente;
- **Carregamento batch:** apesar de possuir mecanismos que atenuem este problema, a inserção de grande quantidade de dados no SGBD não é aconselhável com o JPA, uma vez que ela não foi projetada para essa finalidade;

Fique por dentro

Este artigo é útil por apresentar tópicos que facilitam a vida dos desenvolvedores que resolveram ter maior controle sobre o código que acessa o banco de dados utilizando JDBC. No texto são analisados alguns conceitos avançados desta tecnologia, como gerenciamento de transações, utilização de savepoints, diferenças entre Statement e PreparedStatement, utilização de cursores, carregamento batch e outros. Ademais, sempre que pertinente, é apontado onde o desenvolvedor pode aumentar o desempenho de seu sistema com o uso destes recursos.

- **Mapeamento de tabelas que possuem chaves compostas:** as chaves compostas sempre representaram um problema para o JPA. Quando puder evitá-las, faça-o;
- **Mapeamento do conceito de herança para o esquema relacional:** apesar de a herança ser uma característica muito poderosa da orientação a objetos, ela tem como ponto negativo o fato de aumentar consideravelmente o acoplamento entre as classes, o que dificulta muito a manutenção de um sistema. Ainda assim, é muito comum ver desenvolvedores criando classes que herdam de outras de forma não planejada, empobrecendo muito o projeto final. O JPA prevê alguns meios de contornar este problema, com as técnicas de mapeamento (tabela única por hierarquia, uma tabela para cada classe da hierarquia, etc.). Contudo, nenhum deles consegue satisfazer totalmente os requisitos de orientação a objetos e de normalização de dados. A melhor dica é estudar OO o suficiente para fugir da herança;
- **Geração de relatórios:** o JPA não foi projetado para utilização em ferramentas OLAP (*On-line Analytical Processing*), caindo novamente no problema de geração de consultas ineficientes;
- **Extenso número de artefatos de configuração a serem mantidos:** apesar de existirem ferramentas que mantêm o sincronismo entre a especificação das tabelas no SGBD e as anotações ou arquivos XML, é fácil perder o controle sobre esses artefatos conforme o projeto cresce.

Repare que alguns destes problemas são ocasionados pela diferença dos mundos relacional e orientado a objetos, outros são causados pela utilização da tecnologia para fins diferentes daqueles para os quais ela foi projetada. De todo modo, é comum encontrar situações onde o JPA não se encaixa no seu projeto.

Nestas situações, ao invés de perder um tempo precioso para aprender uma nova ferramenta, a utilização do bom e velho JDBC deve ser considerada. Para isso, os desenvolvedores Java devem ter conhecimentos mais aprofundados sobre os passos que um banco de dados realiza ao processar uma consulta SQL, bem como de algumas funcionalidades existentes nos SGBDRs que facilitam a construção de sistemas.

O grande sucesso da JPA, entretanto, passou a esconder um problema na formação dos desenvolvedores: a maioria deles desconhece algumas funções extremamente importantes do JDBC, pois aprenderam somente o básico: abrir conexão com o SGBD, executar uma consulta, iterar sobre o resultado e fechar a conexão.

Ciclo de vida de uma consulta SQL

Para compreender os conceitos avançados do JDBC, é necessário revisar como um SGBDR processa os comandos SQL recebidos. De forma abreviada, os passos realizados pelo SGBDR são os seguintes:

1. Recebimento do comando SQL;
2. Parsing do SQL;
3. Reescrita do SQL;
4. Busca pelo plano de execução mais eficiente;
5. Execução da consulta, com acesso aos dados.

Além dos passos mencionados, para todos os comandos SQL que envolvam atualização de dados ou de esquema, o controle transacional é executado como um módulo à parte, onde *locks* e *latches* (tipos de locks específicos do sistema operacional e que possuem, como característica principal, a rapidez para obtenção e liberação) são verificados e adquiridos quando necessários. Ainda são de responsabilidade deste módulo a manutenção do log de transação, onde as alterações ainda não confirmadas são mantidas, e do buffer, este contendo os dados lidos do disco.

Ficaram de fora os passos como controle de admissão, verificação de permissões de acesso, protocolos de comunicação com o cliente e modelo de tratamento de requisições concorrentes, por exemplo, de forma proposital. Tais conceitos, apesar de importantes, são de responsabilidade de administradores de banco de dados (DBAs) e pouco acrescentariam ao artigo.

A **Figura 1** mostra os componentes principais da arquitetura de um SGBDR. Para os leitores que quiserem se aprofundar sobre a arquitetura de SGBDRs modernos, o texto *Architecture of Database Systems*, indicado na seção **Links**, é um excelente ponto de partida.

JDBC básico

Neste tópico será mostrado um código simples de utilização da JDBC, no qual os passos mais básicos na interação com um banco de dados serão realizados. Esse código é apresentado na **Listagem 1**, que traz um método que deve recuperar, em ordem alfabética de nome, todos os clientes que pediram um determinado sabor de pizza.

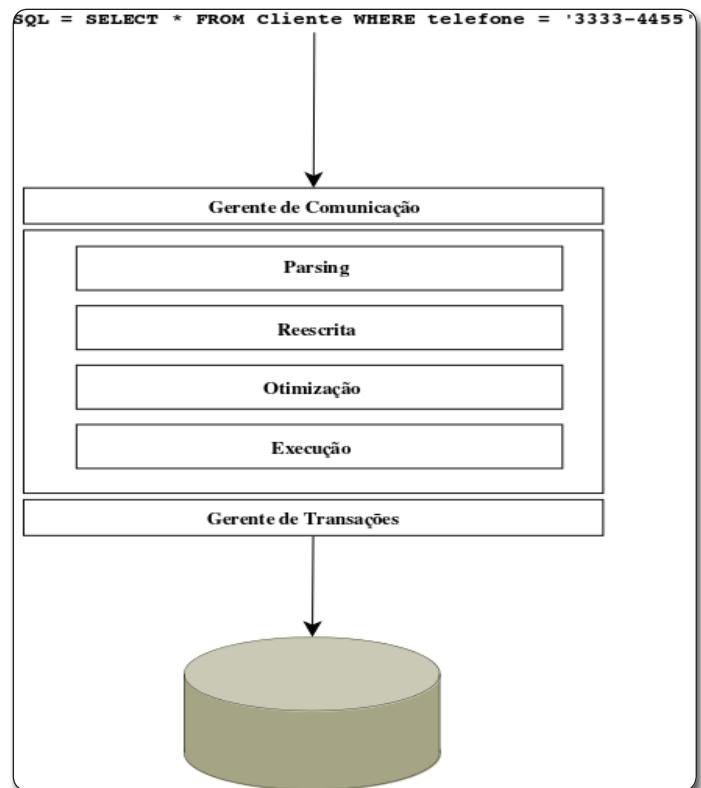


Figura 1. Ciclo de vida de uma consulta SQL

Listagem 1. Acessando um banco de dados com JDBC.

```

01 public Cliente recuperaClientes(String nomePizza) throws Exception{
02     String url = "jdbc:postgresql://127.0.0.1/jdbc_alem_basico";
03     String username = "java_magazine";
04     String passwd = "123";
05     Class.forName("org.postgresql.Driver").newInstance();
06     Connection conexao = DriverManager.getConnection(url, username, passwd);
07     Statement stmt = conexao.createStatement();
08     ResultSet rs = stmt.executeQuery("SELECT c.* FROM CLIENTE c, PEDIDO p
09     WHERE p.telefone = " + c.telefone AND p.nome_pizza = "+nomePizza+
10     " ORDER BY c.nome";
11     List<Cliente> listaClientes = new ArrayList<>();
12     while(rs.next()){
13         String nome = rs.getString("nome");
14         String telefone = rs.getString("telefone");
15         String endereco = rs.getString("endereco");
16         lista.add(new Cliente(telefone, nome, endereco));
17     }
18     rs.close();
19     stmt.close();
20     conexao.close();
21     return listaClientes;
22 }
  
```

Analisando esse código, nota-se que a partir da linha 02 iniciam os comandos da JDBC, mais especificamente na definição da URL de acesso – uma **String** que contém as informações de como o acesso aos dados será realizado. De um modo geral, a URL possui a seguinte forma:

jdbc:[nome_vendedor]://[ip servidor]:[porta]/[nome do banco de dados].

Alguns fabricantes adotam URLs com formatos diferentes e, por isso, recomenda-se checar na documentação do driver JDBC como esta String deve ser formada.

Já as linhas 03 e 04 explicitam o usuário do SGBDR e a sua senha. É importante ressaltar que o usuário deve ter permissão de acesso ao banco de dados que está tentando acessar.

A linha 05 é necessária para carregar a classe que possui os métodos de comunicação com o SGBDR no *ClassLoader*, o chamado *driver JDBC*, enquanto a linha 06 utiliza a classe **java.sql.DriverManager** para criar uma conexão com o banco de dados. Em seguida, nas linhas 07 e 08, um objeto do tipo **java.sql.Statement**, que pode ser entendido como uma “sessão de conversa” com o banco de dados, é criado e uma consulta SQL é enviada para ser executada, tendo seu resultado armazenado em um objeto do tipo **java.sql.ResultSet**. Este objeto pode ser visto como uma representação em memória de uma tabela, com um ponteiro que aponta para *antes da primeira linha*. Assim, ao chamar o método **next()**, anda-se com o ponteiro para a próxima linha e, caso atinja-se o fim do resultado, o método retorna o valor lógico **false**. Desta forma é possível iterar sobre o resultado da consulta e construir sua representação em forma de objetos.

Por fim, as linhas 16, 17 e 18 fecham os objetos **ResultSet**, **Statement** e **Connection**, respectivamente. As linhas 16 e 17 são redundantes, uma vez que o fechamento de uma conexão provoca um efeito em cascata, onde todos os Statements abertos por ela são fechados. Por sua vez, ao ter seu método **close()** invocado, o **Statement** fecha seu **ResultSet** caso este esteja aberto.

Nota:

Na versão 4.0 do JDBC ou posterior, o passo onde o driver JDBC é carregado não é necessário, pois toda classe que implemente a interface **java.sql.Driver** e que esteja no classpath é automaticamente carregada.

Problemas com o método `recuperaClientes()`

Apesar de ser pequeno e relativamente simples, o código da **Listagem 1** apresenta diversos problemas:

1. Concatenar strings ao comando SQL que vai ser executado é uma prática extremamente perigosa, devido ao risco de ataques do tipo *sql injection*;
2. Ao abrir uma conexão com um SGBDR, o comportamento padrão do JDBC é utilizar o gerenciamento de transações automático, o que significa que cada invocação de método de um objeto **Statement** é executada em uma transação isolada, podendo levar a problemas de consistência e/ou desempenho;
3. O SGBDR vai realizar, para cada chamada ao método **recuperaClientes()**, todos os passos do ciclo de vida da consulta. Contudo, como o comando enviado é sempre o mesmo, mudando somente o número de telefone, seria mais inteligente se mandássemos o SGBDR manter em cache o plano de execução mais eficiente, encontrado após a etapa de otimização. Assim, chamadas subsequentes

ao método iriam direto para a fase de execução da consulta, informando somente os parâmetros;

4. Considere que o resultado do SQL enviado é extremamente grande – tão grande que pode se tornar um problema para transmitir os dados do SGBDR para o cliente ou, o que é ainda pior, não caber na memória do cliente. Seria melhor se fosse possível recuperar pequenas porções do resultado, uma de cada vez.

Nos tópicos apresentados a seguir veremos como é possível resolver estes problemas.

Usando PreparedStatements

O código de exemplo da **Listagem 1** utiliza um objeto do tipo **java.sql.Statement** para enviar comandos SQL para o SGBDR. A utilização desta classe força a realização de todos os passos do ciclo de vida de uma consulta, isto é, para cada comando recebido, o SGBDR irá realizar o *parsing*, reescrever o SQL em um formato interno, buscará o plano de consulta mais eficiente e, finalmente, executará a consulta.

Contudo, é bastante comum que a grande maioria das consultas realizadas por um sistema sejam repetitivas, onde apenas parâmetros são fornecidos. Olhando a **Listagem 1** novamente, repare que a parte variável do SQL enviado ao banco de dados é o parâmetro que fornece o nome da pizza. O restante da consulta, incluindo as junções, ordenações e seleções, permanece inalterado entre as invocações. Ou seja, seria preferível realizar os primeiros passos do ciclo de vida da consulta até a geração do plano de execução ótimo, que seria mantido em cache. Para realizar novamente a mesma consulta, basta o cliente informar o novo parâmetro e o SGBDR apenas “insere” o parâmetro no plano em cache e o executa novamente.

Nota:

Para encontrar o plano de execução ótimo, o SGBDR depende da distribuição estatística dos dados das tabelas envolvidas na consulta, da independência estatística entre as colunas das tabelas, bem como de metadados atualizados nas tabelas de sistema do SGBDR. De um modo geral, se as suas tabelas estão normalizadas até a 4ª Forma Normal, o SGBDR será capaz de encontrar um plano de execução quase ótimo, sendo suficiente para a grande maioria dos casos.

Este tipo de cenário é extremamente comum e possui até um nome próprio: Consulta parametrizada. Por ser tão frequente, a maioria dos vendedores de SGBDR preveem uma forma de manter planos de execução em cache – e a maneira do JDBC utilizar esse mecanismo é através da utilização de objetos do tipo **java.sql.PreparedStatement**.

Para melhor ilustrar a utilização de consultas parametrizadas com JDBC, vamos refatorar o código da **Listagem 1** para utilizar o Padrão de Projeto *Command*. Assim, teremos cada consulta em uma classe própria (vide **Listagem 2**).

Note a injeção de dependência na linha 09, onde a conexão é passada como parâmetro do construtor das classes de consulta. Isso permite que esta nova abordagem seja utilizada tanto em

cenários onde conexões são abertas diretamente pelo cliente, muito usada em sistemas desktop, como em sistemas web, onde as conexões são gerenciadas pelo servidor e disponibilizadas para os clientes através de *datasources* (ver **BOX 1**). É importante ressaltar que a injeção de dependência obriga que o método que invocar a consulta abra a conexão e seja responsável pelo seu fechamento ao final.

A **Listagem 3** mostra a implementação da classe **ConsultaRecuperarClientes** utilizando **PreparedStatement**s.

BOX 1. Datasources

DataSource é a forma mais recomendada para adquirir conexões com SGBDRs quando a aplicação é executada em um contêiner Java EE. De forma breve, um datasource é especificado em um descriptor XML, onde as informações para abrir conexões são fornecidas. Quando este arquivo é lido pelo servidor de aplicações, o datasource é armazenado na estrutura JNDI e se torna acessível através de lookups. Por ser um objeto gerenciado pelo contêiner, possui inúmeras vantagens, como utilização de pool de conexões, fechamento automático de conexões abertas “esquecidas” pela aplicação, suporte a transações distribuídas e portabilidade. Para mais informações sobre datasources, veja o endereço indicado na seção [Links](#).

Listagem 2. Utilizando Commands para as consultas.

```
01 public interface Consulta<E>{  
02     List<E> executarConsulta() throws Exception;  
03 }  
  
05 public class ConsultaRecuperarClientes implements Consulta<Cliente>{  
06  
07     private Connection conexao;  
08  
09     public ConsultaRecuperarClientes(Connection conexao, String nomePizza){...}  
10  
11     public List<Cliente> executarConsulta() throws Exception {...}  
12 }
```

Listagem 3. Exemplo de utilização de PreparedStatement.

```
01 public class ConsultaRecuperarClientes implements Consulta<Cliente>{  
02  
03     private Connection conexao;  
04     private String nomePizza;  
05     private String sql = "SELECT * FROM CLIENTE c, PEDIDO p WHERE  
p.telefone = c.telefone AND " + "p.nome_pizza = ? ORDER BY c.nome";  
  
06     public ConsultaRecuperarClientes(Connection conexao, String nomePizza){  
07         this.conexao = conexao;  
08         this.nomePizza = nomePizza;  
09         this.conexao.setReadOnly(true);  
10     }  
  
12     public List<Cliente> executarConsulta() throws Exception{  
13         PreparedStatement pstmt = conexao.prepareStatement(sql);  
14         pstmt.setString(1, nomePizza);  
15         ResultSet rs = pstmt.executeQuery();  
16         List<Cliente> lista = new ArrayList<>();  
17         while(rs.next()){  
18             //cria e popula objetos do tipo Cliente  
19         }  
20         return lista;  
21     }  
22 }
```

Nota:

Uma pergunta pertinente quando utiliza-se JDBC através de datasources é: se o servidor de aplicações gerencia as conexões, como fazer para evitar que os PreparedStatements sejam fechados e, consequentemente, perdidos? A resposta é: praticamente todos os servidores de aplicação fazem cache de PreparedStatements. Assim, quando é solicitado a uma conexão adquirida de um datasource para “preparar” um Statement, é verificado antes se aquele PS já existe em cache. O servidor de aplicações também pode utilizar o número de PS em cache para determinar quais conexões devem ser fechadas primeiro e quais devem ser mantidas no pool de conexões.

No código desta listagem, pode-se ver, na linha 9, que o comando a ser enviado para o SGBDR é apenas para leitura. Desta forma, não serão adquiridos locks/latches no acesso aos dados, evitando que outras consultas fiquem “presas” esperando a finalização do processamento. As linhas 13, 14 e 15 mostram como as consultas parametrizadas funcionam em JDBC: primeiro, prepara-se o SQL que será executado (repare que o parâmetro é representado por um ‘?’ no código enviado ao SGBDR). Nesta etapa, o comando é validado, otimizado e pré-compilado. Por fim, os parâmetros para executar a consulta são informados através dos métodos **setXXX(posição, objeto)**, onde XXX pode ser **String, Int, Float**, etc., dependendo do caso. Por fim, a consulta é executada.

Um efeito colateral positivo da utilização de consultas parametrizadas – e possivelmente uma razão para sua adoção em larga escala – é que elas tornam o código à prova de ataques do tipo *sql injection*. Como os parâmetros agora são apenas inseridos em um plano de execução pré-compilado, se um usuário espertinho tentar inserir trechos de SQL em um campo do formulário não obterá nada além de um resultado vazio.

Aprofundando a interação com o SGBDR através de ResultSets

Um olhar mais atento do leitor que estiver utilizando uma IDE como Eclipse ou NetBeans para reproduzir o código das **Listagem 3** vai reparar que o método **Connection.prepareStatement(sql)** possui duas variantes: **Connection.prepareStatement(String sql, int resultSetType, int resultSetConcurrency)** e **Connection.prepareStatement(String sql, int resultSetType, int resultSetConcurrency, int resultSetHoldability)**. Os parâmetros adicionais neste método são constantes definidas na classe **java.sql.ResultSet** e significam o seguinte:

- Em **resultSetType**, um dos seguintes valores pode ser utilizado:
 - **ResultSet.TYPE_FORWARD_ONLY (valor padrão)**: o cursor somente anda para frente e o objeto **ResultSet** não é sensível a mudanças realizadas por outros clientes após sua criação;
 - **ResultSet.TYPE_SCROLL_INSENSITIVE**: o cursor pode andar para frente ou para trás e o objeto **ResultSet** não é sensível a mudanças após sua criação. Ou seja, se um registro tiver seu valor alterado por outro cliente após a criação do objeto **ResultSet**, este não irá refletir a alteração;
 - **ResultSet.TYPE_SCROLL_SENSITIVE**: o cursor pode andar para frente ou para trás e o objeto **ResultSet** é sensível a mudanças após sua criação. Ou seja, se um registro tiver

seu valor alterado por outro cliente após a criação do objeto **ResultSet**, este irá refletir a alteração.

- Em **resultSetConcurrency**, um dos seguintes valores pode ser utilizado:

- **ResultSet.CONCUR_READ_ONLY (valor padrão)**: cria um objeto **ResultSet** que somente é capaz de ler dados;
- **ResultSet.CONCUR_UPDATABLE**: cria um objeto **ResultSet** atualizável.

- Em **resultSetHoldability**, um dos seguintes valores pode ser utilizado:

- **ResultSet.CLOSE_CURSORS_OVER_COMMIT**: todos os objetos **ResultSet** são fechados quando a transação é confirmada (através da chamada ao método **Connection.commit()**);
- **ResultSet.HOLD_CURSORS_OVER_COMMIT**: os objetos **ResultSet** são mantidos abertos mesmo quando a transação é confirmada.

A **Listagem 4** ilustra uma situação onde estes parâmetros podem ser utilizados para a criação de um **ResultSet** atualizável.

Listagem 4. Criando ResultSets mais avançados.

```
Connection con = getConnection();
PreparedStatement pstmt = con.prepareStatement("SELECT * FROM PIZZA ORDER BY NOME", ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATABLE, ResultSet.HOLD_CURSORS_OVER_COMMIT);
Scanner scanner = new Scanner(System.in);
ResultSet rs = pstmt.executeQuery();
while(rs.next()) {
    System.out.println("Nome da pizza: "+rs.getString("nome"));
    System.out.println("Preço: "+rs.getString("preco"));
    System.out.print("Digite o novo preço:");
    String preco = scanner.nextLine();
    rs.updateString("preco", preco);
    rs.updateRow();
    System.out.println("Preço atualizado com sucesso!");
    con.commit();
}
con.close();
scanner.close();
```

O código deste exemplo exibe, para cada pizza, o seu nome e preço atual. Em seguida, pede para que o usuário digite o novo preço, atualiza o valor dentro do próprio **ResultSet** e confirma a transação. Repare que, após atualizar o valor do campo, o método **ResultSet.updateRow()** deve ser invocado, pois é ele que dispara o comando de atualização para o SGBDR.

Por fim, para que esse código funcione, o **ResultSet** criado deve ser atualizável e não deve ser fechado quando o SGBDR receber o comando para confirmar a transação.

Utilização de SavePoints em transações

Em bancos de dados, as transações têm a função de garantir as propriedades de atomicidade, consistência, isolamento e durabilidade dos acessos, as famosas propriedades ACID. Desta forma, blocos de comandos SQL de um usuário são tratados como uma

única unidade de trabalho, protegida de possível interferência de comandos enviados por outros clientes.

É sabido que para confirmar uma transação e tornar suas modificações duráveis, deve-se utilizar o comando **COMMIT**. Por outro lado, caso algum erro tenha ocorrido durante o processamento, deve-se abortar a transação utilizando o comando **ROLLBACK**. Contudo, existem casos onde pode não ser desejável abortar todas as alterações realizadas pela transação. Considere o seguinte cenário para o sistema de exemplo deste artigo: com o sistema em manutenção, os pedidos passaram a ser anotados em uma planilha Excel (dados dos clientes e as pizzas que eles pediram) para serem lançados posteriormente. Existem duas possíveis inserções neste caso: uma na tabela de clientes, caso o cliente seja novo, e outra na tabela de pedidos, devendo ser realizadas nesta ordem devido à restrição de chave estrangeira.

A rotina para importação destes pedidos realizados offline pode verificar se o cliente é novo através de seu número de telefone. Caso já exista na tabela, o cliente é antigo e nada deve ser feito. Caso contrário, deve-se incluir o novo cliente na tabela antes de realizar a inserção na tabela de pedidos.

Uma vez garantido que o cliente consta na base de dados, é possível inserir o pedido sem problemas, certo? Nem sempre! Como os pedidos foram realizados de maneira *offline*, é possível que o atendente tenha escrito o nome da pizza de forma errada. Seja por erros de ortografia, seja por utilizar uma abreviação para um nome de pizza grande, o fato é que não será possível inserir o pedido deste cliente, uma vez que haverá uma violação de chave estrangeira. Se todos estes passos forem executados dentro de uma única transação, o comando **ROLLBACK** enviado vai desfazer inclusive a inclusão do novo cliente, cujos dados estão corretos. Este é o cenário típico para utilização dos chamados *savepoints*: marcadores dentro de uma transação que permitem desfazer todas as alterações realizadas apenas a partir deles.

A **Listagem 5** mostra a implementação do cenário descrito anteriormente, onde deseja-se importar pedidos que foram anotados de forma offline. O método **realizarVenda()** (linhas 37-52) realiza dois passos distintos: um para cadastrar o cliente (**cadastrarCliente()**, linhas 16-22) e outro para cadastrar o pedido (**cadastrarPedido()**, linhas 24-35). Após realizar a inserção do cliente, um **SAVEPOINT** é criado na linha 21. Deste modo, ao detectar um problema no cadastramento do pedido, um **ROLLBACK** é realizado até o **SAVEPOINT** criado, evitando que a inclusão do cliente seja desfeita. Por fim, o **COMMIT** da linha 48 confirma a inclusão do cliente e do pedido, caso não tenha ocorrido erro na inserção deste último.

Obtendo resultados de consultas baseados em cursores

Quando uma consulta é enviada para ser executada em um SGBDR, a resposta esperada é um conjunto de linhas (também chamadas de *tuplas*). Contudo, podem existir situações onde o resultado é tão grande que pode não caber inteiramente na memória do cliente ou mesmo causar congestionamento na rede. Por exemplo, o SQL executado na **Listagem 1** pode ter um resultado

muito grande para sabores de pizza populares e causar os problemas citados anteriormente: se o resultado possuir 300.000 linhas e cada linha tiver, em média, 1KB, o total de tráfego de rede e aumento de memória da máquina virtual do cliente será de 300MB (**Figura 2**).

Como solução, o ideal é recuperar apenas um conjunto de linhas por vez, evitando as sobrecargas na memória do cliente e no tráfego da rede. Para resolver estes problemas, é possível fazer uso de *cursores* dos SGBDRs em conjunto com o comando *fetch*. A **Figura 3** ilustra como o sistema se comportaria recuperando 100 linhas por vez. Neste caso, o resultado seria fatiado em pequenas quantias com o tamanho de 100KB por vez, algo perfeitamente aceitável. Neste cenário, o bloco de linhas retornado é armazenado em cache do lado do cliente e, quando o método **ResultSet.next()** tentar acessar a primeira tupla que ainda se encontra no servidor, um novo bloco de 100 linhas é enviado para o cliente.

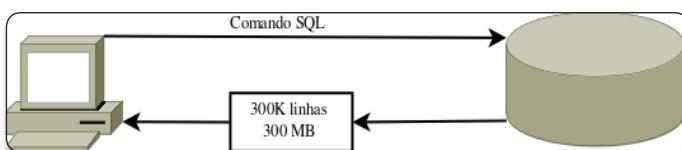


Figura 2. Sem cursores

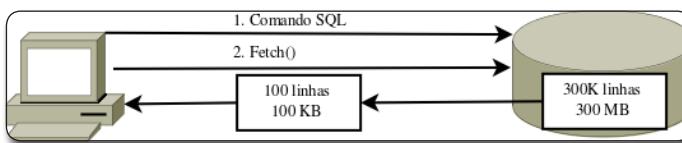


Figura 3. Com cursores

Vale ressaltar que existem algumas restrições para a utilização de cursores. Por exemplo, no PostgreSQL, que é o SGBDR adotado nos exemplos deste artigo, as seguintes condições são necessárias:

- A conexão com o servidor deve ser realizada utilizando o protocolo V3, que é o protocolo padrão de conexão;
- A conexão **não** pode estar no modo *autocommit*, pois o servidor fecha os cursores quando a transação é confirmada;
- O **ResultSet** deve ser do tipo **ResultSet.TYPE_FORWARD_ONLY**, que é o comportamento padrão. Em outras palavras, não é possível “retroceder”;
- O comando enviado ao servidor deve conter uma única consulta SQL.

A **Listagem 6** exibe como utilizar cursores para receber o resultado da consulta enviada ao servidor.

Repare que, para o cliente, a utilização ou não de cursores é transparente: quando o método **rs.next()** (linha 18) tentar acessar a primeira linha que ainda não foi recuperada, um novo bloco do resultado é solicitado ao servidor até que todas as tuplas tenham sido consumidas.

LIMIT vs. FETCH

É comum haver confusão entre as semânticas da cláusula **LIMIT < TAMP >**, utilizada para limitar o número total de linhas do resultado, e do comando **FETCH**. Na verdade, são conceitos bem distintos. Enquanto **LIMIT < TAMP >** define o número de linhas total do resultado da consulta (ex.: os 10 maiores salários),

Listagem 5. Exemplo de utilização de SavePoints.

```

01 public class RealizaVendas{
02
03     private String sqlInsereCliente = "insert into cliente
04         (telefone, nome, endereco) "+ "values(?, ?, ?)";
05     private String sqlInserePedido = "insert into pedido
06         (telefone, data_hora, nome_pizza,"+ "quantidade) values(?, ?, ?, ?)";
07     private PreparedStatement pstmtCliente;
08     private PreparedStatement pstmtPedido;
09     private Connection conexao;
10
11     public RealizaVendas(Connection conexao) throws Exception{
12         this.conexao = conexao;
13         pstmtCliente = conexao.prepareStatement(sqlInsereCliente);
14         pstmtPedido = conexao.prepareStatement(sqlInserePedido);
15     }
16
17     private void cadastrarCliente(Cliente cliente){
18         pstmtCliente.setString(1, cliente.getTelefone());
19         pstmtCliente.setString(2, cliente.getNome());
20         pstmtCliente.setString(3, cliente.getEndereco());
21         pstmtCliente.executeUpdate();
22         cadastroClienteSP = conexao.setSavepoint();
23     }
24
25     private void cadastrarPedido(Pedido pedido) throws SQLException{
26         try {
27             pstmtPedido.setString(1, pedido.getTelefone());
28             pstmtPedido.setDate(2, new java.sql.Date(pedido.getDataHora())
    
```

```

        .getTime()));
29             pstmtPedido.setString(3, pedido.getNomePizza());
30             pstmtPedido.setInt(4, pedido.getQuantidade());
31             pstmtPedido.executeUpdate();
32         } catch (SQLException e) {
33             e.printStackTrace();
34             conexao.rollback(cadastroClienteSP);
35             //desfaz somente a tentativa de inserir novo pedido
36         }
37
38         public void realizarVenda(Cliente cliente, Pizza pizza, Integer quantidade){
39             cadastrarCliente(cliente);
40             Pedido pedido = new Pedido(cliente.getTelefone(), new Date(),
41                 pizza.getNome(), quantidade);
42             try {
43                 cadastrarPedido(pedido);
44             } catch (SQLException e1) {
45                 e1.printStackTrace();
46             }
47             try {
48                 conexao.commit();
49             } catch (SQLException e) {
50                 e.printStackTrace();
51             }
52         }
53     }
    
```

o comando **FETCH** define o número de linhas a ser buscado por vez do resultado total (ex.: todos os salários, recuperando 10 por vez). Repare na **Figura 4** que inclusive o SGBDR pode gerar planos de execução diferentes para consultas com e sem **LIMIT**.

LIMIT...OFFSET ou FETCH para realizar paginação?

Esta é outra dúvida que surge quando o desenvolvedor deseja paginar o resultado de uma consulta. Suponha que o sistema exiba 10 pedidos por vez na sua página. Em situações como esta, deve-se utilizar **LIMIT...OFFSET** ou cursores para realizar a paginação?

Listagem 6. Exemplo de resultado baseado em cursores.

```
01 public class ConsultaPedidosRecentes implements Consulta<Pedido> {
02     private String sql = "select * from pedido";
03     private static final int FETCH_SIZE = 10;
04     private Connection conexao;
05
06     public ConsultaPedidosRecentes(Connection conexao){
07         this.conexao = conexao;
08         this.conexao.setAutoCommit(false);
09     }
10
11    public List<Pedido> executarConsulta() {
12        try {
13            PreparedStatement stmt = conexao.prepareStatement(sql);
14            stmt.setFetchSize(FETCH_SIZE); //indica o número de linhas a ser
15            recuperado por vez.
16            ResultSet rs = stmt.executeQuery();
17            List<Pedido> lista = new ArrayList<>();
18            while(rs.next()){
19                String telefone = rs.getString("telefone");
20                Date dataHora = rs.getTimestamp("data_hora");
21                String nomePizza = rs.getString("nome_pizza");
22                Integer quantidade = rs.getInt("quantidade");
23                Pedido pedido = new Pedido(telefone, dataHora, nomePizza, quantidade);
24                lista.add(pedido);
25            }
26        } catch (SQLException e) {
27            System.err.println("Erro ao recuperar os pedidos do banco de dados.");
28            throw new RuntimeException(e);
29        }
30    }
```

Ambas funcionam, mas possuem uma diferença importante: a utilização de **LIMIT...OFFSET** implicará na mesma consulta sendo executada várias vezes, com apenas um subconjunto do resultado sendo retornado ao cliente. Já o **FETCH** executa a consulta uma única vez (o que pode aumentar consideravelmente o tempo de processamento), retornando subconjuntos do resultado. Neste caso, para melhorar o desempenho, uma combinação dos dois pode ser utilizada.

Importação de grande volume de dados

Apesar de ser uma tarefa comum durante a construção de Data Warehouses, de vez em quando é necessário criar rotinas para importação de dados para um SGBDR transacional (ver **BOX 2**). Seja para unificar bases de dados, seja para importar dados que foram coletados de forma offline durante uma queda do sistema, o fato é que essa tarefa não chega a representar um problema quando a quantidade de dados é pequena.

Conforme o tamanho dos dados aumenta, problemas de desempenho na importação começam a aparecer e podem até inviabilizar a operação. Para ilustrar essa situação, considere a base de dados utilizada como exemplo neste artigo: apesar de poder ser considerada pequena, contendo 3,8MB e 87772 linhas, somadas as três tabelas e excluindo índices e outras estruturas auxiliares, a forma como o carregamento dos dados é realizado pode influenciar muito no tempo total de processamento. A **Listagem 7** mostra uma implementação considerada ingênuo para importação destes dados.

Esse código exibe somente a importação dos clientes (os dados relativos às pizzas e aos pedidos são inseridos de maneira análoga). Qual o problema nesta abordagem? Para saber essa resposta, deve-se ter em mente o que acontece quando uma única linha é inserida, ou seja, o que é realizado pelo SGBDR para cada invocação do método **Statement.executeUpdate(...)**:

1. Uma transação é aberta;
2. Checagens de restrição são executadas para garantir que não há nenhuma violação de integridade;
3. Os novos dados são escritos no log de transação;

```
jdbc_alem_basico=> explain select * from pedido p, cliente c where c.telefone = p.telefone and lower(p.nome_pizza) = 'calabresa' LIMIT 10;
QUERY PLAN
-----
Limit  (cost=0.28..112.38 rows=10 width=135)
 -> Nested Loop  (cost=0.28..4641.18 rows=414 width=135)
      -> Seq Scan on pedido p  (cost=0.00..3399.80 rows=414 width=33)
          Filter: (lower((nome_pizza)::text) = 'calabresa'::text)
      -> Index Scan using cliente_pkey on cliente c  (cost=0.28..2.99 rows=1 width=102)
          Index Cond: ((telefone)::text = (p.telefone)::text)
(6 rows)

jdbc_alem_basico=> explain select * from pedido p, cliente c where c.telefone = p.telefone and lower(p.nome_pizza) = 'calabresa';
QUERY PLAN
-----
Hash Join  (cost=363.50..3771.06 rows=414 width=135)
 Hash Cond: ((p.telefone)::text = (c.telefone)::text)
 -> Seq Scan on pedido p  (cost=0.00..3399.80 rows=414 width=33)
     Filter: (lower((nome_pizza)::text) = 'calabresa'::text)
 -> Hash  (cost=301.00..301.00 rows=5000 width=102)
     -> Seq Scan on cliente c  (cost=0.00..301.00 rows=5000 width=102)
(6 rows)
```

Figura 4. Planos de execução com e sem a cláusula **LIMIT**

4. A transação é confirmada, disparando os seguintes processos:
- 4.1. Os dados são escritos no disco;
 - 4.2. Todos os índices que contêm alguma das colunas da tabela onde os dados foram inseridos são atualizados.

BOX 2. ETL - Extração, Transformação e Carga

Um projeto de criação de Data Warehouse (que é, de forma simples, um banco de dados voltado para a geração de relatórios para a alta diretoria de uma empresa) possui requisitos e etapas diferentes daqueles apresentados para a construção de um sistema transacional. Uma destas etapas é a ETL, que consiste em extrair dados de várias fontes, entre elas bancos de dados relacionais, planilhas Excel e arquivos-texto; transformar, limpar e qualificar estes dados de forma a eliminar redundâncias, discrepâncias, etc.; e carregar estes dados no banco de dados final, que será acessado pelo sistema de geração de relatórios. Esta etapa é tão importante que chega a consumir cerca de 70% do tempo de geração de um DW e existem ferramentas construídas especificamente para este propósito. Ao realizar a carga de dados para o banco de dados de destino, as ferramentas ETL são capazes de utilizar opções avançadas dos SGBDs, acelerando muito esta etapa quando comparada às soluções programáticas vistas aqui. A dica, então, é: caso possua uma quantidade muito, muito grande de dados para carregar, deve-se considerar a utilização de alguma ferramenta ETL para esta tarefa. Quando possível, recorra ao DBA da sua equipe. Uma sugestão de ferramenta ETL gratuita é o Pentaho Kettle (veja a seção [Links](#)).

Listagem 7. Importação de dados – forma 1.

```
Connection conexao = getConexao();
Statement stmt = conexao.createStatement();
while((linha = arquivoClientes.readLine())!= null){
    String [] tokens = linha.split("\t");
    String sql = "insert into cliente(telefone, nome, endereco)
    values(\""+tokens[0]+"\",""+tokens[1]+"\",""+tokens[2]+"\")"; stmt.executeUpdate(sql);
}
arquivoClientes.close();
conexao.close();
```

Desta forma, mesmo a importação de uma quantidade pequena de dados, como a contida no exemplo, pode levar muito tempo para terminar. Em um experimento rápido, onde tanto o programa quanto o SGBDR rodavam na mesma máquina, esta importação demorou 219452ms ou 3,6 minutos. Ou seja, cada byte leva, aproximadamente, 0,05 ms para ser inserido. Se fosse uma massa de dados maior contendo, por exemplo, 1GB, seria necessário esperar 62421902,2 ms ou 17 horas!

Uma maneira de melhorar o desempenho nesta tarefa seria retirar a opção de *autocommit* da conexão aberta. Desta forma, os passos 1, 4, 4.1 e 4.2 seriam realizados somente uma vez para a operação inteira. O código para esta implementação é apresentado na [Listagem 8](#).

Esta simples alteração permite um ganho de tempo enorme. Nos testes realizados, a importação levou 31727 ms, ou seja, aproximadamente 0,5 minuto. Cada byte leva, neste caso, 0,008 ms para ser inserido. No caso de uma massa de dados de 1GB, o tempo total de espera seria de 9024568,8 ms ou 2,5 horas. Contudo, esta projeção não é totalmente precisa, uma vez que a importação ainda ocorre dentro de uma transação e o SGBDR pode decidir, a

qualquer momento, escrever os dados no disco (passo 4.1) e, caso qualquer erro ocorra durante a importação, todas as operações de escrita serão desfeitas (ROLLBACK), gerando um novo e inesperado *overhead*. Para piorar, cada comando de inserção está sendo enviado separadamente, o que significa uma sobrecarga na rede no momento da transferência dos dados. Sendo assim, seria mais interessante se os comandos fossem enviados em blocos (*batch*), conforme a [Listagem 9](#).

Listagem 8. Importação de dados sem autocommit.

```
Connection conexao = getConexao();
conexao.setAutoCommit(false);
Statement stmt = conexao.createStatement();
while((linha = arquivoClientes.readLine())!= null){
    String [] tokens = linha.split("\t");
    String sql = "insert into cliente(telefone, nome, endereco)
    values(\""+tokens[0]+"\",""+tokens[1]+"\",""+tokens[2]+"\")"; stmt.executeUpdate(sql);
}
arquivoClientes.close();
conexao.commit();
conexao.close();
```

Listagem 9. Importação de dados sem autocommit.

```
Connection conexao = getConexao();
Statement stmt = conexao.createStatement();
while((linha = arquivoClientes.readLine())!= null){
    String [] tokens = linha.split("\t");
    String sql = "insert into cliente(telefone, nome, endereco)
    values(\""+tokens[0]+"\",""+tokens[1]+"\",""+tokens[2]+"\")"; stmt.addBatch(sql);
}
arquivoClientes.close();
stmt.executeBatch();
conexao.close();
```

Nesse código cada comando de inserção é adicionado ao bloco que será executado no servidor através do método **Statement.addBatch(sql)**. Ao chamar o método **Statement.executeBatch()**, os comandos são enviados de uma única vez para o servidor e lá são executados. O retorno deste método é um array de inteiros, onde cada posição representa o número de linhas afetadas pelo respectivo comando. Nos testes realizados, esta estratégia levou 14103 ms ou seja, pouco menos de um quarto de minuto. Assim, cada byte demorou 0,003 ms para ser inserido. Um desempenho que levaria 4011520 ms (pouco mais de 1 hora) para inserir 1GB de dados. Repare a diferença entre os tempos: 17 horas, 2,5 horas, 1 hora. Definitivamente, a escolha do método para realizar essa importação de dados realmente faz diferença.

É muito comum encontrarmos no mercado desenvolvedores que possuem um conhecimento bem avançado de uma tecnologia mas não dominam a base dela. Dentre os exemplos no mundo Java deste fenômeno estão aqueles que sabem muito sobre a Java Persistence API (JPA) mas pouco entendem sobre como a JPA acessa de fato os bancos de dados, ou seja, como utiliza a Java Database Connectivity API, ou, como é chamada, JDBC.

JDBC além do básico

Contudo, existem tarefas onde a utilização de JPA não é recomendada, muitas vezes apresentando um desempenho bem ruim. Assim, ter conhecimentos mais aprofundados de JDBC para realizar essas tarefas é fundamental para um bom desenvolvedor. Para isso, é necessário tanto reconhecer e distinguir os passos realizados pelo SGBDR ao processar uma consulta, como ter um bom entendimento de como acontece a interação programa-SGBDR.

Autor



Luís Fernando Orleans

[lforleans@ieee.org, ufrrj.br] - http://lforleans.blogspot.com.br/
É Professor Adjunto do curso de Ciências da Computação da Universidade Federal Rural do Rio de Janeiro (UFRJ) e obteve o grau de Doutor em Engenharia de Sistemas e Computação em 2013. Desenvolve e coordena diversos projetos em Java desde 2001, tendo acompanhado de perto todas as novidades da linguagem desde então. Apaixonado pela família, por tecnologia, café, vinho, Flamengo e bons papos.



Links:

Architecture of a Database System.

<http://db.cs.berkeley.edu/papers/fntdb07-architecture.pdf>

Livro: Padrões de Projeto: Soluções reutilizáveis de software orientado a objetos, Editora Bookman, 2000.

Datasources.

<http://docs.oracle.com/javase/tutorial/jdbc/basics/sqldatasources.html>

Pentaho Kettle.

<http://community.pentaho.com/>

Tutorial de JDBC da Oracle.

<http://docs.oracle.com/javase/tutorial/jdbc/>

FÓRUM DEVMEDIA

O lugar perfeito para você ficar por dentro de tudo o que acontece nas tecnologias do mercado atual

No fórum da DevMedia você irá encontrar uma equipe disponível e altamente qualificada com consultores e colaboradores prontos para te ajudar a qualquer hora e sobre qualquer assunto. Temos as salas de Java, .NET, Delphi, Banco de Dados, Engenharia de Software, PHP, Java Script, Web Design, Automação comercial, Ruby on Rails e muito mais!



ACESSE AGORA
www.devmedia.com.br/forum

Programando com estruturas de dados e padrões de projeto

Aprenda a usar o padrão Iterator para implementar a estrutura de dados de lista duplamente encadeada

Uma das principais atividades da computação é o processamento de dados. Esta atividade exige técnicas eficientes para organizar e acessar esses dados na memória das máquinas, de maneira que operações como inclusão, pesquisa e remoção de dados sejam facilitadas de acordo com as necessidades da aplicação. Para resolver esta categoria de problemas existem muitas soluções, chamadas de estruturas de dados. Dentre estas soluções, uma das mais utilizadas, por ser bastante adaptável e simples, é a lista encadeada. Em uma lista, os dados podem ser inseridos e removidos em qualquer ordem e em qualquer posição, diferentemente de outras estruturas, como o caso das filas e das pilhas, nas quais a ordem de inserção e remoção é determinada pelo algoritmo e não pela aplicação que o está utilizando.

A maioria das estruturas de dados podem ser implementadas utilizando-se vetores, porém isso impõe uma limitação de tamanho e é conveniente apenas quando se conhece antecipadamente a quantidade máxima de elementos que serão armazenados na mesma, como acontece nos *buffers* utilizados em processos de comunicação. Para os casos em que não se conhece a quantidade de elementos que serão inseridos, é indicada a utilização de uma lista encadeada, cujo princípio de funcionamento é a criação de ligações ou apontadores entre os seus elementos. Estes apontadores guardam a sequência dos elementos na lista e são utilizados para viabilizar a navegação e acesso a cada um dos elementos presentes.

As ligações entre os elementos da lista formam na memória uma estrutura como a apresentada na **Figura 1**. Quando as ligações são estabelecidas em apenas um

Fique por dentro

Este artigo apresenta uma explicação do funcionamento de uma das estruturas de dados mais utilizadas no desenvolvimento de aplicações, a lista encadeada. Esta análise será feita de forma crescente, construindo ao final uma solução mais sofisticada e que conte com as características descritas no Padrão Iterator, que é utilizado para permitir o acesso aos dados sem que seja necessário expor características de implementação da estrutura de dados utilizada. Para isso, iniciaremos com o desenvolvimento de uma versão simples, porém funcional, da lista encadeada, a partir da qual pequenos incrementos serão realizados até que se obtenha o resultado desejado.

sentido, a lista é chamada de simplesmente encadeada, e quando as ligações são estabelecidas nos dois sentidos, conforme a **Figura 2**, diz-se que a lista é duplamente encadeada. O exemplo construído ao longo deste artigo é o de uma lista duplamente encadeada.

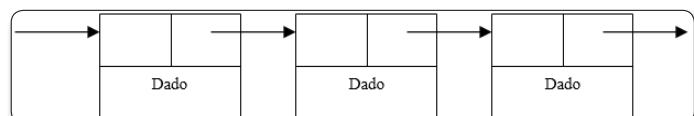


Figura 1. Representação das ligações entre os elementos de uma lista encadeada

Para implementar estas ligações se faz necessária a declaração de uma classe que contenha referências a objetos da própria classe. Estas auto-referências representam as conexões entre os elementos mostradas na **Figura 1**. A classe que representa os elementos de dados como ilustrado nas caixas da mesma figura, também chamados de nós da lista, deve guardar os dados referentes à correspondente posição da lista, e para não ser necessário

alterar o código dela a cada novo tipo de dado a ser armazenado, recomenda-se adotar um tipo de referência que possa ser utilizada com qualquer tipo de dado que por ventura venha a ser armazenado na lista. Deste modo, é declarado um atributo do tipo **Object**, que é a superclasse para todas as classes definidas em Java, permitindo com isso que qualquer objeto seja tratado como sendo do tipo **Object** (uma alternativa ao uso do atributo deste tipo é a parametrização de classes, o que será assunto em um próximo artigo). A classe que representa o nó de uma lista duplamente encadeada é apresentada na **Listagem 1**.

Listagem 1. Código da classe No.

```
class No{  
    public No proximo;  
    public No anterior;  
    public Object dado;  
  
    public No(Object obj){  
        proximo = null;  
        anterior=null;  
        dado=obj;  
    }  
  
    No(Object obj, No prox, No ant)  
    {  
        proximo = prox;  
        anterior = ant;  
        dado = obj;  
    }  
}
```

Para a classe **No** mostrada na listagem foram definidos dois construtores: um para o caso no qual ainda não se sabe quais são os elementos aos quais o novo nó da lista estará ligado, como no caso de se estar instanciando o primeiro nó da lista (portanto os atributos que representam as conexões ao **próximo** nó e ao **nó anterior** são inicializados como **null**, indicando que este elemento não está ligado a outros nós da lista); e outro construtor para o caso em que já se sabe a quais elementos o novo nó estará ligado, como nos casos de elementos a serem inseridos no meio da lista (portanto, as ligações com os seus vizinhos são inicializadas com as respectivas referências passadas como parâmetros ao construtor e armazenadas nos atributos correspondentes). Em ambos os casos, a referência ao objeto que representa o dado a ser armazenado no nó é recebida como parâmetro.

A lista encadeada é definida não apenas pelos nós que representam os seus elementos, mas também pelas operações que são realizadas nestes elementos/nós. Portanto, são necessários métodos para inserir, consultar e remover elementos da lista encadeada.

Para representar a lista também se faz necessária a declaração de uma classe, na qual estes métodos serão implementados. Além disso, a implementação da classe também precisa conhecer os seus nós **inicial** e **final**. Resumindo, uma lista duplamente encadeada é constituída de elementos ou nós, conectados entre si em uma determinada sequência (definida pela aplicação que faz uso da lista) e também de duas referências, que indicam dentre os elementos armazenados na lista qual é o primeiro e qual é o último, como está representado na **Figura 2**.

A classe que representa a lista duplamente encadeada é apresentada na **Listagem 2**, porém, sem as implementações correspondentes aos métodos de inserção, remoção e busca, que serão detalhados mais adiante. Nesta classe são declarados também os atributos descritos anteriormente, do tipo **No**, que servem para representar o primeiro e o último elemento armazenados na lista. Também é declarado um valor inteiro como atributo, para armazenar o tamanho atual da lista.

Deste modo, ao criar uma lista, com a instanciação de um objeto do tipo **ListaDupEncadeada**, a mesma se encontrará vazia, ou seja, as referências ao primeiro e último elementos serão nulas e seu tamanho será zero, conforme a inicialização realizada no construtor da classe. Nesta classe também está implementado um método de acesso para o valor do tamanho da lista, que é atualizado nos métodos de inserção e remoção.

Continuando com as funcionalidades disponíveis na classe **ListaDupEncadeada**, tem-se a declaração de um método de inserção, que retorna o novo tamanho da lista como resultado. Este valor pode ser utilizado para verificar se o elemento foi inserido corretamente, comparando-o com o tamanho da lista antes da solicitação de inserção. Já o método de remoção retorna o dado correspondente à posição da lista que foi removida, permitindo ao objeto que manipula a lista realizar qualquer cálculo necessário com o mesmo. No caso do método de consulta, uma referência para o objeto que representa o dado, ou informação, associada à posição solicitada é fornecida como resposta, mas nenhuma alteração é realizada na lista ou em seus nós.

Inserção e remoção da lista

Para completar as funcionalidades da classe apresentada na **Listagem 2**, os métodos de inserção e remoção para listas podem ser implementados para permitir executar estas operações no **início**, no **final** ou no **meio** da lista, o que é possível realizar com diferentes métodos, um para cada caso, ou também é possível sobrepor estes métodos, informando qual das três situações deve ser executada pela diferenciação dos parâmetros. Outra situação bastante comum é a possibilidade de o método de inserção

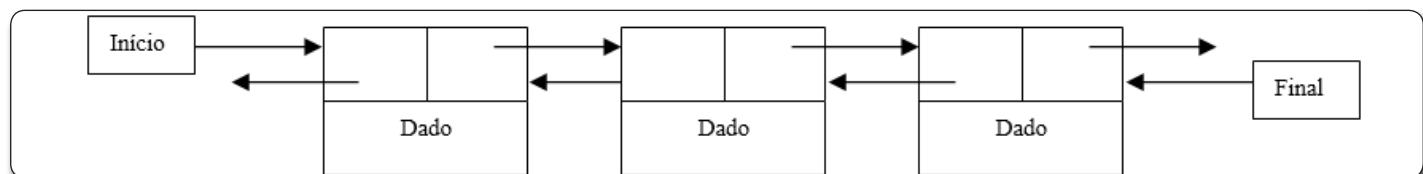


Figura 2. Representação de uma lista duplamente encadeada

buscar na lista o local correto do elemento, por exemplo, no caso de se desejar criar uma lista ordenada.

Listagem 2. Código de uma classe para uma lista duplamente encadeada.

```
public class ListaDupEncadeada {  
    No inicio;  
    No fim;  
    int tamanho;  
  
    public ListaDupEncadeada() {  
        inicio = null;  
        fim = null;  
        tamanho = 0;  
    }  
  
    public int getTamanho(){  
        return tamanho;  
    }  
  
    public int insere(Object obj, int pos)  
    {  
        ...  
        tamanho++;  
        return tamanho;  
    }  
  
    public Object remove(int pos)  
    {  
        Object obj = null;  
        ...  
        tamanho--;  
        return obj;  
    }  
  
    public Object consulta(int pos)  
    {  
        No aux = inicio;  
        ...  
        return aux.dado;  
    }  
}
```

De uma maneira geral, é possível realizar as operações de inserção e remoção informando-se em qual posição da lista o novo elemento deve ser inserido, ou qual a posição que deve ser removida. O método de inserção que recebe a posição do novo elemento é apresentado na **Listagem 3** e o método que remove uma determinada posição da lista é implementado na **Listagem 4**.

Para realizar a operação de inserção, em primeiro lugar é necessário verificar se a posição de inserção está no intervalo válido, ou seja, entre 1 e o tamanho da lista mais 1; na sequência, verificar também se já existe algum elemento na lista ou se a mesma ainda está vazia, e em caso positivo o novo elemento passa a ser tanto o início quanto o final da lista. Caso já exista pelo menos um elemento na lista, então a operação de inserção é realizada de acordo com a posição passada como parâmetro. Se for solicitado que o novo elemento ocupe a posição de índice 1, ele será colocado no início da lista, atualizando-se em seguida as referências do atributo **ListaDupEncadeada.inicio**, para que este agora indique o novo elemento, e o nó que antes estava como início da lista passa então a ser o segundo.

Seguindo a mesma lógica, se for solicitado que o novo elemento ocupe a posição de índice igual ao tamanho da lista mais 1, ele será colocado no final, atualizando-se então as referências do atributo **ListaDupEncadeada.fim**, e da mesma forma que no início, o novo elemento passa a ser o último da lista e o que antes era o último passa a ser o penúltimo.

Como último caso, se a posição de inserção do novo elemento estiver entre 1 e o tamanho da lista, então é necessário percorrer as conexões entre os elementos até que se encontre a posição indicada e então inserir o novo elemento entre dois nós. Para isto é necessário alterar as ligações de **No.proximo** e **No.anterior** destes nós e informar quais são estes nós para o novo elemento da lista, conforme o código da **Listagem 3**.

Listagem 3. Código do método de inserção.

```
public int insere(Object obj, int pos){  
    No no;  
  
    if(obj == null)  
        return tamanho;  
  
    if(pos < 1 || pos > tamanho + 1) // posição solicitada fora do intervalo  
        return tamanho;  
  
    if(inicio == null)//primeiro elemento - lista vazia - ou (tamanho == 0)  
    {  
        no = new No(obj);  
        inicio = fim = no;  
    }  
    else // já existem elementos na lista  
    {  
        if(pos == 1)//inserir no inicio da lista  
        {  
            no = new No(obj, inicio, null);  
            inicio.anterior = no;  
            inicio = no;  
        }  
        else if(pos == tamanho + 1)//inserir no final da lista  
        {  
            no = new No(obj, null, fim);  
            fim.proximo = no;  
            fim = no;  
        }  
        else// inserir no meio da lista  
        {  
            No aux = inicio;  
            while(pos > 1)  
            {  
                aux = aux.proximo;  
                pos--;  
            }  
            // inserir na posição de aux  
            no = new No(obj, aux, aux.anterior);  
            aux.anterior.proximo = no;  
            aux.anterior = no;  
        }  
    }  
    tamanho++;  
    return tamanho;  
}
```

A remoção de elementos da lista também precisa passar por algumas verificações. A primeira delas é para saber se existe algum elemento ou se a lista está vazia e se a posição a ser removida está no intervalo válido, ou seja, entre 1 e o tamanho da lista. Em seguida é necessário saber se existe apenas um elemento, caso especial no qual depois da remoção a lista passar a estar vazia, e tanto a referência do início quanto a do final devem ser alteradas para `null`, indicando que não há nós na lista. Como no caso da inserção, um índice indicando qual é a posição a ser removida é passado como parâmetro. A partir disso, três casos podem ocorrer: remover o primeiro elemento; remover o último elemento; ou remover um elemento do meio da lista. Nos dois primeiros casos o procedimento é similar, apenas com o cuidado de se manipular a referência do início ou do final da lista. Já para o caso de se desejar remover um nó do meio da lista, primeiro é necessário encontrá-lo na sequência de encadeamentos, contando-se o número de nós visitados até chegar ao índice solicitado, para então realizar a remoção.

Em todos os casos, uma referência para o objeto que representa o dado armazenado na lista é criada, e ao final do processo de remoção do nó, esta referência é retornada como resultado, indicando que o elemento foi corretamente removido da lista. No caso de a solicitação não poder ser realizada, uma referência nula é retornada, indicando que nenhum objeto foi removido (o tratamento destas condições de erro, tanto na inserção quanto na remoção, pode ser aprimorado utilizando-se o lançamento de exceções, o que será visto em um próximo artigo). Na **Listagem 4** é apresentado o código correspondente à remoção de um elemento da lista encadeada.

Consultando a lista

Com os métodos de inserção e remoção já é possível trabalhar com a lista, porém é necessário também poder consultar o seu conteúdo, isto é, deve ser possível solicitar um objeto armazenado na lista a partir de algum parâmetro de busca, e o procedimento de consulta, ou busca, deste valor é realizado de forma similar aos de inserção e remoção, com a ressalva de que a ordem dos nós no encadeamento da lista não é afetada. Na **Listagem 5** é apresentado o método de consulta baseado em um índice de um elemento da lista e, como pode ser verificado, novamente os testes para as condições de lista vazia e índice fora do intervalo válido são realizados para determinar se a consulta é válida, e caso não seja um índice válido, ou seja, a solicitação de busca pede uma posição fora da lista, ou ainda se a lista estiver vazia, uma referência nula é retornada (esta também é uma situação que pode ser tratada com o lançamento de uma exceção).

Para os casos de índices válidos, é necessário navegar pelos encadeamentos, a partir do início ou do final da lista, até a posição desejada, para então retornar a referência do objeto armazenado no nó solicitado. Para os casos especiais de se desejar consultar o primeiro ou o último elemento da lista, basta acessar a referência do início ou do final da lista, conforme o caso, para retornar o dado correto.

Listagem 4. Código do método de remoção.

```
public Object remove(int pos)
{
    Object obj;
    if(inicio == null)//se a lista está vazia
        return null;
    if(pos < 1 || pos > tamanho) // posição solicitada fora do intervalo
        return null;
    if(inicio == fim)//se existe apenas um elemento na lista - ou (tamanho == 1)
    {
        obj = inicio.dado;
        inicio = fim = null;
        tamanho--;
        return obj;
    }
    if(pos == 1)//remover o primeiro elemento da lista
    {
        obj = inicio.dado;
        inicio = inicio.prox;
    }
    else if(pos == tamanho)//remover o último elemento da lista
    {
        obj = fim.dado;
        fim = fim.ant;
    }
    else //remover um elemento no meio da lista
    {
        No aux = inicio;
        while(pos > 1){
            aux = aux.prox;
            pos--;
        }
        //remover o elemento aux
        obj = aux.dado;
        aux.ant.prox = aux.prox;
        aux.prox.ant = aux.ant;
    }
    tamanho--;
    return obj;
}
```

O código apresentado na **Listagem 5** sempre realiza a consulta a partir do início da lista, o que pode ser aprimorado verificando-se antes de iniciar a navegação se a posição solicitada está na primeira ou na segunda metade da lista, para então iniciar a navegação a partir do início ou do final, considerando o que estiver mais próximo do elemento solicitado.

É importante perceber que, ao realizar a navegação por todos os elementos da lista, este método de busca irá visitar cada nó diversas vezes até que todos os nós da lista sejam acessados. Para entender melhor esta situação, considere uma lista cujo conteúdo deva ser exibido ao usuário. Para tanto, todos os nós da lista devem ser consultados. Com um laço controlado por um contador é fácil realizar esta tarefa, bastando chamar o método `ListaDupEncadeada.consulta()` com os índices sequenciais de cada nó da lista para obter os dados neles armazenados. Assim, ao consultar o primeiro nó da lista, o seu conteúdo é retornado de imediato, já na segunda iteração do laço, o método `ListaDupEncadeada.consulta()` é chamado com o parâmetro “2”, e conforme o código apresentado na **Listagem 5**, o primeiro e o segundo nós são visitados para que o conteúdo do segundo elemento seja retornado. Na terceira iteração do laço, a nova

chamada do método de consulta recebe o parâmetro “3”, o que faz com que o método visite os nós “1, 2 e 3”, para só então retornar o seu conteúdo. Este comportamento existe devido à definição da estrutura da lista encadeada.

Listagem 5. Código do método de consulta.

```
public Object consulta(int pos)
{
    if(inicio == null)//se a lista está vazia
        return null;
    if(pos < 1 || pos > tamanho) // posição solicitada fora do intervalo
        return null;
    if(pos==1)//consulta o primeiro elemento da lista
        return inicio.dado;
    else if(pos == tamanho)//consulta o último elemento da lista
        return fim.dado;
    else //consulta um elemento no meio da lista
    {
        No aux = inicio;
        while(pos > 1){
            aux = aux.proximo;
            pos--;
        }
        // consulta o elemento aux
        return aux.dado;
    }
}
```

Listagem 6. Casos especiais de inserção e remoção.

```
public int inserelnicio(Object obj){
    return insere(obj, 1);
}
public int insereFim(Object obj){
    return insere(obj, tamanho + 1);
}

public Object removelnicio()
{
}
public Object removeFim()
{
    return remove(tamanho);
}
```

A cada chamada a estes métodos o tamanho da lista é retornado, o que pode ser utilizado para verificar se a inserção realmente ocorreu. No exemplo, o tamanho da lista é mostrado no console juntamente com a **String** que foi adicionada.

Uma vez inseridos os dados, o conteúdo da lista pode ser pesquisado, o que é feito utilizando-se um laço com um contador variando de 1 até o tamanho da lista. Neste laço, cada um dos elementos consultados é mostrado no console juntamente com o seu valor de índice, que é a variável de controle do laço. O valor de cada elemento é obtido da lista com uma chamada ao método **Lista.consulta()**.

Depois de mostrar o conteúdo, os nós da lista são removidos chamando-se os métodos **Lista.remove()**, **Lista.removeInício()** e **Lista.removeFim()**. Estes métodos retornam o dado armazenado na lista que está sendo removido, o qual é mostrado no console seguido do tamanho da lista após a remoção do elemento solicitado. As saídas produzidas por este teste são apresentadas na **Listagem 8**.

Como pode ser verificado, nos casos de consulta e inserção ou remoção de elementos que estão no meio da lista, os métodos implementados necessitam percorrer todo o encadeamento de nós até encontrar a posição desejada. Para listas de pequeno porte isso não é um problema, mas se uma lista com muitos elementos estiver sendo utilizada por uma aplicação que, por exemplo, necessita realizar a ordenação dos dados, então o fato de se percorrer a lista elemento a elemento para cada operação de consulta, remoção ou inserção torna o uso desta implementação muito custoso.

Este problema seria solucionado facilmente se a classe que está utilizando a lista tivesse uma referência direta ao nó da lista com o qual está interagindo. Mas isso fere a ideia de encapsulamento, da orientação a objetos, ou seja, que os atributos de uma classe devem ser conhecidos e manipulados apenas pela própria classe. Neste caso a ideia é que apenas a classe que implementa a lista conheça e acesse os seus dados. Permitir que outras classes façam isso pode levar a situações em que as referências entre os nós da lista sejam modificadas incorretamente, fazendo com que a aplicação deixe de funcionar quando tentar acessar os elementos da lista e isso não for possível devido a uma ligação errada entre os nós.

Uma opção para solucionar esta situação, mantendo o encapsulamento e permitindo o acesso a elementos específicos da lista por

Casos especiais de inserção e remoção

Como as operações de inserção e remoção na primeira e últimas posições são muito utilizadas, é comum existir um método específico para cada uma, facilitando assim a leitura do código que utiliza a implementação da lista encadeada em questão.

Para tratar cada um dos casos, não é necessário escrever novamente o código de inserção ou remoção, basta chamar o método que realiza a inserção para qualquer um dos casos com os parâmetros corretos, como mostra a **Listagem 6**.

Nestes casos não cabe o uso de versões sobreencarregadas dos métodos, pois a funcionalidade de cada um é ligeiramente diferente, e também se deseja distinguir esta funcionalidade textualmente no código. Assim, quando alguém ler um método que faz uso de um objeto do tipo **ListaDupEncadeada**, será fácil entender quando um novo elemento está sendo inserido no início, no final ou no meio da lista, e o mesmo vale para as operações de remoção de elementos da lista.

Testando a lista

O código apresentado até o momento é de uma classe utilitária, ou seja, que será utilizado por outras classes para auxiliar na solução de problemas que envolvem a guarda e organização de dados em memória. Para verificar o seu funcionamento é apresentada na **Listagem 7** uma classe que realiza algumas inserções em uma lista, em seguida mostra o seu conteúdo e por fim remove os elementos na ordem inversa à qual foram inseridos.

No exemplo são instanciadas seis **Strings** de teste para serem inseridas na lista. Para a inserção são chamados os métodos **Lista.insere()**, **Lista.insereInício()** e **Lista.insereFim()**.

outras classes, é a utilização de uma classe interna que fornece acesso aos elementos da lista de forma controlada. Esta solução recebe o nome de iterador (o padrão de projeto *Iterator*) e é descrita em detalhes a seguir.

Implementando um iterator

O iterador é um objeto que permite o acesso aos elementos em uma estrutura de dados sem que seja necessário conhecer como os dados estão organizados, ou seja, sem expor a representação interna da classe que implementa a estrutura de dados em questão. Outro motivo para utilizar um iterador para acessar os elementos da estrutura de dados é que mais de um cliente pode utilizar a mesma lista simultaneamente. Por exemplo: enquanto um método salva o conteúdo da lista em um arquivo, um segundo método

pode enviar os mesmos elementos pela rede. Com a implementação do método de busca apresentado anteriormente, esta tarefa de percorrer os nós da lista é facilmente realizável, porém, como já observado, a maneira como o **ListaDupEncadeada.consulta()** lida com os acessos aos elementos da lista não é eficiente.

Uma possível solução para melhorar o desempenho do método de consulta seria modificá-lo para utilizar um atributo protegido que armazene a posição atual, e a cada nova chamada ao método a referência ao dado correspondente ao nó atual seja retornada e também a referência ao nó atual seja atualizada para o próximo nó, ou o nó anterior, de acordo com a direção da busca. Esta solução também possui uma grande desvantagem, que é o fato de permitir apenas um laço de busca ocorra de cada vez, devido a utilizar apenas uma referência para o nó atual no processo de iterar pelos elementos da lista, o que também não é satisfatório.

Como a solução anterior propõe a criação de uma referência auxiliar a ser utilizada no laço que implementa a iteração com os nós da lista, esta referência poderia ser fornecida à classe que irá utilizar a lista encadeada, permitindo assim que cada vez que seja necessário percorrer a lista, uma nova referência seja criada, eliminando a restrição de poder existir apenas um laço de cada vez. Bastante promissor, porém esta ideia também apresenta uma grave falha ao fornecer uma referência do nó da lista à classe cliente, pois isso permite que os valores de ligação entre os elementos da lista sejam alterados por métodos que não são os da classe **ListaDupEncadeada**, ou seja, rompe o encapsulamento da classe.

O padrão *Iterator* aproveita os conceitos descritos nestas duas abordagens de maneira a permitir o acesso aos elementos da lista por mais de um laço simultaneamente e também sem ferir o

Listagem 7. Código da classe de teste para a lista duplamente encadeada.

```
public class TesteLista_01 {
    public static void main(String[] args) {
        ListaDupEncadeada lista = new ListaDupEncadeada();
        int val;

        String a="str teste 01";
        String b="str teste 02";
        String c="str teste 03";
        String d="str teste 04";
        String e="str teste 05";
        String f="str teste 06";

        System.out.println("Tamanho: " +lista.getTamanho());
        val = lista.inserirInicio(f);
        System.out.println("Inserindo: " + f + " -> Tamanho: " + val);
        val = lista.inserirInicio(e);
        System.out.println("Inserindo: " + e + " -> Tamanho: " + val);
        val = lista.insere(d, 2);
        System.out.println("Inserindo: " + d + " -> Tamanho: " + val);
        val = lista.insere(c, 2);
        System.out.println("Inserindo: " + c + " -> Tamanho: " + val);
        val = lista.insereFim(b);
        System.out.println("Inserindo: " + b + " -> Tamanho: " + val);
        val = lista.insereFim(a);
        System.out.println("Inserindo: " + a + " -> Tamanho: " + val);

        System.out.println("Tamanho: " +lista.getTamanho());

        for(int i = 1; i<=lista.getTamanho(); i++)
            System.out.println("Posição " + i + ":->" + lista.consulta(i));

        System.out.println("");

        System.out.println(lista.removeFim());
        System.out.println("Tamanho: " +lista.getTamanho());
        System.out.println(lista.removeFim());
        System.out.println("Tamanho: " +lista.getTamanho());
        System.out.println(lista.remove(2));
        System.out.println("Tamanho: " +lista.getTamanho());
        System.out.println(lista.remove(2));
        System.out.println("Tamanho: " +lista.getTamanho());
        System.out.println(lista.removeInicio());
        System.out.println("Tamanho: " +lista.getTamanho());
        System.out.println(lista.removeInicio());
        System.out.println("Tamanho: " +lista.getTamanho());
    }
}
```

Listagem 8. Saída do teste da lista duplamente encadeada.

```
Tamanho: 0
Inserindo: str teste 06 -> Tamanho: 1
Inserindo: str teste 05 -> Tamanho: 2
Inserindo: str teste 04 -> Tamanho: 3
Inserindo: str teste 03 -> Tamanho: 4
Inserindo: str teste 02 -> Tamanho: 5
Inserindo: str teste 01 -> Tamanho: 6
Tamanho: 6
Posição 1: ->str teste 05
Posição 2: ->str teste 03
Posição 3: ->str teste 04
Posição 4: ->str teste 06
Posição 5: ->str teste 02
Posição 6: ->str teste 01

str teste 01
Tamanho: 5
str teste 02
Tamanho: 4
str teste 03
Tamanho: 3
str teste 04
Tamanho: 2
str teste 05
Tamanho: 1
str teste 06
Tamanho: 0
```

encapsulamento dos atributos desta. Esta solução envolve a implementação de um novo meio de realizar as operações, e para isso, se necessita de uma interface, que será utilizada para manipular os dados armazenados na estrutura de dados. Nesta interface devem ser declaradas todas as operações que serão executadas para manipular os dados, neste caso, da lista encadeada.

Os relacionamentos entre esta nova interface e as classes que fazem parte do padrão *Iterator* são apresentados na **Figura 3**, onde ficam claros os relacionamentos da classe **Cliente** com a interface **IteratorLista** e com a classe **ListaDupEncadeada**. Apesar de a classe **Cliente** poder acessar diretamente a classe **ListaDupEncadeada**, a navegação por seus nós será mais eficiente se realizada pela interface **IteratorLista**.

Na **Listagem 9** é apresentada a declaração da interface para um *iterator* que define operações de navegação e consulta aos elementos de uma lista duplamente encadeada.

Nesta interface devem ser declaradas todas as operações que são permitidas a outros objetos (**Clientes**) de realizarem com os elementos armazenados na lista. Por exemplo: além das operações de navegação e consulta, podem ser declaradas operações de inserção antes ou após o nó atual, e remoção do nó atual.

Para viabilizar a manipulação dos elementos armazenados na lista, a classe que implementa a interface **IteratorLista** é criada como uma classe interna de **ListaDupEncadeada**, de acordo com a **Listagem 10**. De maneira a evitar que instâncias desta classe

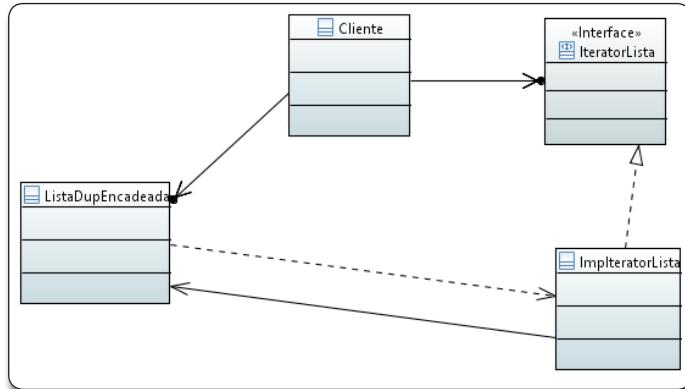


Figura 3. Diagrama de classes do Padrão Iterador

Listagem 9. Interface de um iterador simples para uma lista duplamente encadeada.

```

public interface IteradorLista {
    // Retorna o dado associado ao nó atual
    public Object dado();
    // Coloca o iterador no próximo elemento da lista e retorna o dado
    // associado ao nó atual
    public Object proximo();
    // Coloca o iterador no elemento anterior da lista e retorna o dado
    // associado ao nó atual
    public Object anterior();
    // Verifica se existe um nó após o nó atual
    public boolean temProximo();
    // Verifica se existe um nó antes do nó atual
    public boolean temAnterior();
}

```

sejam criadas de forma inadequada, ela é declarada como **private**, o que significa que apenas a classe **ListaDupEncadeada** pode instanciar e manipular objetos do tipo **ImplIteradorLista**. Este tipo de declaração garante o encapsulamento dos dados e também possibilita que outras classes consigam um acesso controlado aos mesmos, utilizando a interface definida para isso.

Para implementar o iterador é necessária uma referência a um nó da lista, declarado como o atributo **ImplIteradorLista.noAtual**. Este nó é utilizado para acessar os elementos da lista sem que seja preciso navegar por todos os seus antecessores, mas para isso é preciso garantir em todos os métodos da classe **ImplIteradorLista** que o atributo **ImplIteradorLista.noAtual** é uma referência válida antes de executar a operação solicitada. Isto porque a referência do *iterator* pode estar fora da lista, situação em que assume o valor **null**.

Este fato pode ocorrer quando em um laço se utiliza um *iterator*, e após percorrer todos os elementos armazenados na lista, o método **IteradorLista.proximo()** é invocado para que o valor do atributo **ImplIteradorLista.noAtual** seja atualizado e o valor armazenado neste nó retornado. Neste processo, a referência de **ImplIteradorLista.noAtual** será atualizada para o próximo nó, que não existe, porque o nó atual corresponde ao último elemento da lista; portanto, sendo atribuído a **ImplIteradorLista.noAtual** o valor **null**. Esta situação pode ser melhor compreendida analisando-se a **Figura 2**, acompanhando as atualizações no atributo a partir da referência **Início**. A primeira chamada a **IteradorLista.proximo()** atualiza o valor da referência **IteradorLista.noAtual** para a primeira caixa, que corresponde ao primeiro nó da lista. Logo após, a segunda chamada a **IteradorLista.proximo()** coloca **IteradorLista.noAtual** no segundo nó, e ao se realizar a quarta chamada ao método, quando **IteradorLista.noAtual** estiver referenciando o terceiro elemento, o valor a ser atribuído ao atributo não corresponde a um elemento da lista, e portanto passa a ser **null**. O mesmo ocorre quando o *iterator* está na primeira posição e o método **IteradorLista.anterior()** é chamado, ao realizar a navegação na lista em sentido contrário.

O método **ImplIteradorLista.dado()** simplesmente retorna o objeto correspondente ao dado armazenado em **ImplIteradorLista.noAtual** depois de verificar se o iterador é válido. Os métodos de navegação **ImplIteradorLista.proximo()** e **ImplIteradorLista.anterior()**, após verificarem que o iterador é válido, criam uma referência ao objeto que representa a informação guardada em **ImplIteradorLista.noAtual**, que será retornado após atualizar a referência de **noAtual** para o próximo elemento ou o anterior, dependendo do caso. Já nos métodos de verificação, o objetivo é saber se o iterador está ou não na primeira ou na última posição da lista, sendo realizados dois testes: um para saber se o iterador é válido e outro para descobrir se o próximo ou o elemento anterior é válido, de acordo com o solicitado.

Com a implementação desta classe pronta, é necessário criar métodos de acesso para o *iterator* na classe **ListaDupEncadeada** para que outras classes, ao utilizarem um objeto do tipo **ListaDupEncadeada**, possam solicitar os iteradores, que per-

mite interagir com a lista de maneira mais fácil e eficiente. Na **Listagem 11** são mostrados dois métodos: um para solicitar um iterador para o primeiro nó da lista e outro para o último nó. Em ambos os métodos, um objeto do tipo **ImplIteradorLista** é criado e o seu atributo **noAtual** é inicializado com o nó correspondente à solicitação, permitindo então a navegação pela lista de forma linear, ou seja, evitando chamadas sucessivas ao método **ListaDupEncadeada.consulta()**, que para encontrar o elemento solicitado passa por todos os nós da lista.

O mesmo conceito pode ser utilizado para incluir no iterador os métodos de inserção e remoção, aprimorando assim as suas opções e melhorando o desempenho no uso da lista encadeada.

Listagem 10. Código da classe que implementa um iterador para uma lista duplamente encadeada.

```
public class ListaDupEncadeada {  
    ...  
    private class ImplIteradorLista implements IteradorLista {  
        public No noAtual;  
  
        // Retorna o dado associado ao nó atual  
        public Object dado()  
        {  
            if(noAtual == null) return null;  
            return noAtual.dado;  
        }  
  
        // Coloca o iterador no próximo elemento da lista e retorna o dado  
        // associado ao nó atual  
        public Object proximo()  
        {  
            if(noAtual == null) return null;  
            Object obj = noAtual.dado;  
            noAtual = noAtual.proximo;  
            return obj;  
        }  
  
        // Coloca o iterador no elemento anterior da lista e retorna o dado  
        // associado ao nó atual  
        public Object anterior()  
        {  
            if(noAtual == null) return null;  
            Object obj = noAtual.dado;  
            noAtual = noAtual.anterior;  
            return obj;  
        }  
  
        // Verifica se existe um nó após o nó atual  
        public boolean temProximo() {  
            if(noAtual == null) return false;  
            return noAtual.proximo != null ? true : false;  
        }  
  
        // Verifica se existe um nó antes do nó atual  
        public boolean temAnterior()  
        {  
            if(noAtual == null) return false;  
            return noAtual.anterior != null ? true : false;  
        }  
    }  
}
```

Listagem 11. Métodos para solicitar iteradores para a lista encadeada.

```
public class ListaDupEncadeada {  
    ...  
    public IteradorLista iteradorInicio()  
    {  
        // Instanciar o novo iterador  
        ImplIteradorLista iterador = new ImplIteradorLista();  
        // Inicializar o atributo noAtual com o primeiro elemento da lista  
        // Caso a lista esteja vazia, o nó atual será nulo e nenhuma iteração será possível  
        iterador.noAtual = inicio;  
        return iterador;  
    }  
    public IteradorLista iteradorFim()  
    {  
        // Instanciar o novo iterador  
        ImplIteradorLista iterador = new ImplIteradorLista();  
        // Inicializar o atributo noAtual com o último elemento da lista  
        // Caso a lista esteja vazia, o nó atual será nulo e nenhuma iteração será possível  
        iterador.noAtual = fim;  
        return iterador;  
    }  
    ...  
}
```

Testando o iterator da lista

Como o *iterator* implementado é apenas uma nova opção de consulta aos elementos da lista, a inserção e a remoção de elementos é realizada da mesma maneira. Depois de ser instanciado um objeto do tipo **ListaDupEncadeada** e o mesmo já contiver elementos inseridos, as consultas utilizando o *iterator* podem ser realizadas de maneira sequencial, sem que seja necessário um contador para informar qual a posição desejada, uma vez que a lista será percorrida do início ao final pelo iterador, o que é realizado substituindo-se o laço **for** do primeiro teste (presente na **Listagem 7**) por um laço **while** controlado pelo conteúdo do iterador, recebido como resultado da chamada ao método **IteradorLista.proximo()**, conforme mostrado na **Listagem 12**. O resultado desse teste é exatamente o mesmo do apresentado pelo código da **Listagem 7**, diferindo apenas na manipulação da lista, que é feita através do iterador, e o número de vezes que cada elemento da lista é acessado para realizar cada uma das consultas solicitadas.

Para percorrer a lista, primeiro é necessário solicitar um iterador à mesma, que pode ser inicializado com a primeira ou a última posição. Como o método de navegação **IteradorLista.proximo()** retorna o objeto correspondente à informação armazenada no nó associado ao iterador ou, quando o iterador não é válido, o valor **null** é retornado. Esta condição é utilizada para controlar o laço de navegação pela lista encadeada. Assim, quando o último nó for consultado, a referência interna do iterador passará a ser **null** e a chamada seguinte ao método **IteradorLista.proximo()** retornará **null**, o que interrompe o laço após todos os elementos da lista serem mostrados.

Diferente da versão anterior da classe **ListaDupEncadeada**, a que faz uso do padrão *iterator* é mais eficiente por permitir percorrer toda a lista acessando cada nó apenas uma vez, e não percorrendo todo o encadeamento para cada consulta, como é o caso da implementação do método **ListaDupEncadeada.consulta()**.

Listagem 12. Modificação no método de teste para utilizar o iterador na lista encadeada.

```
public class TesteLista_02 {  
    public static void main(String[] args) {  
  
        // Iterador utilizado para percorrer a lista encadeada  
        IteradorLista iterador = lista.iteradorInicio();  
  
        // Variável para receber o elemento armazenado na lista  
        String str;  
        // Contador de posições  
        int i = 1;  
        // Laço para percorrer a lista utilizando o iterador  
        while((str = (String) iterador.proximo()) != null)  
            // Mostra o valor armazenado em cada posição da lista  
            System.out.println("Posição " + i++ + ":>" + str);  
  
    }  
}
```

O **iterator** da lista também pode ser utilizado de outra forma; verificando-se a existência de um próximo elemento antes de solicitar o dado, como mostrado na **Listagem 13**. Novamente, o resultado do teste é exatamente o mesmo, apenas alterando-se a maneira de utilizar o iterador. Esta segunda forma de acesso deve ser adotada quando não se pode permitir que o iterador se torne inválido, ou seja, não é permitido chamar o método **IteradorLista.proximo()** para o último elemento da lista e o método **IteradorLista.anterior()** para o primeiro elemento da lista.

Listagem 13. Uma segunda maneira de utilizar o iterador na lista encadeada.

```
public class TesteLista_03 {  
    public static void main(String[] args) {  
  
        // Iterador utilizado para percorrer a lista encadeada  
        IteradorLista iterador = lista.iteradorInicio();  
  
        // Contador de posições  
        int i = 1;  
        // Laço para percorrer a lista utilizando o iterador  
        while(iterador.temProximo())  
        {  
            // Mostrar o valor armazenado em cada posição da lista  
            System.out.println("Posição " + i++ + ":>" + iterador.dado());  
            iterador.proximo();  
        }  
  
    }  
}
```

A compreensão das estruturas de dados é um dos requisitos mais importantes para o desenvolvimento de aplicações de qualidade, pois permite que o programador entenda a organização dos dados na memória e assim possa aproveitar os recursos da melhor maneira, seja com relação à quantidade de memória utilizada ou otimizando o tempo necessário para o processamento dos dados.

Com o grande poder dos processadores atuais, juntamente com o grande volume de memória disponível nos diversos dispositivos computacionais, é fácil esquecer-se destes detalhes, porém o volume de dados criados e manipulados pelos usuários está aumentando rapidamente, exigindo cada vez mais tanto dos equipamentos quanto dos profissionais envolvidos das diversas áreas de TI.

Este grande volume de dados, presente em praticamente todos os ambientes computacionais, exige também novas técnicas para organizá-los, levando ao desenvolvimento de novas estruturas de dados. Ao se deparar com situações como estas, a implementação deve ser muito bem avaliada, pois já existem muitas APIs disponíveis com códigos já testados que podem ser utilizados pelos desenvolvedores e analistas, não só na linguagem Java. A utilização de uma classe própria como a apresentada deve ser considerada apenas para situações muito específicas, para as quais as APIs não apresentem uma solução satisfatória. No entanto, isso não exclui a necessidade e importância de conhecer a teoria e os conceitos adotados por estas bibliotecas.

Autor



Miguel Diogenes Matrakas

mdmatrakas@yahoo.com.br

É Mestre em Informática pela PUC-PR e doutorando em Métodos Numéricos em Engenharia, pela UFPR (Universidade Federal do Paraná). Trabalha como professor de Java há quatro anos nas Faculdades Anglo-Americanas de Foz do Iguaçu.



Links:

Artigo sobre o padrão de projeto Iterator.

<http://www.oodesign.com/iterator-pattern.html>

Tutorial sobre Padrões de projeto em Java.

http://www.tutorialspoint.com/design_pattern/

Programação Orientada a Objetos.

<http://www.dca.fee.unicamp.br/cursos/PooJava/Aulas/poopjava.pdf>

Artigo sobre Estruturas de Dados.

<http://www.javaworld.com/article/2073390/core-java/datastructures-and-algorithms-part-1.html>

Apresentação sobre Estruturas de Dados.

http://www.lca.ic.unicamp.br/~jlopez/transp_ho4_2011.pdf

Conteúdo sobre Listas Encadeadas.

<http://www.ime.usp.br/~pf/algoritmos/aulas/lista.html>

Algoritmos e Estruturas de Dados.

<http://www.das.ufsc.br/~romulo/discipli/cad-fei/Aula-EstruturasDados-04f-Listaduplamente-encadeada.pdf>

Hibernate 4: Aprenda a persistir os dados de sua aplicação

Este artigo demonstra na prática como armazenar dados utilizando a versão mais recente do Hibernate

Hoje em dia, com a necessidade de desenvolver sistemas cada vez mais complexos e que levem em conta fatores como facilidade de manutenção e reaproveitamento de código, muitos desenvolvedores, durante o processo de análise e desenvolvimento de software, têm optado pelo uso do paradigma de programação orientada a objetos. Por outro lado, o modelo relacional há muito tempo vem sendo o modelo mais aceito e utilizado quando se deseja armazenar dados, se comparado a outros modelos (rede, hierárquico, orientado a objetos e objeto-relacional).

Segundo Bauer e King (2005, p.11), os bancos de dados relacionais estão fortemente presentes no núcleo da empresa moderna, e por serem largamente utilizados em projetos de software orientados a objetos, faz-se necessário um mapeamento entre as tabelas do banco de dados e os objetos da aplicação. A fim de tornar compatível o paradigma da orientação a objetos e o paradigma de entidade e relacionamento, foram desenvolvidos frameworks de mapeamento objeto relacional (*Object Relational Mapping – ORM*).

O mapeamento objeto relacional surge como uma alternativa para os desenvolvedores de sistemas que não querem abrir mão dos benefícios que a linguagem de programação orientada a objetos possui, e que também sabem que um banco de dados puramente orientado a objetos está longe de conseguir uma boa aceitação no mercado. A ideia por trás do ORM é ter um mecanismo que faça a conversão entre os objetos do sistema e as tabelas do banco de dados relacional.

O framework ORM mais difundido atualmente é o Hibernate. Suas principais vantagens são simplificar a persistência dos dados, permitir que a aplicação permaneça totalmente orientada a objetos e também fazer

Fique por dentro

Neste artigo mostraremos como desenvolver uma aplicação Java utilizando Hibernate, MySQL e NetBeans. Aprenderemos de forma básica e simples os principais conceitos relativos ao framework, faremos uma análise geral sobre seu funcionamento, como configurar um ambiente de desenvolvimento e como realizar o mapeamento objeto-relacional, visando preencher a lacuna semântica existente entre as estruturas de tabelas e sua representação através de classes.

com que possíveis mudanças na base de dados impliquem em um menor impacto sobre a aplicação, tendo em vista que apenas os objetos relacionados com as tabelas alteradas necessitarão de mudanças. Sendo assim, ficam evidentes as vantagens da adoção do Hibernate no desenvolvimento de sistemas orientados a objetos.

Com base nisso, este artigo visa fazer um estudo sobre o Hibernate, apresentando suas características, seus principais recursos, assim como uma visão geral sobre seu funcionamento e como utilizá-lo em meio à construção de um exemplo simples de sistema de informação que mostre de forma prática como desenvolver uma aplicação orientada a objetos com acesso a um banco de dados relacional. Para tanto, serão empregados à linguagem de programação Java, o ambiente de desenvolvimento NetBeans e o sistema de gerenciamento de banco de dados (SGBD) MySQL. O Hibernate será usado como a camada de persistência que deverá fazer a interface entre a aplicação e o MySQL.

Mapeamento objeto relacional

Segundo Bauer e King (2005, p.23), mapeamento objeto relacional significa “persistir de maneira automática e transparente os objetos de um aplicativo para tabelas em um banco de dados relacional. Em essência, transformar dados de uma representação

para a outra". Se o banco de dados possui uma tabela chamada **Pedido**, a aplicação possuirá, provavelmente, uma classe denominada **Pedido**. Essa classe definirá atributos, que serão usados para receber e alterar os dados dos campos das tabelas, além de métodos para calcular os juros, valor total do pedido, entre outras operações relacionadas à tabela.

No desenvolvimento de um sistema, muitas vezes o programador dedica uma parte considerável do seu tempo construindo comandos em linguagem de banco de dados para realizar o armazenamento dos dados no banco de dados relacional. A presença de uma camada intermediária de mapeamento objeto relacional facilita muito a vida do programador, pois ela é responsável por traduzir as estruturas e operações do sistema orientado a objetos para a estrutura existente no banco de dados relacional.

A persistência diz respeito ao armazenamento de dados que estão em meio volátil, como a memória RAM, para dispositivos de memória secundária, como o disco rígido. Em outras palavras, consiste em manter os dados em meio físico recuperável, como um banco de dados ou um arquivo, de modo a garantir a permanência das informações de um determinado estado de um objeto lógico.

Como vantagens, o mapeamento objeto relacional proporciona:

- **Produtividade:** com a eliminação dos códigos SQL no código-fonte, as classes passam a ser mais simples e com isso o sistema é desenvolvido em um tempo menor;
- **Manutenibilidade:** tendo em vista que o número de linhas de código do sistema diminui, o trabalho de manutenção também será menor;
- **Desempenho:** o tempo economizado no desenvolvimento pode ser dedicado a implementar melhorias no sistema;
- **Independência de fornecedor:** Abstrai do aplicativo o sistema de banco de dados e oferece maior portabilidade. Assim, o programador pode mudar de SGBD sem ter que realizar alterações no software.

O objetivo do ORM é construir um código mais uniforme e que utilize de fato as características da linguagem orientada a objetos. Dessa forma, sistemas em que a maior parte da lógica da aplicação encontra-se dentro do banco de dados ou mesmo fazem uso extensivo de *stored procedures* e *triggers* não irão se beneficiar do Hibernate, assim como do mapeamento objeto relacional, pois não há necessidade de um modelo de objetos para trabalhar a lógica do sistema. Nesse caso, o ideal seria tirar a lógica de negócios de dentro do banco e transferir para a aplicação.

Hibernate

O Hibernate é uma ferramenta open source de mapeamento objeto relacional para aplicações Java de grande aceitação entre os desenvolvedores. Distribuído com a licença LGPL, foi criado por Gavin King em 2001, sendo, sem dúvida, o framework de persistência de dados mais utilizado, sobretudo por dar suporte a classes desenvolvidas com agregações (aqueles classes que usam outras classes em suas operações), herança, polimorfismo,

composições e coleções, por implementar a especificação *Java Persistence API* (JPA), por não restringir a arquitetura da aplicação e por ser amplamente documentado. Segundo a documentação oficial: "o Hibernate pretende retirar do desenvolvedor cerca de 95% das tarefas mais comuns de persistência de dados".

Sua principal característica é a transformação das classes em Java para tabelas de dados (e dos tipos de dados Java para os da SQL). O Hibernate gera as chamadas SQL e libera o desenvolvedor do trabalho manual de transformação, mantendo o programa portável para quaisquer bancos de dados SQL.

A partir da versão 5 do Java a configuração do Hibernate passou a ser realizada através de anotações do JPA, deixando de lado os complexos arquivos XML. Anotações são metadados que são adicionados ao código-fonte para descrever características dele e como vantagens possibilitou um código mais intuitivo e mais enxuto.

Quando utilizamos anotações, simplesmente as adicionamos sobre os atributos/métodos de acordo com o que projetamos. Após isso, o *Java Runtime Environment* (JRE) analisa essas anotações. Para ler as anotações e aplicar as informações de mapeamento, o Hibernate utiliza Reflection. Para mais informações, leia o artigo "Entendendo Anotações", publicado na Easy Java Magazine 25.

Além de realizar o mapeamento objeto relacional, principal característica do *framework*, o Hibernate disponibiliza outras funcionalidades, disponibilizadas através de cinco projetos. A seguir veremos cada um e suas respectivas funções.

O primeiro e mais conhecido é o Hibernate ORM, que é o projeto original, *core* do Hibernate, sendo dedicado à persistência de objetos em bancos de dados relacionais. O Hibernate Search é o responsável por implementar buscas textuais no modelo de dados das aplicações. O Hibernate Validator consiste em um conjunto de anotações responsáveis por implementar a validação nas entidades da aplicação. O Hibernate OGM, por sua vez, trabalha a persistência em bancos de dados do tipo NoSQL. Por fim, o Hibernate Tools disponibiliza uma série de ferramentas em linhas de comando que automatizam diversas tarefas referentes aos outros projetos Hibernate. Apesar de todos esses projetos fazerem parte da mesma *brand*, quando se fala em Hibernate, a referência é sempre relacionada ao Hibernate ORM, o projeto embrião, com maior popularidade.

O Hibernate pode ser utilizado por uma aplicação Java diretamente ou podemos acessá-lo através de outro *framework*. Isto é, podemos invocar o Hibernate através de uma aplicação Swing, ou qualquer outra aplicação Java que mantenha acesso a um banco de dados, ou mesmo a partir de um *framework* como Spring MVC, JSF, etc.

O HQL (*Hibernate Query Language*) é um dialeto SQL próprio do Hibernate. Trata-se de uma poderosa linguagem de consulta que se parece com o SQL, mas é totalmente orientada a objetos, incluindo os conceitos de herança, polimorfismo e encapsulamento. Sendo assim, o desenvolvedor tem a possibilidade de trabalhar diretamente com objetos persistentes e suas propriedades, ao invés de realizar as operações diretamente nas tabelas e colunas da base

de dados, aumentando assim a distância entre o desenvolvimento das regras de negócio e o banco de dados.

Ademais, recentemente foi lançada a versão 4 do Hibernate, que trouxe várias melhorias e facilidades para os desenvolvedores. Entre as novidades podemos citar a introdução do ServiceRegistry, que é uma nova forma do Hibernate gerenciar os seus serviços. Outra novidade diz respeito a melhorias na performance do framework com a introdução de Multi-Tenancy. O modelo arquitetural Multi-Tenancy permite que uma instância da aplicação ou banco de dados, sendo executada em um servidor, possa ser compartilhada entre diversos inquilinos ou *tenants*, com a garantia de isolamento dos dados.

Hibernate e JPA

Algum tempo depois do Hibernate fazer sucesso entre os desenvolvedores, a Sun reconheceu que o mapeamento objeto-relacional era algo necessário no desenvolvimento de aplicações e criou a especificação JPA baseando-se nas funcionalidades que o Hibernate já disponibilizava.

Como a Sun tinha maior alcance e influência sobre os desenvolvedores do que os criadores do Hibernate, foi natural que a JPA se tornasse conhecida rapidamente. Após o lançamento da especificação, os criadores do Hibernate a implementaram para manter o framework compatível com as outras especificações da Java EE (EJB, JSF, etc.) e os outros frameworks que utilizam a JPA.

Uma situação que gera dúvidas nos desenvolvedores é que apesar do Hibernate ter surgido antes da JPA e ter sido utilizado como referência para a sua especificação, ele também a implementa, ou seja, pode ser utilizado como um framework independente, levando em conta tão somente suas características, ou pode ser usado como uma implementação da JPA. Para empregar o Hibernate como implementação da JPA, deve-se utilizar as anotações e classes que se encontram no pacote `javax.persistence.*`.

Atualmente, a JPA se encontra na versão 2.1, lançada em maio de 2013 através da JSR 338. Esta nova especificação, que compõe a lista de tecnologias da plataforma Java EE 7, mantém o foco na atualização da API, incorporando melhorias e novos recursos solicitados pela comunidade.

Configuração

Após conhecermos os principais conceitos relacionados ao Hibernate, vamos partir para a apresentação da parte prática, desenvolvendo uma aplicação que possibilitará ao leitor visualizar como o framework funciona, assim como alguns dos seus recursos.

No entanto, antes de começarmos a codificar, é importante instalar os softwares que nos auxiliarão no desenvolvimento e também obter as bibliotecas necessárias para a utilização do Hibernate.

Download e instalação das bibliotecas

Existem duas maneiras de configurar o ambiente para usar o Hibernate. A primeira é através de uma configuração automatizada, utilizando o Maven, por exemplo, e a segunda é de forma manual. Para este artigo optamos pela segunda opção, visto que

para a utilização da primeira opção é preciso conhecer e dominar o Maven.

Antes de começar a utilizar o Hibernate é necessário baixar do site oficial do *framework* as bibliotecas obrigatórias e todas as suas dependências. Neste artigo utilizaremos a versão 4.3.8 Final, que no momento da elaboração desse material era a última versão lançada. A URL do site oficial se encontra na seção **Links**. Após acessar o site, busque pela seção de downloads e localize o Hibernate ORM. Em seguida, realize o download da versão 4.3.8, que está disponível em dois tipos de arquivos compactados: zip e tar.gz. Faça o download do arquivo de acordo com o formato de sua preferência.

Finalizado o download, descompacte o arquivo. A pasta gerada contém, entre outros arquivos, a documentação em inglês, manuais em várias línguas (inclusive português), o código-fonte e os jars (as bibliotecas), que se encontram dentro da pasta *lib* e é o que nos interessa nesse momento.

Dentro da pasta *lib* existem algumas subpastas. São elas:

- */lib/required/* – contém todos os jars obrigatórios para utilização do Hibernate e que devem ser incluídos no *classpath* do projeto. Dentro dessa pasta está o hibernate-core, artefato principal do Hibernate que possui toda a parte central do *framework*;
- */lib/jpa/* – contém a biblioteca hibernate-entitymanager, que fornece a implementação do JPA ao Hibernate;
- */lib/envers/* – nessa pasta encontra-se o hibernate-envers, módulo opcional que fornece o histórico de auditoria de alterações sobre as entidades;
- */lib/optional/* – contém as bibliotecas necessárias para recursos opcionais do Hibernate. Dentro desse diretório podemos citar as bibliotecas do hibernate-ehcache, que provê integração entre Hibernate e EhCache, um cache de segundo nível; hibernate-infinispan, que provê integração entre o Hibernate e o Infinispan, que também é um cache de segundo nível; hibernate-c3p0, que provê a integração entre o Hibernate e o C3P0 connection pool library, biblioteca usada para prover recursos de connection pooling; e o hibernate-proxool, que provê a integração entre Hibernate e Proxool connection pool library, também utilizada para prover recursos de connection pooling.

O primeiro passo é copiar em uma nova pasta, que vamos chamar de *lib*, todos os arquivos .jar que estão dentro da */lib/required/*. Ao todo são 10 arquivos.

Além dessas bibliotecas, é preciso ter o driver JDBC do banco de dados. O driver é um componente de software que permite que a aplicação interaja com o banco de dados. Apesar de utilizarmos o Hibernate, por trás dos panos ele faz uso de chamadas JDBC, sendo assim necessária a adição do driver.

Na seção **Links** encontra-se o endereço para *download* do driver. Trata-se de um arquivo zip. Feito o *download*, descompacte o pacote e extraia dele o arquivo *mysql-connector-java-5.1.34-bin.jar*, colocando-o junto com os arquivos da pasta *lib*. Sendo assim, a pasta *lib* conterá todas as bibliotecas visualizadas na **Figura 1**. Com essas bibliotecas é possível construir nossa aplicação.

Nome	Data de modificação	Tipo	Tamanho
antlr-2.7.7.jar	01/05/2014 12:50	Executable Jar File	435 KB
dom4j-1.6.1.jar	01/05/2014 12:49	Executable Jar File	307 KB
hibernate-commons-annotations-4.0.5.F...	15/07/2014 09:31	Executable Jar File	74 KB
hibernate-core-4.3.8.Final.jar	06/01/2015 12:10	Executable Jar File	5.150 KB
hibernate-jpa-2.1-api-1.0.0.Final.jar	01/05/2014 12:50	Executable Jar File	111 KB
jandex-1.1.0.Final.jar	01/05/2014 12:50	Executable Jar File	75 KB
javassist-3.18.1-GA.jar	01/05/2014 12:49	Executable Jar File	698 KB
jboss-logging-3.1.3.GA.jar	01/05/2014 12:50	Executable Jar File	56 KB
jboss-logging-annotations-1.2.0.Beta1.jar	01/05/2014 12:50	Executable Jar File	12 KB
jboss-transaction-api_1.2_spec-1.0.0.Fina...	01/05/2014 12:50	Executable Jar File	28 KB
mysql-connector-java-5.1.34-bin.jar	17/10/2014 07:05	Executable Jar File	938 KB

Figura 1. Bibliotecas obrigatórias

Ambiente de desenvolvimento

Neste artigo optamos por utilizar o NetBeans, que é um ambiente de desenvolvimento integrado (IDE) gratuito e de código aberto. O NetBeans oferece aos desenvolvedores as ferramentas necessárias para criar aplicativos profissionais para desktop, empresariais, web e para dispositivos móveis. Na seção **Links** você encontrará o endereço para download do software. Neste artigo foi utilizada a versão 8.0.1.

O processo de instalação do NetBeans é bastante simples, sendo necessário apenas ter o Java SE Development Kit (JDK) 6 ou superior instalado no sistema. Na seção **Links** também está disponível o endereço onde pode ser obtido o JDK. Caso o leitor tenha dificuldade em instalá-lo, neste mesmo tópico foi disponibilizado o endereço de um material que mostra o passo a passo de como instalar o NetBeans.

Os passos para instalação do IDE são os que seguem:

1. Após baixar o instalador, execute-o clicando duas vezes com o botão esquerdo do mouse e espere descompactar os arquivos. Como o instalador já está em português, clique no botão *Próximo* para continuar, como mostra a **Figura 2**;
2. Na próxima tela, marque a opção *Eu aceito os termos no contrato de licença* e clique no botão *Próximo*;
3. A etapa seguinte é dependente da instalação do JDK. Se você ainda não o instalou, a instalação do IDE não poderá continuar. Mas se já instalou, notará que o caminho do JDK automaticamente será preenchido e aparecerá na tela. Como está tudo certo, apenas clique no botão *Próximo* para continuar a instalação, conforme a **Figura 3**;
4. No passo seguinte, será mostrada a pasta de instalação do software e algumas informações adicionais como, por exemplo, o tamanho total da instalação e a opção que permite ao NetBeans verificar atualizações de forma automática. Clique no botão *Instalar* e espere o processo terminar;
5. Concluída a instalação, basta clicar no botão *Finalizar*.

O sistema gerenciador de banco de dados

O banco de dados que utilizaremos será o MySQL, uma opção gratuita, de código aberto, simples e muito utilizado pelos desenvolvedores. Entre suas características podemos citar: a facilidade no manuseio, excelente desempenho e estabilidade, portabilidade, interface gráfica de fácil utilização, etc.

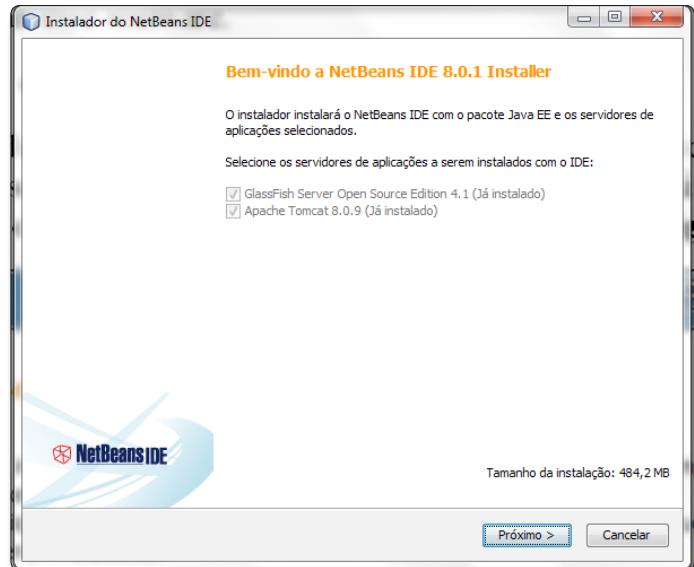


Figura 2. Tela de boas-vindas do NetBeans

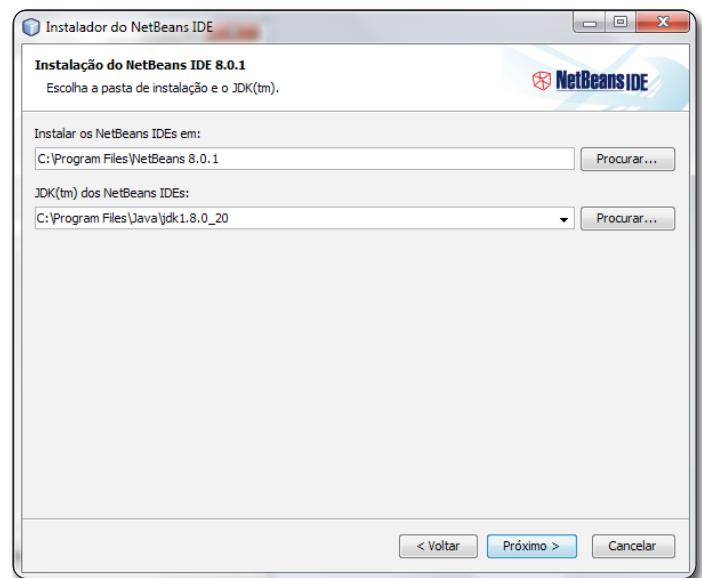


Figura 3. Escolha da pasta de instalação da IDE e do JDK

Na seção **Links** você encontrará o endereço para download e mais informações sobre o software. A versão adotada neste artigo foi a 5.6.22 Community Server.

Após descarregar o instalador do MySQL os seguintes passos devem ser executados para instalar o banco de dados no sistema:

1. Clique no instalador e depois em executar. Feito isso, uma nova janela será exibida, solicitando que o usuário aguarde enquanto o Sistema Operacional configura o instalador do SGBD;
2. Após o Sistema Operacional iniciar a instalação, uma tela deve ser mostrada (conforme a **Figura 4**). Essa tela exibe os termos do contrato de licença do produto. Aceite os termos e clique em *Next*;
3. O próximo passo é escolher o tipo de instalação. Abaixo de cada tipo existe uma breve explicação sobre cada um. O usuário deve

Hibernate 4: Aprenda a persistir os dados de sua aplicação

escolher a opção que melhor atenda aos seus propósitos. Por fim, clique em *Next*, como sugere a **Figura 5**;

4. A próxima tela oferece ao usuário a opção de escolher os produtos que deseja instalar. Feito a escolha, clique em *Next*;

5. Em seguida, é possível ver os produtos escolhidos no passo anterior, conforme a **Figura 6**. Clique então em *Execute*, para dar sequência à instalação;

6. Após a instalação ser concluída, o que pode ser visto através da mudança da mensagem presente na coluna *Status*, que passará de *Ready to Download* para *Complete*, clique em *Next*;

7. Finalizada a instalação, o próximo passo é configurar o MySQL Server. Sendo assim, clique mais uma vez *Next* (ver **Figura 7**);

8. Nesse momento o usuário deve escolher o tipo de configuração para o MySQL Server. Neste passo vamos manter os valores padrão e clicar em *Next*;

9. A tela seguinte (**Figura 8**) pede para definir a senha do administrador. Portanto, informe uma senha e clique em *Next*;

10. Feito isso, a próxima tela permite ao usuário configurar o MySQL como um serviço do Sistema Operacional Windows. Mantenha os valores *default* e clique em *Next*;

11. A janela que aparecerá detalha todos os passos de configuração que serão aplicados. Confirmadas estas informações, clique em *Execute* para que a configuração seja realizada, como pode ser visto na **Figura 9**;

12. Após todas as configurações serem aplicadas, uma tela

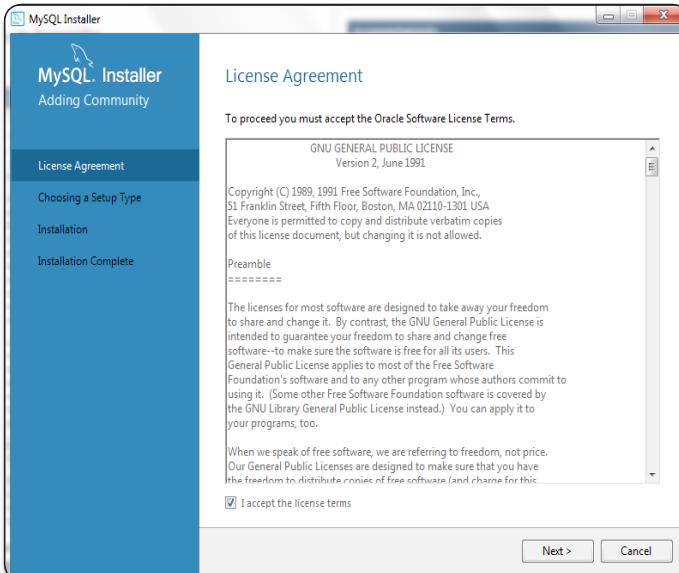


Figura 4. Termos do contrato de licença.

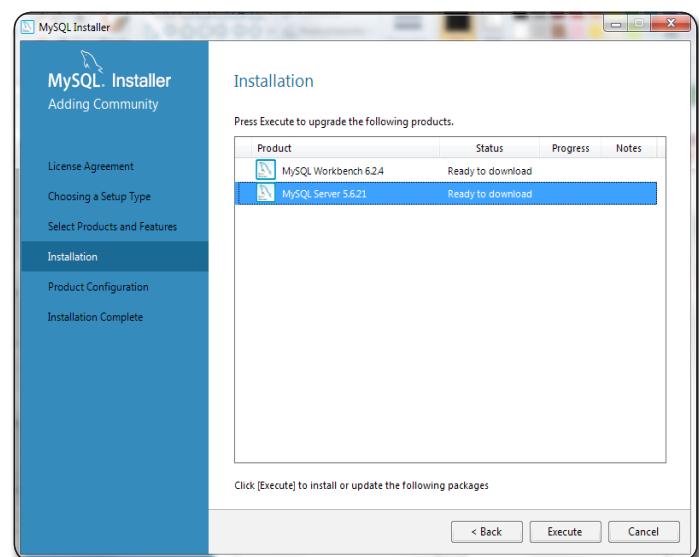


Figura 6. Produtos que serão instalados.

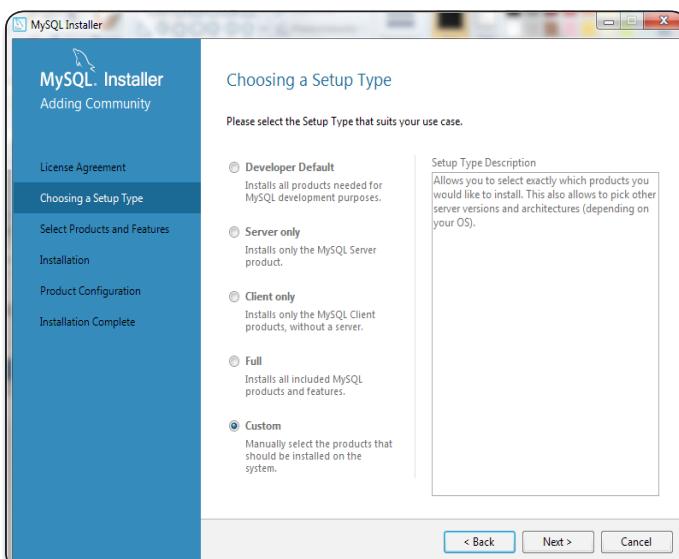


Figura 5. Escolhendo o tipo de instalação.

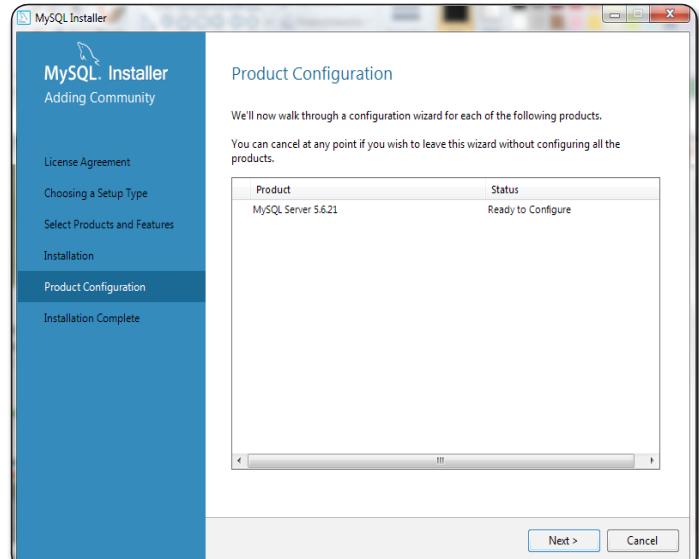


Figura 7. Configuração do MySQL Server.

mostrando o fim do processo de configuração aparecerá. Clique então em *Finish*;

13. Logo após, uma nova janela informa que a configuração do MySQL Server foi realizada. Essa janela, apresentada na **Figura 10**, permite que o usuário cancele todas as configurações realizadas antes de serem aplicadas. Caso o usuário não queira cancelar e deseje aplicar as configurações, basta clicar em *Next*;

14. Por fim, aparecerá uma janela explicitando que a instalação foi concluída. Neste momento, apenas clique em *Finish*.

Desenvolvendo o cadastro de funcionários

Agora que temos o banco de dados instalado, assim como o ambiente de desenvolvimento e as bibliotecas necessárias para rodar o Hibernate, vamos partir para a elaboração de uma aplicação desktop. O objetivo dessa aplicação é gerenciar o cadastro de funcionários de uma companhia.

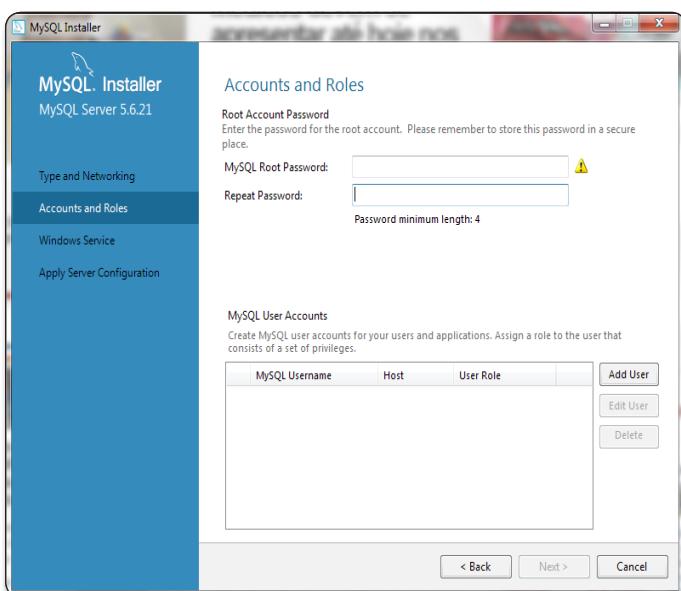


Figura 8. Definindo a senha do administrador.

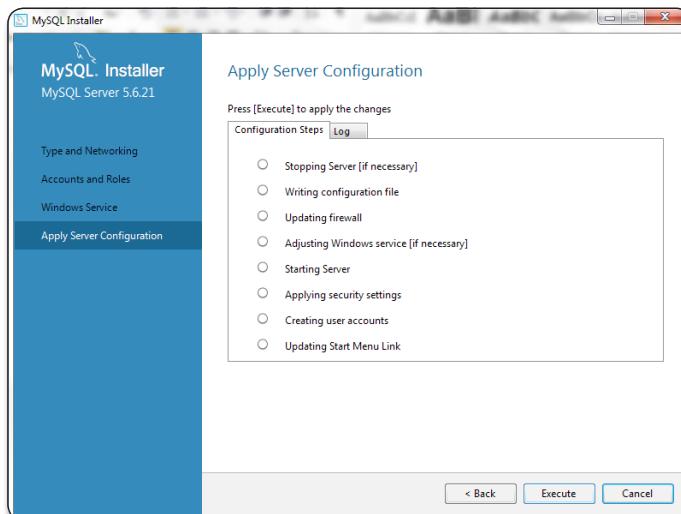


Figura 9. Aplicando configuração no MySQL Server.

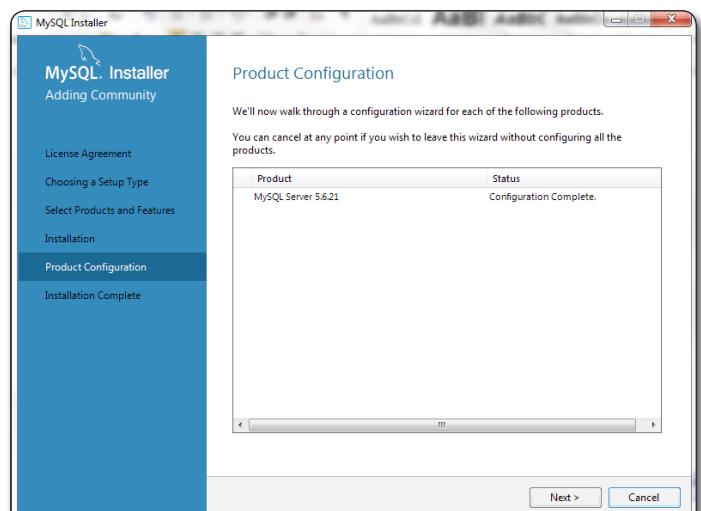


Figura 10. Configuração do MySQL completada.

Com o intuito de manter um aspecto didático e propiciar o fácil entendimento desse material, o sistema que construiremos conterá apenas uma classe, **Funcionario**. A partir dela serão construídas as principais operações realizadas sobre um cadastro, como salvar, atualizar, listar funcionários e excluir.

As informações que cada funcionário carregará são: código, nome, CPF, cargo, endereço e salário.

Criando o banco de dados

O primeiro passo é criar o banco de dados. Com o MySQL instalado, abra-o e crie o banco *empresa* conforme o script SQL indicado na **Listagem 1**. Esse banco contém apenas uma tabela, chamada *Funcionario*.

Listagem 1. Script para criação do banco de dados.

```
01. CREATE database empresa;
02. USE EMPRESA;
03.
04. CREATE TABLE FUNCIONARIO (
05.   CODIGO_INT NOT NULL AUTO-INCREMENT,
06.   NOME VARCHAR(100),
07.   CARGO VARCHAR(30),
08.   SALARIO DOUBLE,
09.   CPF VARCHAR(20),
10.  ENDERECO VARCHAR (100),
11.  PRIMARY KEY (ID)
12. );
```

Criação do projeto no NetBeans

Com o banco de dados pronto, vamos partir para o Java, criando um novo projeto no NetBeans. Como utilizaremos annotations, é importante que o projeto utilize o Java 1.5 ou superior. Dito isso, com o NetBeans aberto, clique em *Arquivo > Novo Projeto (Ctrl-Shift-N)* e na tela que aparecer, selecione *Java* na coluna *Categorias* e *Aplicação Java* na coluna *Projetos*. Logo após, clique em *Próximo*, conforme a **Figura 11**.

Hibernate 4: Aprenda a persistir os dados de sua aplicação

Na tela seguinte é necessário informar o nome do projeto. No nosso caso, nomeie o projeto como "HibernateJM". Por fim, clique em *Finalizar*, assim como mostra a **Figura 12**.

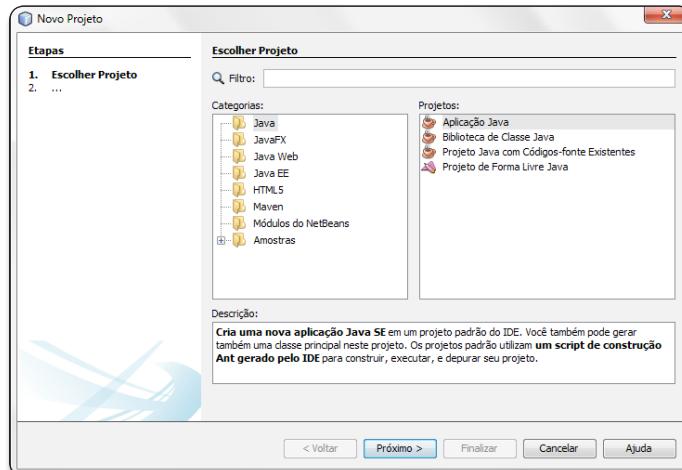


Figura 11. Escolha do tipo de projeto

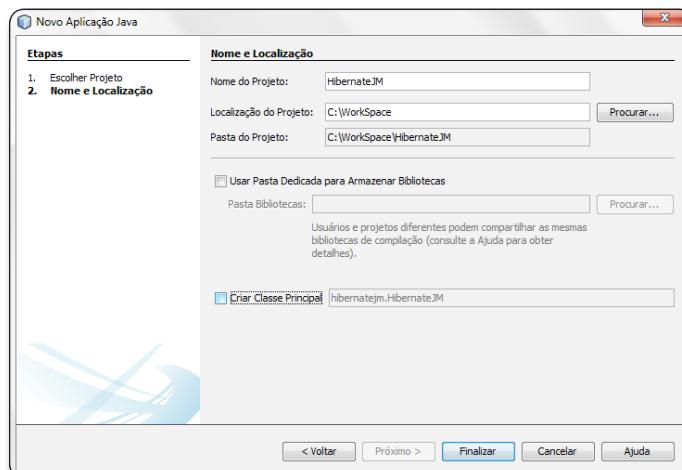


Figura 12. Criando um novo projeto

Concluída a criação do projeto, a próxima etapa consiste em adicionar ao *classpath* deste as bibliotecas do Hibernate, assim como a biblioteca referente ao driver do MySQL. Essas bibliotecas encontram-se na pasta *lib* criada anteriormente.

Para adicionar as bibliotecas ao *classpath*, clique com o botão direito do mouse sobre o projeto e escolha a opção *Propriedades*. Na tela que aparecer, selecione *Bibliotecas* na coluna *Categorias* e, com a aba *Compilar* selecionada, clique em *Adicionar JAR/Pasta*. Logo após, localize a pasta *lib*, selecione todas as bibliotecas e as adicione ao projeto. Por fim, clique em *Ok*, como mostra a **Figura 13**.

Construindo a entidade

Nosso próximo passo é criar e mapear a entidade **Funcionario**, lançando mão de *annotations*. Para isso, devemos criar a classe **Funcionario**. Essa é a nossa classe de negócio da aplicação e será mapeada pelo Hibernate como uma tabela no banco de dados.

Classes desse tipo são classes Java simples, definidas como POJOs (*Plain Old Java Object*). Elas contêm todos os seus atributos encapsulados através dos métodos de acesso get e set, e também disponibilizam o construtor padrão.

A classe de negócio ficará dentro do pacote **com.jm.entidade**. Para criá-lo, clique com o botão direito do mouse sobre o projeto **HibernateJM**, selecione *Novo > Pacote Java* e informe seu nome. Com o pacote criado, clique com o botão direito do mouse sobre ele, escolha a opção *Novo > Classe Java*, conforme a **Figura 14**, e na tela que aparecer dê o nome **Funcionario** à classe.

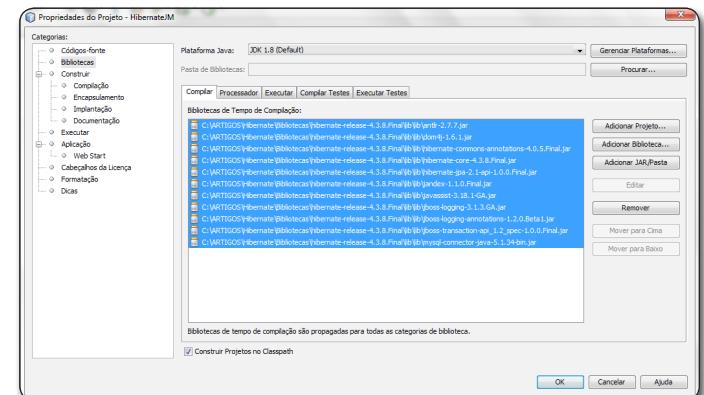


Figura 13. Adicionando as bibliotecas ao projeto

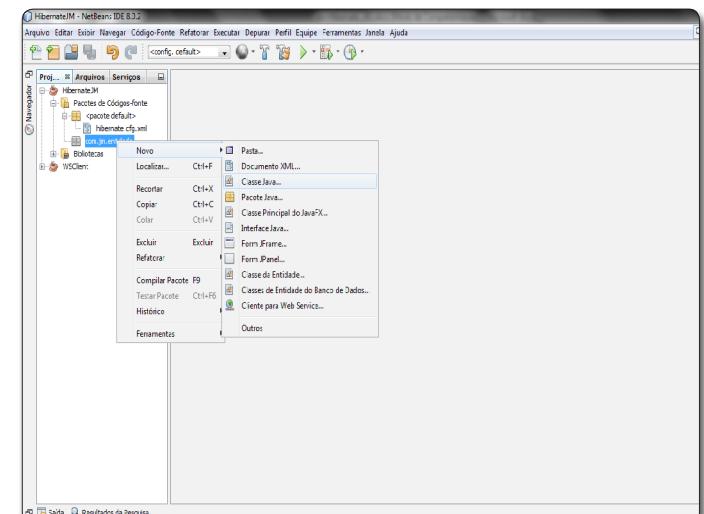


Figura 14. Criando a classe Funcionario

Note que o POJO mostrado na **Listagem 2** possui seis atributos (**codigo**, **nome**, **cpf**, **salario**, **endereco** e **cargo**), assim como seus respectivos métodos get e set e também o construtor padrão.

Na linha 5 podemos verificar a anotação **@Entity**, principal anotação do JPA. Ela aparece antes do nome da classe e sinaliza que haverá uma tabela relacionada a essa classe no banco de dados e que os objetos desta serão persistidos.

Já a anotação **@id**, vista na linha 11, é utilizada para indicar qual atributo de uma classe anotada com **@Entity** será mapeado como

a chave primária da tabela correspondente à classe. Sendo assim, a propriedade **codigo** manterá um único valor de identificação para cada um dos funcionários cadastrados.

Uma anotação que geralmente acompanha **@id** é a anotação **@GeneratedValue**, presente na linha 13. Esta serve para indicar que o valor do atributo que compõe a chave primária deve ser gerado pelo banco no momento em que um novo registro for inserido.

Ainda podemos utilizar as anotações **@Column** e **@Table**, que são usadas para personalizar o nome das tabelas e das colunas, bastando colocar o nome entre parênteses. No nosso caso, como não adicionamos essa informação, fica considerado que o nome das tabelas e colunas será exatamente igual ao nome das classes e suas propriedades.

Por fim, o método **toString()** foi criado na linha 85 para que seja possível conseguir uma representação textual de um objeto do tipo

Funcionario. Perceba que todas as informações de um funcionário podem ser obtidas em forma de texto invocando esse método.

Configurando a aplicação

Nesse momento precisamos dizer ao Hibernate onde está nosso banco de dados e como se conectar a ele. Existem três modos diferentes para configurar a *engine* do Hibernate, a saber: instanciando um objeto de configuração (**org.hibernate.cfg.Configuration**) e inserindo as suas propriedades programaticamente; utilizando um arquivo de extensão **.properties** com as suas configurações e indicando os arquivos de mapeamento programaticamente; ou mesmo usando um arquivo XML (**hibernate.cfg.xml**). Nesse artigo, utilizaremos o arquivo XML para configurar o Hibernate, pois é o modo mais simples e o mais difundido pelos desenvolvedores.

Listagem 2. Código da classe Funcionario.

```
01. package com.jm.entidade;
02.
03. //imports omitidos
04.
05. @Entity
06. @Table
07. public class Funcionario implements Serializable {
08.
09.     private static final long serialVersionUID = 1L;
10.
11.     @Id
12.     @Column
13.     @GeneratedValue(strategy = GenerationType.AUTO)
14.     private int codigo;
15.
16.     @Column
17.     private String nome;
18.     @Column
19.     private String cargo;
20.     @Column
21.     private double salario;
22.     @Column
23.     private String cpf;
24.     @Column
25.     private String endereco;
26.
27.     public Funcionario() {}
28.
29.     public Funcionario(String nome, String cargo, double salario,
30.                         String cpf, String endereco) {
31.         this.nome = nome;
32.         this.cargo = cargo;
33.         this.salario = salario;
34.         this.cpf = cpf;
35.         this.endereco = endereco;
36.     }
37.     public int getId() {
38.         return codigo;
39.     }
40.     public void setId(int id) {
41.         this.codigo = id;
42.     }
43.     }
44.
45.     public String getNome() {
46.         return nome;
47.     }
48.
49.     public void setNome(String nome) {
50.         this.nome = nome;
51.     }
52.
53.     public String getEndereco() {
54.         return endereco;
55.     }
56.
57.     public void setEndereco(String endereco) {
58.         this.endereco = endereco;
59.     }
60.
61.     public String getCargo() {
62.         return cargo;
63.     }
64.
65.     public void setCargo(String cargo) {
66.         this.cargo = cargo;
67.     }
68.
69.     public double getSalario() {
70.         return salario;
71.     }
72.
73.     public void setSalario(double salario) {
74.         this.salario = salario;
75.     }
76.
77.     public String getCpf() {
78.         return cpf;
79.     }
80.
81.     public void setCpf(String cpf) {
82.         this.cpf = cpf;
83.     }
84.
85.     public String toString() {
86.         return "Funcionario [codigo=" + codigo + ", nome=" + nome +
87.                 endereco+" + endereco +
88.                 ", cpf=" + cpf + ", cargo=" + cargo + ", salario=" + salario+"]";
89.     }
}
```

Sendo assim, devemos criar o arquivo *hibernate.cfg.xml*, que deve ficar fora de qualquer pacote existente na aplicação. Para isso, clique com o botão direito sobre o projeto, selecione *Novo > Documento XML* e o chame de *hibernate.cfg*. Seu conteúdo deve ser semelhante ao da **Listagem 3**.

Listagem 3. Conteúdo do arquivo hibernate.cfg.xml.

```
01. <?xml version="1.0" encoding="UTF-8"?>
02. <!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN" "http://hibernate.sourceforge.net/
   hibernate-configuration-3.0.dtd">
03. <hibernate-configuration>
04.   <session-factory>
05.     <property name="hibernate.dialect">org.hibernate.dialect.MySQL
        Dialect</property>
06.     <property name="hibernate.connection.driver_class">
        com.mysql.jdbc.Driver</property>
07.     <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/
        empresa?zeroDateTimeBehavior=convertToNull</property>
08.     <property name="hibernate.connection.username">root</property>
09.     <property name="hibernate.connection.password">1234</property>
10.     <property name="hibernate.show_sql">true</property>
11.     <mapping class="com.jm.entidade.Funcioario"/>
12.   </session-factory>
13. </hibernate-configuration>
```

Este arquivo começa com a definição do DTD (*Document Type Definition*) e na linha 3 temos o elemento raiz **<hibernate-configuration>**, seguido pelo elemento **<session-factory>**, na linha 4, local onde se inicia a configuração da conexão com o banco de dados e também são adicionadas as informações de mapeamento.

As propriedades configuradas são analisadas a seguir:

- **dialect (linha 5):** especifica o dialeto com o qual o Hibernate se comunicará com a base de dados, isto é, informa ao Hibernate quais comandos SQL gerar diante de um banco de dados relacional em particular;
- **connection.driver_class (linha 6):** especifica o nome da classe do driver JDBC utilizado. No nosso caso, trata-se do driver para o MySQL;
- **connection.url (linha 7):** especifica a URL de conexão com o banco de dados;
- **connection.username (linha 8):** refere-se ao nome de usuário com o qual o Hibernate deve se conectar ao banco de dados;
- **connection.password (linha 9):** especifica a senha do usuário com o qual o Hibernate deve se conectar ao banco;
- **show_sql (linha 10):** flag que permite tornar visível no console o SQL que o Hibernate está gerando. Assim, o desenvolvedor pode copiar e colar os comandos no banco de dados para verificar se está tudo conforme o esperado;
- **mapping (linha 11):** informa as classes que estão mapeadas para realizar operações no banco. Nesse caso, a classe **Funcioario** está mapeada para realizar operações sobre a tabela de mesmo nome.

Como mencionado anteriormente, uma das principais características do Hibernate é a independência de fornecedor com

relação ao SGBD. No nosso projeto, utilizamos o MySQL, mas a maioria dos bancos de dados recebem suporte por parte do framework e por isso poderiam substituir a nossa opção sem gerar problemas.

Para alterar o SGBD, basta modificar os atributos **connection.url**, **connection.driver_class** e **dialect**. Assim, atualizando apenas três linhas do arquivo de configuração é possível trocar o SGBD utilizado por outro, não sendo necessária qualquer alteração no código da aplicação. Além disso, não se esqueça de trocar o **driver** do banco.

Criando a conexão com o banco de dados

Configurada a aplicação, agora vamos criar a classe auxiliar **HibernateUtil**, responsável por entregar uma instância de **SessionFactory** ao contexto da aplicação.

Com o intuito de manter a organização do projeto, a classe **HibernateUtil** será criada dentro do pacote **com.jm.util**. Para criar esse pacote, clique com o botão direito do mouse sobre o projeto **HibernateJM**, escolha *Novo > Pacote Java* e informe “**com.jm.util**” como nome.

Feito isso, crie a classe **HibernateUtil**. Para tanto, clique com o botão direito do mouse sobre o pacote recém-criado, escolha a opção *Novo > Classe Java* e insira “**HibernateUtil**” no campo relacionado ao nome. O código dessa classe pode ser visto na **Listagem 4**.

A **SessionFactory** é uma classe de infraestrutura do Hibernate que implementa o *design pattern Abstract Factory* para construir instâncias de objetos do tipo **org.hibernate.Session**. Essas instâncias são usadas para realizar as tarefas de persistência do framework, sendo que cada instância de **SessionFactory** possui um estado interno imutável, ou seja, uma vez que ela é criada, os seus atributos internos não serão modificados. Este estado interno inclui o metadata referente ao mapeamento Objeto Relacional das entidades do sistema, e também a referência de quem irá fornecer as conexões com o banco de dados. Por esse motivo, a classe **HibernateUtil** utiliza-se do método estático **getSessionFactory()**, visto na linha 12, para oferecer sempre a mesma instância de um **SessionFactory** no contexto da aplicação. Por fim, na linha 21, temos o método privado estático **buildSessionFactory()**, que é responsável por construir a instância de **SessionFactory**. Essa construção é feita com base no arquivo de configuração do Hibernate.

Persistindo dados no banco

Agora que a classe mapeada, o arquivo de configuração e a classe auxiliar foram implementadas, criaremos uma classe que será responsável por viabilizar todas as operações de persistência que serão disponibilizadas pela aplicação exemplo. Para isso, devemos criar uma classe dentro do pacote **com.jm.acessobd**.

Primeiramente, no entanto, devemos criar o pacote. Portanto, clique com o botão direito do mouse sobre o projeto **HibernateJM**, escolha *Novo > Pacote Java* e informe “**com.jm.acessobd**” como nome.

Listagem 4. Código da classe HibernateUtil.

```
01. package util;
02.
03. import org.hibernate.SessionFactory;
04. import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
05. import org.hibernate.cfg.Configuration;
06. import org.hibernate.service.ServiceRegistry;
07.
08. public class HibernateUtil {
09.
10.     private static SessionFactory sessionFactory;
11.
12.     public static SessionFactory getSessionFactory() {
13.         if (sessionFactory == null) {
14.
15.             Configuration configuration = new Configuration().configure();
16.             ServiceRegistry serviceRegistry
17.                 = new StandardServiceRegistryBuilder()
18.                     .applySettings(configuration.getProperties()).build();
19.
20.
21.             sessionFactory = configuration.buildSessionFactory(serviceRegistry);
22.         }
23.
24.         return sessionFactory;
25.     }
26. }
```

Depois de criado o pacote, clique sobre ele com o botão direito do mouse, escolha *Novo > Classe Java* e dê o nome à classe de **FuncionarioDAO**. O código fonte dessa classe deve ficar semelhante ao apresentado na **Listagem 5**.

Na linha 9, dentro do construtor da classe **FuncionarioDAO**, o método **getSessionFactory()** é invocado para a criação de um **SessionFactory**. Esse objeto deve ser criado uma única vez durante a execução da aplicação. Objetos desse tipo armazenam os mapeamentos e configurações do Hibernate e são muito pesados e lentos de se criar.

Já na linha 12, o método **adicionarFuncionario()** recebe como parâmetro os dados do funcionário e realiza a inserção deste no banco de dados. Para isso, na linha 13 é obtida uma sessão a partir de um **SessionFactory**. Um objeto **Session** pode ser considerado como uma sessão de comunicação com o banco de dados através de uma conexão JDBC. A operação **beginTransaction()**, presente na linha 17, inicia uma transação. Em seguida, o método **save()**, invocado na linha 18, realiza a persistência do objeto. Com isso, um registro é adicionado na tabela *funcionario* de acordo com os valores definidos no objeto. Por fim, na linha 19, o **commit** finaliza a transação e a sessão é fechada na linha 26.

DÊ UM SALTO EM CONHECIMENTO!



Acesse o maior
portal para
desenvolvedores
da América
Latina!

20
mil
posts

430
mil
cadastrados

10
milhões de
page-views
por mês

Além deste, outros três métodos foram criados, a saber: **apagarFuncionario()**, **atualizarFuncionario()** e **ListarFuncionarios()**. Basicamente, esses métodos possuem a mesma estrutura com relação ao método que já foi explicado. As diferenças encontram-se nas linhas 37, 61 e 79.

Na linha 37, o método **createCriteria()**, da classe **Session**, cria uma *query* passando como parâmetro a classe que vai ser pesquisada; no nosso caso, **Funcionario**. Por sua vez, o método **update()**, na linha 61, realiza a atualização do objeto passado como parâmetro, assim como o método **delete()**, chamado na linha 79, remove o funcionário passado como parâmetro da base de dados.

Testando a aplicação

Nesse momento, com toda a aplicação finalizada, podemos realizar alguns testes para avaliar se tudo está funcionando

conforme o esperado. Para tal, vamos criar apenas mais uma classe de teste.

Primeiramente, no entanto, devemos criar o pacote que armazenará essa classe. Sendo assim, clique com o botão direito do mouse sobre o projeto *HibernateJM*, escolha *Novo > Pacote Java* e informe “com.jm.teste” como nome.

Feito isso, clique com o botão direito do mouse sobre esse pacote, escolha *Novo > Classe Java* e dê o nome à classe de **TesteHibernate**. O código fonte dessa classe deve ficar igual ao da **Listagem 6**.

Na linha 7 a variável **funcionarioDao** armazena uma instância da classe **FuncionarioDAO**. Nessa classe, conforme demonstrado na **Listagem 5**, estão todas as operações de persistência que iremos realizar sobre o banco.

Em nosso teste, a primeira operação chamada foi a **adicionarFuncionario()**, que realiza a inserção de um funcionário no banco

Listagem 5. Classe DAO que contém todas as operações que serão realizadas no banco de dados.

```
01. package com.jm.acessobd;
02.
03. //imports omitidos
04.
05. public class FuncionarioDAO {
06.     private static SessionFactory factory;
07.
08.     public FuncionarioDAO() {
09.         factory = HibernateUtil.getSessionFactory();
10.     }
11.
12.     public Integer adicionarFuncionario(String nome, String cargo,
13.                                         double salario, String cpf, String endereco) {
14.         Session session = factory.openSession();
15.         Transaction tx = null;
16.         Integer cod_func = null;
17.         try {
18.             tx = session.beginTransaction(); //inicia a transacao
19.             cod_func = (Integer) session.save(new Funcionario(
20.                 nome, cargo, salario, cpf, endereco)); //salva na sessao o objeto
21.             tx.commit(); //executa o commit
22.         } catch (HibernateException e) {
23.             if (tx != null) {
24.                 tx.rollback();
25.             }
26.             e.printStackTrace();
27.         } finally {
28.             session.close();
29.         }
30.
31.         public void listarFuncionarios() {
32.             Session session = factory.openSession();
33.             Transaction tx = null;
34.             Funcionario funcionario = null;
35.             try {
36.                 tx = session.beginTransaction();
37.                 List funcionarios = session.createCriteria(Funcionario.class).list();
38.
39.                 for (Object obj : funcionarios) {
40.                     funcionario = (Funcionario) obj;
41.                     System.out.println(funcionario.toString());
42.                 }
43.                 tx.commit();
44.             } catch (HibernateException e) {
45.                 if (tx != null) {
```

```
46.                     tx.rollback();
47.                 }
48.                 e.printStackTrace();
49.             } finally {
50.                 session.close();
51.             }
52.         }
53.
54.         public void atualizarFuncionario(Integer codigoFuncionario, int salario) {
55.             Session session = factory.openSession();
56.             Transaction tx = null;
57.             try {
58.                 tx = session.beginTransaction();
59.                 Funcionario funcionario = (Funcionario) session.get(
60.                     Funcionario.class, codigoFuncionario);
61.                 funcionario.setSalario(salario);
62.                 session.update(funcionario);
63.                 tx.commit();
64.             } catch (HibernateException e) {
65.                 if (tx != null) {
66.                     tx.rollback();
67.                 }
68.                 e.printStackTrace();
69.             } finally {
70.                 session.close();
71.             }
72.
73.         public void apagarFuncionario(Integer codigoFuncionario) {
74.             Session session = factory.openSession();
75.             Transaction tx = null;
76.             try {
77.                 tx = session.beginTransaction();
78.                 Funcionario funcionario = (Funcionario) session.get(
79.                     Funcionario.class, codigoFuncionario);
80.                 session.delete(funcionario);
81.                 tx.commit();
82.             } catch (HibernateException e) {
83.                 if (tx != null) {
84.                     tx.rollback();
85.                 }
86.                 e.printStackTrace();
87.             } finally {
88.                 session.close();
89.             }
90.         }
```

de dados. Essa operação foi repetida quatro vezes, da linha 9 à linha 12. Assim, ao final da execução desse trecho de código e caso nenhum erro ocorra, teremos quatro funcionários inseridos no banco de dados.

Listagem 6. Código da classe de teste.

```
01. package com.jm.teste;
02.
03. public class TesteHibernate {
04.
05.     public static void main(String[] args) {
06.
07.         FuncionarioDAO funcionarioDao = new FuncionarioDAO();
08.
09.         Integer funcionario1 = funcionarioDao.adicionarFuncionario
("Clenio Rocha", "Analista de TI", 5600, "06493084765", "Rua Rio Paranaiba");
10.        Integer funcionario2 = funcionarioDao.adicionarFuncionario
("Carlos Martins", "Analista de TI", 5600, "02637848867",
"Rua Abadia dos dourados");
11.        Integer funcionario3 = funcionarioDao.adicionarFuncionario
("Yuko Rodrigues", "Gerente de Projeto", 70000, "07898378863",
"Rua Araxá");
12.        Integer funcionario4 = funcionarioDao.adicionarFuncionario
("Camila Rodrigues", "Esteticista", 3000, "05855578533",
"Rua do Garimpeiro");
13.
14.        funcionarioDao.listarFuncionarios();
15.        funcionarioDao.atualizarFuncionario(funcionario1, 5000);
16.        funcionarioDao.apagarFuncionario(funcionario1);
17.        funcionarioDao.listarFuncionarios();
18.    }
19. }
```

Em seguida, na linha 14, o método **listarFuncionarios()** é invocado. Esse método recupera e lista todos os funcionários cadastrados. Já o método **atualizarFuncionario()**, chamado na linha 15, atualiza o valor referente ao salário do funcionário passado como parâmetro. Em seguida, o método **apagarFuncionario()**, chamado na linha 16, remove da base de dados o funcionário passado como parâmetro. Por fim, chamamos novamente o método **listarFuncionarios()**, para verificar o estado dos objetos persistidos no banco.

Resultado da execução

A execução da classe **TesteHibernate** apresentará o resultado exposto na **Listagem 7** no console da IDE NetBeans.

Como a flag **show_log** está definida como **true** no arquivo de configurações, o console da IDE mostrará os comandos SQL executados pelo Hibernate. Note que nas quatro primeiras linhas são exibidos os *inserts* referentes à adição dos funcionários no banco de dados. Na linha 5, por sua vez, encontramos um *select*, sem a cláusula *where*, cujo objetivo é obter todos os registros (funcionários) cadastrados. Em seguida, todos esses registros são listados. Já na linha 11, é realizada uma atualização através do comando *update*. Logo após, na linha 13, o comando *delete* sinaliza que um funcionário foi removido da base de dados. Por fim, nas linhas

seguintes, temos um *select* para retornar e apresentar novamente todos os funcionários presentes na base.

Listagem 7. Saída da execução da classe TesteHibernate.

```
01. Hibernate: insert into Funcionario (cargo, cpf, endereco, nome, salario)
values (?, ?, ?, ?, ?)
02. Hibernate: insert into Funcionario (cargo, cpf, endereco, nome, salario)
values (?, ?, ?, ?, ?)
03. Hibernate: insert into Funcionario (cargo, cpf, endereco, nome, salario)
values (?, ?, ?, ?, ?)
04. Hibernate: insert into Funcionario (cargo, cpf, endereco, nome, salario)
values (?, ?, ?, ?, ?)
05. Hibernate: select this_.codigo as codigo1_0_0_, this_.cargo as cargo2_0_0_,
this_.cpf as cpf3_0_0_, this_.endereco as endereco4_0_0_, this_.nome as
nome5_0_0_, this_.salario as salario6_0_0_ from Funcionario this_
06. Funcionario [codigo=58, nome=Clenio Rocha, endereco=Rua Rio Paranaiba,
cpf=06493084765, cargo=Analista de TI, salario=5600.0]
07. Funcionario [codigo=59, nome=Carlos Martins, endereco=Rua Abadia dos
dourados, cpf=02637848867, cargo=Analista de TI, salario=5600.0]
08. Funcionario [codigo=60, nome=Yuko Rodrigues, endereco=Rua Araxá,
cpf=07898378863, cargo=Gerente de Projeto, salario=70000.0]
09. Funcionario [codigo=61, nome=Camila Rodrigues, endereco=Rua do
Garimpeiro, cpf=05855578533, cargo=Esteticista, salario=3000.0]
10. Hibernate: select funcionari0_.codigo as codigo1_0_0_, funcionari0_.
cargo as cargo2_0_0_, funcionari0_.cpf as cpf3_0_0_, funcionari0_.endereco
as endereco4_0_0_, funcionari0_.nome as nome5_0_0_, funcionari0_.salario
as salario6_0_0_ from Funcionario funcionari0_ where funcionari0_.codigo=?
11. Hibernate: update Funcionario set cargo=?, cpf=?, endereco=?, nome=?,
salario=? where codigo=?
12. Hibernate: select funcionari0_.codigo as codigo1_0_0_, funcionari0_.
cargo as cargo2_0_0_, funcionari0_.cpf as cpf3_0_0_, funcionari0_.endereco
as endereco4_0_0_, funcionari0_.nome as nome5_0_0_, funcionari0_.salario
as salario6_0_0_ from Funcionario funcionari0_ where funcionari0_.codigo=?
13. Hibernate: delete from Funcionario where codigo=?
14. Hibernate: select this_.codigo as codigo1_0_0_, this_.cargo as cargo2_0_0_,
this_.cpf as cpf3_0_0_, this_.endereco as endereco4_0_0_, this_.nome as
nome5_0_0_, this_.salario as salario6_0_0_ from Funcionario this_
15. Funcionario [codigo=58, nome=Clenio Rocha, endereco=Rua Rio Paranaiba,
cpf=06493084765, cargo=Analista de TI, salario=5000.0]
16. Funcionario [codigo=60, nome=Yuko Rodrigues, endereco=Rua Araxá,
cpf=07898378863, cargo=Gerente de Projeto, salario=70000.0]
17. Funcionario [codigo=61, nome=Camila Rodrigues, endereco=Rua do
Garimpeiro, cpf=05855578533, cargo=Esteticista, salario=3000.0]
```

Para confirmar todos esses passos, vamos realizar mais um *select*, mas dessa vez diretamente no terminal do MySQL. O resultado dessa execução deverá ser semelhante ao conteúdo exposto na **Figura 15**.

Além de ser um ótimo framework open source para realização do mapeamento objeto relacional, o Hibernate também é a solução mais adotada atualmente. A estabilidade em que se encontra o projeto e as facilidades que ele oferece ao programador estão entre suas principais características.

Assim, o Hibernate faz com que a camada de persistência seja desenvolvida com muita liberdade, tanto por dar mais legibilidade ao código, pois o desenvolvedor abstrai o conceito de tabelas do banco de dados, quanto pela portabilidade oferecida pelos dialetos de banco de dados, que abstraem as particularidades de cada SGBD.

The screenshot shows a MySQL command-line interface window titled "MySQL 5.6 Command Line Client - Unicode". It displays the following text and a table:

```
Enter password: ****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 29
Server version: 5.6.21-Log MySQL Community Server (GPL)

Copyright (c) 2000, 2014, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type 'c' to clear the current input statement.

mysql> use empresa;
Database changed
mysql> select * from funcionario;
+----+-----+-----+-----+-----+
| CODIGO | NOME | CARGO | SALARIO | CPF |
+----+-----+-----+-----+-----+
| S1 | Carlos Martins | Analista de TI | 3600 | 02637848867 |
| S2 | Hugo Rodrigues | Gerente do Projeto | 70000 | 07898378863 |
| S3 | Camila Rodrigues | Esteticista | 3000 | 0565578533 |
+----+-----+-----+-----+-----+
3 rows in set (0.00 sec)

mysql> -
```

CODIGO	NOME	CARGO	SALARIO	CPF
S1	Carlos Martins	Analista de TI	3600	02637848867
S2	Hugo Rodrigues	Gerente do Projeto	70000	07898378863
S3	Camila Rodrigues	Esteticista	3000	0565578533

Figura 15. Visão do banco de dados após a execução da classe de teste

Enfim, podemos afirmar que o framework desempenha muito bem o que lhe é incumbido, que é viabilizar a persistência de objetos em bases de dados relacionais, e preenche com sobras a lacuna existente entre as estruturas relacionais e a programação orientada a objetos.

Autor



Carlos Alberto Silva

casilvamg@hotmail.com



É formado em Ciéncia da Computação pela Universidade Federal de Uberlândia (UFU), com especialização em Desenvolvimento Java pelo Centro Universitário do Triângulo (UNITRI). Trabalha na empresa Algar Telecom como Analista de TI e atualmente é aluno do curso de especialização em Análise e Desenvolvimento de Sistemas Aplicados a Gestão Empresarial no Instituto Federal do Triângulo Mineiro (IFTM). Possui as seguintes certificações: OCJP, OCWCD e ITIL.

Autor



Eduardo Augusto Silvestre

eduardosilvestre@iftm.edu.br



É formado em Ciéncia da Computação pela Universidade Federal de Uberlândia (UFU), especialista em Governança de TI pela Universidade Federal de Lavras (UFLA), mestre em Ciéncia da Computação pela Universidade Federal de Uberlândia (UFU) e doutorando em Computação pela Universidade Federal Fluminense (UFF). Tem experiência na área de Ciéncia da Computação, com ênfase em Engenharia de Software, Desenvolvimento de Aplicações para Internet e Sistemas Multiagentes. Atualmente é professor do Instituto Federal do Triângulo Mineiro (IFTM).

Links:

Site Oficial do Hibernate.

<http://hibernate.org/>

Site Oficial do MySQL.

<http://www.mysql.com/>

Site driver do MySQL.

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

Site Oficial do NetBeans.

<https://netbeans.org>

Site para download do JDK.

<http://www.oracle.com/technetwork/java/javase/downloads>

Processo de instalação do NetBeans.

https://netbeans.org/community/releases/80/install_pt_BR.html

Processo de instalação do JDK.

<http://www.oracle.com/technetwork/java/javase/downloads/jdk-7-netbeans-install-pt-br-433855.html>

Christian Bauer, Gavin King. Livro Java Persistence com Hibernate, 2005.

Somos tão apaixonados por tecnologia que o nome da empresa diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,
assim como você.



Estrutura

100% NACIONAL.
Servidores de primeira linha, links de alta capacidade.



Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.



Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.



1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única.
Conheça!



Porta 80

WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce |
Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486