



Edição 37

DESAFIO: Comprendendo o Big Data
Uma introdução aos seus
conceitos e principais tecnologias

Compactando dados com Java

Aprenda a compactar e
descompactar arquivos e objetos

Novidades do Groovy 2.2

Pequenas melhorias para
aumentar a produtividade

PROGRAMANDO COM GRAILS

Crie aplicações web em minutos

A standard linear barcode is located in the bottom left corner of the page.

ISSN 2179625-4



00037



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**.

CERTIFICAÇÕES:

➤ Oracle Certified Professional, Java SE 6 Programmer

➤ Oracle Certified Professional, Java EE 5 Web Component Developer



Sumário

Conteúdo sobre Boas Práticas

06 – Compressão de dados em Java

[John Soldera]

Artigo no estilo Solução Completa

13 – Grails Framework: Criando aplicações web com Grails

[Alessandro Jatobá]

Conteúdo sobre Novidades

21 – Novidades do Groovy 2.2

[Henrique Lobo Weissmann]

Conteúdo sobre Boas Práticas

26 – Analisando o Big Data na teoria e na prática

[Diego Travassos Balduini]

Aplicações Java

Na Impacta você tem a formação mais completa na linguagem mais relevante do mercado*



Treinamentos em: JEE - JavaServer Pages + Servlets • JAVA - Design Patterns • JAVA - Hibernate • JAVA Spring Framework • JEE - Apache Struts • JEE - Enterprise • JAVABeans • JEE - Java Web Services • JME - Aplicações para dispositivos móveis • JSF - JavaServer Faces • Java Programmer • Android

* Segundo o índice TIOBE, que considera a posição de cada linguagem de programação nos buscadores.



Conheça também a Pós-Graduação em Engenharia de Software na Faculdade Impacta

- Os melhores professores e instrutores do mercado;
- A melhor Infraestrutura;
- A melhor metodologia;



IMPACTA
CERTIFICAÇÃO
E TREINAMENTO

Ligue (11) 3254 2200 ou acesse www.impacta.com.br



blogimpacta.com.br



@grupoimpacta



/grupoimpacta



Edição 37 • 2014 • 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 Davi Costa (davigc_08@hotmail.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

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



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

Compressão de dados em Java

Aprenda a compactar arquivos e objetos de forma simples e rápida

Algumas aplicações podem gerar um volume muito grande de dados a serem gravados no sistema de arquivos, levando à necessidade cada vez maior de espaço de armazenamento. Um problema semelhante ocorre em aplicações que transmitem grandes quantidades de dados por uma *stream*, podendo causar a demanda de uma conexão com maior largura de banda.

A fim de oferecer suporte à linguagem Java para compactar dados de forma simples e rápida, sem demandar integração com aplicações de terceiros, o pacote `java.util.zip` é oferecido desde o JDK 1.1, onde é proposta uma API com comandos capazes de compactar e descompactar arquivos e objetos. A API `java.util.zip` suporta os formatos ZIP e GZIP, possibilita a verificação de *checksums*, além de disponibilizar outros recursos que são acessados de forma simples e eficaz. A seguir são listadas as principais classes desse pacote:

- **ZipEntry:** É usada para representar arquivos ou diretórios compactados (entradas) em um arquivo ZIP;
- **ZipFile:** É usada para abrir arquivos ZIP e acessar as suas entradas;
- **ZipInputStream e ZipOutputStream:** Stream para ler e gravar dados em arquivos compactados no formato ZIP;
- **Deflater e Inflater:** Oferece recursos de uso geral para compactar e descompactar arquivos usando a popular biblioteca ZLIB;
- **DeflaterInputStream, DeflaterOutputStream, InflaterInputStream, InflaterOutputStream:** Permite compactar e descompactar arquivos usando o algoritmo “deflate”;
- **GZIPInputStream e GZIPOutputStream:** Permite ler e criar arquivos compactados no formato GZIP;
- **CRC32:** Classe usada para calcular o *checksum* CRC-32 de uma *stream* de dados.

Utilizando as classes do pacote `java.util.zip` é possível acessar de forma simples e rápida os recursos de

Fique por dentro

Este artigo apresenta as principais classes do pacote `java.util.zip`, que oferece uma API de acesso a funcionalidades de compactação de arquivos ou objetos em Java. Nesse artigo, as principais classes desse pacote serão comentadas e através de exemplos de aplicações será mostrado como ler as entradas de um arquivo ZIP, como compactar todos os arquivos de um diretório e como compactar e descompactar objetos. Além disso, será ensinado como acionar os comandos do shell para compilar e executar aplicações.

compactação e descompactação de arquivos e objetos. Além disso, são oferecidos vários formatos de compactação, que são adequados para a grande maioria dos casos.

Com base nisso, a seguir serão analisadas algumas das classes do pacote `java.util.zip` através de exemplos para ler arquivos compactados, gravar arquivos compactados e compactar e descompactar objetos em memória em Java.

Entradas de um arquivo ZIP: a classe ZipEntry

Na linguagem Java, um arquivo ZIP é representado como uma lista de entradas. Uma entrada pode ser um arquivo ou um diretório, que por sua vez também é uma lista de entradas. Para representar uma entrada, sendo arquivo ou diretório, é usado um objeto da classe `ZipEntry`.

O construtor `ZipEntry()` recebe como parâmetro o nome da entrada, ou seja, o nome completo do diretório ou arquivo compactado (entrada) pertencente ao arquivo ZIP. Usando os métodos `getComment()` e `setComment()` é possível recuperar ou redefinir o comentário que descreve essa entrada. Os métodos `getCompressedSize()` e `setCompressedSize()` permitem acessar o tamanho compactado da entrada. Já os métodos `getSize()` e `setSize()` possibilitam acessar o tamanho original da entrada. Os métodos `getCRC()` e `setCRC()`, por sua vez, possibilitam acessar o *checksum* CRC-32 da entrada. Por fim, `getName()` retorna o nome da entrada e `isDirectory()` permite verificar se ela é um diretório.

Lendo arquivos compactados: a classe ZipFile

A classe **ZipFile** oferece recursos básicos para a leitura de arquivos ZIP. Com ela, é possível abrir um arquivo ZIP e acessar as suas entradas. Para isso, são oferecidos construtores sobrecarregados para informar o nome do arquivo ZIP e o *charset* (se necessário), que é o padrão de codificação de nomes de entradas e demais descrições textuais.

No construtor **ZipFile(String name)**, é informado o nome do arquivo ZIP a ser aberto. Através do método **getComment()**, é possível acessar os comentários do arquivo ZIP. O método **getEntry()**, por sua vez, retorna uma entrada no arquivo compactado, dado o nome da mesma. Já **getInputStream()** retorna uma *stream* para ler os dados de uma determinada entrada no arquivo ZIP. O método **getName()** retorna o nome do arquivo ZIP. O método **size()** retorna a quantidade de entradas no arquivo compactado. O método **entries()** retorna a lista das entradas presentes no arquivo compactado na forma de uma **Enumeration**. Para fechar o arquivo ZIP, é usado o método **close()**, e para liberar os recursos do sistema operacional utilizados por esse objeto, é usado o **finalize()**.

Na **Listagem 1** é apresentado um exemplo simples de uso da classe **ZipFile**. Neste código, o objetivo é ler um arquivo ZIP e listar o nome de suas entradas que forem arquivos.

Esse código fonte declara a classe **ZipFileReadTest**, criada no pacote **zipFilePkg** (linha 1). As classes referenciadas são importadas nas linhas 3 a 8, e a classe **ZipFileReadTest** é declarada na linha 10. No início do processamento, o método **main()** (linhas 12-24) verifica se apenas um argumento foi informado pela linha de comando. Se sim, o nome do arquivo ZIP a ser lido será o argumento de índice zero (linha 14). Em seguida, na linha 16, o método **getEntryNames()** é chamado (linha 26) e realiza a leitura do arquivo ZIP retornando uma lista com os nomes das entradas, que, a seguir, é impressa (linha 18).

O método **getEntryNames()** recebe como argumento o nome do arquivo ZIP a ser aberto, e na linha 30, cria um objeto **ZipFile** para o arquivo ZIP informado. Através do método **entries()** (linha 31), é possível obter a lista de entradas na forma de uma **Enumeration**, que é percorrida com o método **nextElement()** (linha 33) até **hasMoreElements()** (linha 32) retornar **false**. Entradas que sejam diretórios são ignoradas (linhas 34-36).

Portanto, através desse exemplo, podem-se ler as entradas de um arquivo ZIP que não sejam diretórios. A fim de executar o programa e ver os seus resultados, os comandos apresentados na **Listagem 2** podem ser digitados na linha de comando do shell do sistema operacional para compilar e executar o programa exposto na **Listagem 1**, desde que o JDK esteja instalado e os arquivos **java.exe** e **javac.exe** estejam no caminho de pesquisa de arquivos executáveis do sistema operacional, ou seja, na variável PATH do sistema operacional.

Os comandos indicados na **Listagem 2** devem ser executados no shell na mesma ordem em que foram expostos. Primeiro, o comando **dir** é executado para posicionar o shell no diretório logo acima do pacote **zipFilePkg**, que também é um diretório.

Listagem 1. Exemplo de Programa para leitura de um arquivo ZIP com ZipFile.

```
01. package zipFilePkg;
02.
03. import java.io.IOException;
04. import java.util.ArrayList;
05. import java.util.Enumeration;
06. import java.util.List;
07. import java.util.zip.ZipEntry;
08. import java.util.zip.ZipFile;
09.
10. public class ZipFileReadTest {
11.
12.     public static void main(String[] args) {
13.         if (args.length == 1) {
14.             String fileName = args[0];
15.             try {
16.                 List<String> nameList = getEntryNames(fileName);
17.                 for (String entryName : nameList) {
18.                     System.out.println(entryName);
19.                 }
20.             } catch (IOException e) {
21.                 e.printStackTrace();
22.             }
23.         }
24.     }
25.
26.     public static List<String> getEntryNames(String zipFilename)
27.             throws IOException {
28.         ZipFile zipFile = null;
29.         try {
30.             List<String> list = new ArrayList<String>();
31.             zipFile = new ZipFile(zipFilename);
32.             Enumeration<? extends ZipEntry> entries = zipFile.entries();
33.             ZipEntry zipEntry = entries.nextElement();
34.             if (zipEntry.isDirectory())
35.                 continue;
36.             list.add(zipEntry.getName());
37.         }
38.         return list;
39.     } finally {
40.         zipFile.close();
41.     }
42. }
43. }
```

Listagem 2. Comandos para execução do programa ZipFileReadTest.

```
dir <diretório logo acima do diretório zipFilePkg>
javac zipFilePkg\ZipFileReadTest.java
java zipFilePkg.ZipFileReadTest <parâmetros>
```

Uma vez que a estrutura de diretórios seja respeitada, o comando **javac** é usado para compilar classes (no caso, a classe **ZipFileReadTest**). Esse comando recebe como parâmetro o caminho relativo da classe a ser compilada e cria o arquivo binário **ZipFileReadTest.class** no mesmo diretório de **ZipFileReadTest.java**.

Por fim, com o comando **java**, pode-se executar o *bytecode* gerado no passo anterior (**ZipFileReadTest.class**). Em **<parâmetros>** são informados os parâmetros ao programa **ZipFileReadTest**. Por exemplo, para ler o arquivo **c:\teste.zip**, o mesmo comando é escrito da seguinte forma: **java zipFilePkg.ZipFileReadTest c:\teste.zip**.

Compressão de dados em Java

O parâmetro informado ao programa **ZipFileReadTest** é repassado ao método **main()**, preenchendo a variável **args** (linha 12 da **Listagem 2**), e em seguida, a execução do programa é iniciada. dessa forma, é possível ler as entradas de um arquivo ZIP, porém não é possível adicionar ou alterar entradas no mesmo. Para criar arquivos compactados, a classe **ZipOutputStream** pode ser usada.

Criando arquivos compactados: a classe ZipOutputStream

A classe **ZipOutputStream** também é encontrada no pacote **java.util.zip**, e disponibiliza os recursos necessários para a criação de arquivos compactados, o que inclui métodos simples para adicionar arquivos e diretórios (entradas) nesses arquivos ZIP. Essa classe oferece construtores sobrecarregados, pelos quais pode se informar uma *stream* para o arquivo ZIP aberto e o parâmetro opcional: o *charset*, que é o padrão de codificação de nomes de entradas e comentários no arquivo ZIP.

A classe **ZipOutputStream** disponibiliza diversos métodos, como é o caso do método **setComment()**, que permite definir um comentário para o arquivo ZIP. Já o método **putNextEntry()** possibilita adicionar uma nova entrada no arquivo ZIP. O método **setLevel()** permite definir o método de compressão, sendo

o *default* dado pela constante **DEFLATED** (compactado), e para criar um arquivo ZIP sem compactação, pode-se usar a constante **STORED**. Por fim, o método **write()** possibilita escrever vetores de *bytes* no arquivo.

Uma vez que todos os dados de uma entrada tenham sido gravados no arquivo ZIP, o método **closeEntry()** fecha a entrada corrente e o método **close()** pode ser usado para fechar a *stream* do arquivo ZIP. Por sua vez, o método **finish()** completa qualquer operação de escrita pendente mantendo a *stream* aberta.

A **Listagem 3** apresenta um exemplo de programa utilizando os métodos da classe **ZipOutputStream** que foram analisados. O objetivo desse programa é receber um diretório pela linha de comando do shell do sistema operacional e compactar todos os arquivos desse diretório. Para isso, ele cria uma *stream* **ZipOutputStream** informando no construtor uma **FileOutputStream** para o diretório *e*, em seguida, lista todos os arquivos desse diretório, criando no arquivo compactado uma entrada para cada arquivo. Ao final, as *streams* das classes **ZipOutputStream** e **FileOutputStream** são fechadas e o arquivo compactado é salvo.

A organização do código fonte da **Listagem 3** segue o mesmo padrão. Portanto, note que o pacote é declarado na linha 1 e os

Listagem 3. Exemplo de programa com **ZipOutputStream**.

```
01. package zipFilePkg;
02.
03. import java.io.File;
04. import java.io.FileInputStream;
05. import java.io.FileOutputStream;
06. import java.io.IOException;
07. import java.util.zip.ZipEntry;
08. import java.util.zip.ZipOutputStream;
09.
10. public class ZipCompact {
11.
12.     String zipFilename;
13.     String directory;
14.
15.     public ZipCompact(String zipFilename, String directory) {
16.
17.         this.zipFilename = zipFilename;
18.         this.directory = directory;
19.     }
20.
21.     public static void main(String[] args) {
22.         try {
23.             if (args.length != 2) {
24.                 System.out.println("Primeiro parâm: nome do arq. ZIP, segundo parâm: destino.");
25.                 return;
26.             }
27.             else {
28.                 System.out.println("Compactando arquivos...");
29.                 ZipCompact zipApp = new ZipCompact(args[0], args[1]);
30.                 zipApp.compress();
31.                 System.out.println("Feito.");
32.             }
33.         }
34.         catch (IOException e) {
35.             e.printStackTrace();
36.         }
37.     }
38.
39.     private boolean compress() throws IOException {
```



```
40.         FileOutputStream fout = null;
41.         ZipOutputStream zout = null;
42.         try {
43.             fout = new FileOutputStream(zipFilename);
44.             zout = new ZipOutputStream(fout);
45.             File dir = new File(directory);
46.             boolean isDirectory = dir.isDirectory();
47.             if (isDirectory) {
48.                 File[] files = dir.listFiles();
49.                 for (File file : files) {
50.                     if (file.isFile()) {
51.                         String name = file.getName();
52.                         ZipEntry zipEntry = new ZipEntry(name);
53.                         zout.putNextEntry(zipEntry);
54.                         FileInputStream sout = new FileInputStream(file);
55.                         byte[] buffer = new byte[1024];
56.                         int length;
57.                         while ((length = sout.read(buffer)) > 0) {
58.                             zout.write(buffer, 0, length);
59.                         }
60.                         sout.closeEntry();
61.                         sout.close();
62.                     } else {
63.                         System.out.println("Desconsiderando: " + file.getName());
64.                         return false;
65.                     }
66.                 }
67.             }
68.             else {
69.                 System.out.println("Segundo parâmetro deve ser um diretório.");
70.                 return false;
71.             }
72.         }
73.         finally {
74.             zout.close();
75.             fout.close();
76.         }
77.         return true;
78.     }
79. }
```

imports da linha 3 até a 8. A classe **ZipCompact** (linhas 10 a 79) contém dois atributos, um numérico e uma **String** (linhas 12 e 13), que são informados no construtor (linhas 15 a 19). O método **main()** inicia na linha 21 e termina na linha 78, e recebe os parâmetros da linha de comando.

Logo no início do método **main()**, é verificado se exatamente dois parâmetros foram informados (linha 23). O primeiro parâmetro é o diretório a ser compactado e o segundo parâmetro é o nome completo do arquivo ZIP a ser criado. Depois, é criado um objeto da classe **ZipCompact** (linha 29) e o método **compress()** é chamado para compactar os arquivos do diretório (linha 30).

O método **compress()** (linha 39), inicialmente cria uma *stream* para o diretório informado que é repassada ao construtor de **ZipOutputStream** (linhas 43 e 44). Em seguida, é verificado se o primeiro parâmetro informado é realmente um diretório (linha 46). Se confirmado, é usado o método **listFiles()** (linha 48) da classe **File** para listar os arquivos do diretório a compactar. Para cada arquivo do diretório, é criada uma entrada (linhas 49-66), que é um objeto da classe **ZipEntry** (linha 52) e, logo após, o método **putNextEntry()** (linha 53) da classe **ZipOutputStream** é usado para vincular a nova entrada ao arquivo ZIP, usando o nome do arquivo a compactar como nome da entrada.

Em seguida, é necessário gravar os dados da entrada no arquivo ZIP. Para isso, o arquivo é lido usando uma **FileInputStream** (linha 54) e gravado com o método **write()** (linhas 57-59) da classe **ZipOutputStream**. Feito isso, a entrada é fechada e também a *stream* dessa entrada (linhas 60 e 61). O programa processará todas as entradas da mesma forma, fechando logo após todas as *streams* restantes (linhas 74 e 75).

Uma observação importante é que esse programa não compacta subdiretórios. É possível adicionar essa funcionalidade facilmente se considerarmos a possibilidade de definir uma estrutura recursiva para o código. Uma forma de fazer isso é criar um método novo que recebe como entrada um diretório e uma *stream* para um arquivo ZIP. Esse método tem a funcionalidade de listar o diretório informado e, a cada vez que encontrar um arquivo, ele cria uma entrada no arquivo ZIP informado, ou se encontrar um diretório, ele faz uma chamada a si mesmo recursivamente, dessa forma percorrendo toda a hierarquia de diretórios de entrada até que todos os arquivos internos sejam lidos e compactados. Como cada arquivo da hierarquia de diretórios está vinculado a um caminho específico, é importante preservar a estrutura de diretórios de cada entrada ao inseri-las no arquivo ZIP. Para isso, pode-se adicionar a informação de diretório como prefixo no nome de cada entrada.

Portanto, através desse exemplo, pode-se ler um diretório e compactar todos os arquivos, gerando como resultado um arquivo ZIP. Para rodar esse programa (**Listagem 3**), execute os comandos expostos na **Listagem 4**.

Como já informado no exemplo anterior, **dir** é necessário para posicionar o shell no diretório que contém o pacote **zipFilePkg**, e, no próximo comando, **javac** realiza a compilação da classe **ZipCompact**. Observe que **javac** recebe como parâmetro o

caminho completo da classe e vai criar o *bytecode ZipCompact.class* no mesmo diretório de **ZipCompact.java**. Por fim, com o comando **java** pode-se executar o arquivo binário **.class**. Deste modo o programa **ZipCompact** é executado, recebendo dois parâmetros separados por espaço, o diretório a ser compactado e o nome do arquivo ZIP a ser criado, que são informados pelo usuário.

Listagem 4. Comandos para execução do programa ZipCompact.

```
dir <diretório logo acima do diretório zipFilePkg>
javac zipFilePkg/ZipCompact.java
java zipFilePkg.ZipCompact <diretorio> <nome do arquivo ZIP>
```

Compactando objetos em Java

Nos exemplos anteriores, foi exposto como ler e criar arquivos compactados em Java. Porém, também são oferecidos recursos para compactar objetos em memória, de forma que os bytes obtidos como resultado da compactação possam ser transmitidos facilmente por uma *stream* de dados. Para o processo inverso, isto é, receber os dados, também são oferecidos recursos para a descompactação dos mesmos, onde vetores de bytes são convertidos novamente à forma original (objetos da classe a que eles pertencem).

**Não perca tempo
reinventando a roda!**

COBREBEMX

**Componente completo para sua
Cobrança por Boleto Bancário
e Débito em Conta Corrente**

**Mais de 40 exemplos
em diversas linguagens
de programação**

**Geração e leitura de arquivos
(remessa e retorno) nos padrões
FEBRABAN e CNAB**

**Testes e Downloads
gratuitos em nosso site**

**ACESSE E CONHEÇA O COMPONENTE EM:
WWW.COBREBEM.COM**

Compressão de dados em Java

Para exemplificar esses recursos, na **Listagem 5** são criados dois objetos da classe **ZipObject**, e em seguida eles são compactados na forma de *array of bytes*, descompactados e convertidos à classe original. Por fim, os objetos originais são comparados com os objetos recuperados para verificar se eles apresentam os mesmos valores.

Observe que a classe **ZipObject** (linha 11) contém dois atributos, um inteiro e uma **String** (linhas 14 e 15), que servem como exemplo de atributos que devem ficar inalterados mesmo depois dos objetos que os contêm terem sido compactados e descompactados. Na declaração da classe **ZipObject**, o construtor é exposto nas linhas 17 a 20, onde são informados um número inteiro e uma **String** como parâmetros, a fim de que sejam atributos do objeto sendo construído. Além disso, é importante notar que a classe **ZipObject** implementa a interface **java.io.Serializable** (linha 11), permitindo assim que um objeto dessa classe possa ser recebido por uma *stream* da classe **ObjectOutputStream**.

O método **main()**, declarado entre linhas 22 e 28, realiza a execução do exemplo proposto. Ele chama o método **doTest()** e, se ocorrer algum erro durante a sua execução, este erro é apresentado na tela. Este método, que inicia na linha 37, instancia os objetos a serem gravados (linhas 38 e 39) e, em seguida, são criadas as streams **ByteArrayInputStream**, **GZIPInputStream** e **ObjectInputStream** (linhas 40 a 42). A partir da relação entre elas, ao invocar o método **writeObject()** de **ObjectInputStream**, os objetos informados por parâmetro são compactados (linhas 43 e 44).

A partir da linha 46 é iniciado o processo inverso, quando ocorre a conversão da stream final da etapa anterior em um array de bytes. A partir dele, da linha 47 à linha 52 recuperamos novamente os objetos, através da descompactação, e depois os convertemos novamente a objetos da classe **ZipObject**, com o uso de *casting* (linhas 50 e 51). Feito isso, na linha 53, os objetos originais são comparados com os objetos recuperados para verificar se eles contêm os mesmos valores nos atributos. Para isso, foi utilizado o método **compareTo()**, que é definido na interface **Comparable**. Nesse exemplo, é exibido na tela: “Objetos Iguais: true”.

Com esse exemplo, pôde-se conferir que objetos compactados representados na forma de *array of bytes* correspondem a uma representação compacta dos objetos originais e estão prontos para serem transmitidos por uma *stream*. Esse exemplo pode servir de guia para o caso da necessidade de criar aplicações nas quais é útil compactar objetos para diminuir a utilização da rede ao adotar *streams*. Para isso, ressalta-se mais uma vez que a classe do objeto a ser compactado deve implementar a interface **java.io.Serializable**.

Usando o compilador javac

Para compilar as classes Java, usamos o comando **javac**, que corresponde ao programa **javac.exe** do JDK. O **javac** compila arquivos de código fonte com extensão **.java** e gera arquivos de **bytecode** com extensão **.class**.

Em um projeto de desenvolvimento de *software Java*, podem ser criadas muitas classes e, de acordo com as necessidades do

Listagem 5. Exemplo de programa para compactar e descompactar objetos.

```
01. package zipFilePkg;
02.
03. import java.io.ByteArrayInputStream;
04. import java.io.ByteArrayOutputStream;
05. import java.io.IOException;
06. import java.io.ObjectInputStream;
07. import java.io.ObjectOutputStream;
08. import java.util.zip.GZIPOutputStream;
09. import java.util.zip.GZIPInputStream;
10.
11. public class ZipObject implements java.io.Serializable, Comparable<ZipObject> {
12.
13.     private static final long serialVersionUID = 5345860100202700611L;
14.     protected int a;
15.     protected String b;
16.
17.     public ZipObject (int a, String b) {
18.         this.a = a;
19.         this.b = b;
20.     }
21.
22.     public static void main(String[] args) {
23.         try {
24.             doTest();
25.         } catch (Exception e) {
26.             e.printStackTrace();
27.         }
28.     }
29.
30.     public int compareTo(ZipObject o) {
31.         if (o.a == a && o.b.equals(b))
32.             return 1;
33.         else
34.             return 0;
35.     }
36.
37.     public static void doTest() throws IOException, ClassNotFoundException {
38.         ZipObject zipObject1 = new ZipObject(10, "Objeto1");
39.         ZipObject zipObject2 = new ZipObject(44, "Objeto2");
40.         ByteArrayOutputStream byteOutputStream = new ByteArrayOutputStream();
41.         GZIPOutputStream zipOutputStream = new GZIPOutputStream(
42.             byteOutputStream);
43.         ObjectOutputStream objOutputStream = new ObjectOutputStream(
44.             zipOutputStream);
45.         objOutputStream.writeObject(zipObject1);
46.         objOutputStream.writeObject(zipObject2);
47.         objOutputStream.close();
48.         byte[] bytes = byteOutputStream.toByteArray();
49.         ByteArrayInputStream byteInputStream = new ByteArrayInputStream(bytes);
50.         GZIPInputStream zipInputStream = new GZIPInputStream(byteInputStream);
51.         ObjectInputStream objInputStream = new ObjectInputStream(zipInputStream);
52.         ZipObject zipObject3 = (ZipObject) objInputStream.readObject();
53.         ZipObject zipObject4 = (ZipObject) objInputStream.readObject();
54.         objInputStream.close();
55.         System.out.println("Objetos Iguais: " + (zipObject1.compareTo(zipObject3)
56.             ==1 && zipObject2.compareTo(zipObject4)==1));
57.     }
58. }
```

projeto, algumas opções do comando *javac* podem ser utilizadas para ajustar o processo de compilação dessas classes.

O comando *javac* tem a seguinte forma geral:

```
javac <opções> <classes>
```

Onde, em *<opções>* podem ser informados parâmetros para o compilador e em *<classes>* são informados todos os arquivos Java a serem compilados separados por espaço.

Em projetos de grande porte, é útil separar os arquivos *.class* dos arquivos *.java*, a fim de obter uma melhor organização do código fonte e do código binário, o que é recomendado em muitas situações, como, por exemplo, em projetos nos quais se usa uma ferramenta de controle de versão. Para viabilizar essa separação, pode-se utilizar a opção *-d* seguida do nome do diretório para gerar os arquivos binários, conforme o comando:

```
javac -d <diretório> <classes>
```

Dessa forma, todo o código binário será gravado em uma pasta separada informada por: *<diretório>*, organizando melhor o projeto, como no exemplo:

```
javac -d c:\bin zipFilePkg\ZipFileReadTest.java
```

Além disso, projetos grandes podem conter uma grande quantidade de bibliotecas e coleções de códigos fontes que precisam ser acessadas ao compilar o projeto Java. Para referenciar dependências do projeto de forma simples, a opção *-classpath* (ou simplesmente, *-cp*) pode ser utilizada. Veja este exemplo:

```
javac -classpath <diretório das dependências> <classes>
```

Outra opção, que é útil em projetos com equipes distribuídas em diversos países, nos quais pessoas de diferentes nacionalidades manipulam os mesmos arquivos de código fonte, é a *-encoding*, que permite ao compilador usar o padrão de codificação de uma determinada localidade para ler **Strings** e demais textos nos arquivos de código fonte. A sintaxe para uso dessa opção é exibida no comando:

```
javac -encoding <nome do encoding> <classes>
```

Além destas, existem mais opções que podem ser empregadas no compilador *javac*, como suprimir avisos de compilação (*warnings*) ou escolher uma versão específica da JVM. Depois de o código fonte ter sido compilado com sucesso, pode-se executar os arquivos de *bytecodes* gerados. Para isso, basta utilizar o comando: *java*.

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

Compressão de dados em Java

Usando o interpretador Java

Como Java é uma linguagem interpretada, executar um programa é o mesmo que interpretar o seu código binário obtido com o comando *javac*. Para realizar a tarefa de interpretação, é utilizado o comando *java*, que corresponde ao aplicativo *java.exe* localizado no JDK.

Para customizar o processo de interpretação do código binário, algumas opções são disponibilizadas no interpretador *java*, como a opção *-classpath*, que informa as dependências necessárias para a execução do programa. Um exemplo de uso deste recurso é apresentado a seguir:

```
java -classpath <diretório das dependências> <classes>
```

Existem outros fatores que podem influenciar a interpretação. Por exemplo, em alguns projetos é necessário habilitar ou desabilitar *assertions*, que são afirmações adicionadas ao código fonte a fim de conferir se o mesmo está funcionando corretamente em estágio de produção. A opção *-esa* ou *-enablesystemassertions* habilita *assertions* e a opção *-dsa* ou *-disablesystemassertions* desabilita as *assertions*. Veja alguns exemplos na **Listagem 6**.

Listagem 6. Comandos java para assertions.

```
java -esa <classes>
java -enablesystemassertions <classes>
java -dsa <classes>
java -disablesystemassertions <classes>
```

Outra situação que pode ocorrer e que pode ser tratada pelo comando *java* é o caso em que falta memória na JVM para criar novos objetos. Para aumentar a quantidade de memória reservada para a JVM, algumas opções estão disponíveis para configurar o tamanho inicial e máximo do *heap* (o espaço de memória acessível pela JVM). A opção *-Xms* permite definir o tamanho inicial do *heap* e o comando *-Xmx* permite definir o tamanho máximo. Ambos os comandos são exemplificados a seguir:

```
java -Xms <tamanho> <classes>
java -Xmx <tamanho> <classes>
```

Para simplificar, podemos configurar esses dois parâmetros de uma só vez. Deste modo, o comando a ser executado é:

```
java -Xmx2048m -Xms256m zipFilePkg.ZipFileReadTest
```

Nesse caso, a máquina virtual inicia com 256MB de memória e pode alocar até 2048MB. Além das opções listadas anteriormente, outras opções são oferecidas no interpretador *java*, como a que permite selecionar o modelo de dados específico para interpretação do código binário, sendo disponível modelos de 32 e 64 bits.

Este artigo abordou em detalhes e com exemplos o uso do pacote *java.util.zip* para compactar e descompactar arquivos e objetos. Como pode ser verificado, Java oferece suporte ao formato ZIP e GZIP desde o JDK 1.1 e desde então essa API tem sido expandida nas atualizações da plataforma, sendo adicionados novos recursos.

Com isso, dados podem ser compactados rapidamente, acelerando aplicações onde objetos necessitam ser transmitidos por uma conexão de dados, economizando largura de banda e otimizando o uso do disco para sistemas que geram/lidam com grandes quantidades de dados.

Autor



John Soldera

johnsoldera@gmail.com

É bacharel em Ciências da Computação pela UCS (Universidade de Caxias do Sul), mestre em Computação Aplicada pela Unisinos e cursa atualmente doutorado em Ciências da Computação pela UFRGS (Universidade Federal do Rio Grande do Sul). Trabalha com Java há 12 anos e possui a certificação SCJP.



Links:

Javadoc no site da Oracle.

<http://docs.oracle.com/javase/7/docs/api/>

Eclipse IDE.

<http://www.eclipse.org/>

NetBeans IDE.

<https://netbeans.org/>

Grails Framework: Criando aplicações web com Grails

Primeiros passos na iniciativa baseada na plataforma Java para promover o desenvolvimento ágil na web

Os métodos ágeis de desenvolvimento de software partem do princípio de que os requisitos e as soluções evoluem por meio da colaboração entre as partes envolvidas. Nesse sentido, o Manifesto Ágil, publicado em 2001, define o seguinte conjunto de valores e práticas para tornar o desenvolvimento de software mais “leve”, rápido e preciso:

- Foco nos indivíduos e suas interações ao invés de processos e ferramentas;
- Software funcional ao invés de documentação abrangente;
- Colaboração com o cliente ao invés de negociação de contratos;
- Resposta a mudanças ao invés de ficar preso ao planejado.

O que está por traz desse conjunto de valores é a busca pela solução para problemas clássicos do desenvolvimento de software, como a inadequação às necessidades dos clientes e o tempo de desenvolvimento. De forma geral, esses problemas vêm combinados e o desgaste entre as partes é inevitável – entregar no prazo é um grande problema e entregar exatamente o que o cliente deseja é um problema ainda maior.

Não é objetivo deste artigo se aprofundar nos princípios que regem o Manifesto Ágil, mas vale destacar que algumas técnicas de engenharia de software aplicam tais princípios, promovendo, por exemplo, a entrega contínua de artefatos funcionais, aproximação entre cliente e desenvolvedores, favorecimento da auto-organização do trabalho pelas equipes de desenvolvimento, etc. Destaca-se nesse contexto o SCRUM, que ganha cada vez mais espaço na comunidade de desenvolvedores pelo mundo.

Apesar desses esforços, algumas perguntas persistem, por exemplo: como o processo de desenvolvimento pode

Fique por dentro

Esse artigo traz um exemplo que demonstra os primeiros passos no desenvolvimento web com o Grails, framework de código aberto criado para trazer o paradigma da alta produtividade para a plataforma Java. Sendo assim, esse artigo é útil por promover um primeiro contato do desenvolvedor com um framework de alta produtividade feito para a plataforma Java e, portanto, compatível com todas as ferramentas e conceitos que Java disponibiliza.

ser efetivamente acelerado? É realmente possível abandonar uma parte da documentação em prol do software? A tarefa de programação pode ser simplificada?

Diversas iniciativas do campo da engenharia de software ou das linguagens de programação oferecem respostas para essas perguntas, acompanhando a crescente popularização do paradigma ágil. O Grails é uma dessas iniciativas, e responde ao menos uma dessas questões: a tarefa de programação pode sim ser simplificada.

Inicialmente chamado de *Groovy on Rails* – uma vez que incorpora a linguagem Groovy –, o Grails começou a ser desenvolvido em 2005 e seu primeiro beta foi lançado em 2006. Seu objetivo é ser um framework de alta produtividade para o desenvolvimento de aplicações web sobre a plataforma Java. Como é parte da arquitetura Java, é possível utilizar, a partir do Grails, de forma livre e transparente, todo o ferramental disponível, como Hibernate, Spring, Ant, etc.

Embora estáveis, as especificações de *JavaServer Pages* (JSP) e *Servlets* para o desenvolvimento na Web com Java não são simples. Para criar aplicações, são necessários conhecimentos sólidos nessas APIs e realizar tarefas de configuração e manutenção de ambientes. Isso pode representar uma desvantagem em relação a

outras plataformas nas quais, supostamente, o desenvolvimento pode ser mais simples, como PHP. Embora a literatura a respeito do tema não trace essa relação direta – e esse não seja o único problema que o Grails se propõe a resolver –, esse tipo de ocorrência justifica a opção por um framework de alta produtividade.

Mas o que é “alta produtividade” no desenvolvimento de software? Em resumo, significa tornar o desenvolvimento mais fácil e veloz, a partir da simples premissa de que o melhor resultado é aquele com menor tempo de desenvolvimento e que entrega o produto mais próximo do desejo do cliente.

Para perseguir esse objetivo, o Grails incorpora um conjunto de boas práticas que visam aumentar a produtividade do desenvolvimento, das quais se destaca o uso da Convenção sobre Configuração.

Convenção sobre configuração não é uma técnica ou um padrão, mas um paradigma de projeto cuja aplicação pode ser útil na redução da carga de trabalho cognitivo do desenvolvedor. Não se trata de limitar as decisões que o desenvolvedor toma, mas de livrá-lo das decisões mais simples para que ele volte sua capacidade cognitiva para as decisões mais complicadas.

Nesse caso, significa que o framework vai se ocupar das convenções mais comuns, como a nomenclatura de determinados conjuntos de classes ou do mapeamento objeto-relacional, livrando o desenvolvedor dessas responsabilidades, voltando sua atenção para o desenvolvimento, ao invés da configuração.

Outra característica importante do Grails é o fato de ter sido desenvolvido de acordo com o padrão *Model-View-Controller*. Dessa forma, o MVC é incorporado de maneira natural, sem que o desenvolvedor precise ter qualquer preocupação com sua implementação.

Assim como em outras linguagens dinâmicas, uma característica marcante na linguagem Groovy é a Meta-programação (ver **BOX 1**). Groovy disponibiliza de forma eficiente o recurso de *Reflection*, que permite que os objetos leiam seus atributos de maneira transparente. Outra característica comum nas linguagens dinâmicas, como a capacidade de se alterar o comportamento de um objeto em tempo de execução, também está presente. Além disso, recursos não tão eficientes, como a tipagem dinâmica, também estão disponíveis e, dependendo da aplicação, podem ser úteis para ganhar produtividade.

BOX 1. Meta-programação

O termo meta-programação é utilizado para descrever o recurso que algumas linguagens fornecem para que o software desenvolvido possa gerar ou modificar o próprio código, por exemplo, por meio da geração de funções ou classes “generalistas”. Linguagens que fornecem esse tipo de recurso geralmente são chamadas de linguagens dinâmicas.

Ainda no que diz respeito à integração com a plataforma Java, a compilação do código Groovy gera *bytecodes* para a JVM – o que faz com que as classes desenvolvidas em Groovy sejam completamente integradas a toda a biblioteca de objetos Java. Já as páginas da camada de apresentação são arquivos GSP (Groovy

Server Pages), bastante semelhantes às páginas JSP, utilizando, por exemplo, *Taglibs*.

Segundo a documentação disponível no site do Grails, aplicações com essa tecnologia podem rodar sobre qualquer *container* que suporte *servlets* a partir da especificação 2.5, como o Tomcat, JBoss, WebLogic etc. e o *deployment* se dá por meio de arquivos WAR, da mesma forma que aplicações Web em Java.

Com isso, o Grails se torna uma alternativa bastante eficiente para os desenvolvedores que querem ter a tarefa de programação simplificada, sem perder os benefícios oferecidos pela plataforma Java. Para demonstrar essas vantagens, neste artigo vamos desenvolver uma aplicação Grails simples que persiste dados em um banco de dados MySQL. Serão apresentados aqui desde a instalação do Grails até a construção do CRUD (*Create, Read, Update and Delete*) no banco de dados. Além disso, será demonstrado o uso da IDE NetBeans para apoiar o desenvolvimento.

Instalação do Grails

Como já citado, o Grails foi construído para funcionar sobre a máquina virtual do Java. Portanto, a instalação do ambiente de desenvolvimento depende da existência de um JDK. É importante ressaltar que uma JRE não é suficiente. Também é necessário que variáveis de ambiente estejam corretamente configuradas. Dessa forma, além da variável *JAVA_HOME*, é preciso que o diretório *bin* de seu JDK esteja presente na variável de ambiente *PATH*.

O exemplo apresentado nesse artigo foi desenvolvido no Ubuntu. Caso o leitor também esteja utilizando o Linux, a configuração das variáveis de ambiente deve ser feita conforme demonstra a **Listagem 1**. Na primeira linha, o comando *export* é utilizado para atribuir o valor correspondente ao caminho da instalação do JDK à variável *JAVA_HOME*. Na segunda linha, é demonstrado o uso do comando *export* para adicionar o diretório *bin* à variável de ambiente *PATH*.

Listagem 1. Configuração das variáveis de ambiente no Linux.

```
export JAVA_HOME=/caminho/do/Java  
export PATH="$PATH:$JAVA_HOME/bin"
```

No Windows deve ser utilizada a configuração das variáveis de ambiente disponível no *Painel de Controle*. Conforme mostra a **Figura 1**, no Painel de Controle, clique duas vezes na opção *Sistema*.

Na janela que será aberta, intitulada *Exibir Informações Básicas Sobre o Computador*, clique em *Configurações avançadas do sistema*. Feito isso será exibida a janela *Propriedades do Sistema*, como mostra a **Figura 2**. Nessa janela, clique no botão *Variáveis de Ambiente*.

Na janela *Variáveis de Ambiente*, mostrada na **Figura 3**, é possível configurar as variáveis *PATH* e *JAVA_HOME*. Para isso, no espaço *Variáveis do sistema*, selecione a variável em questão e clique em *Editar*.

Depois de baixar uma versão do Grails (o endereço para download está disponível na seção **Links**), descompacte o

arquivo e configure a variável de ambiente *GRAILS_HOME*, utilizando, no Linux, o comando *export*, conforme mostra a primeira linha da **Listagem 2**. Além disso, o diretório *bin* da instalação do Grails também precisa estar presente no *PATH* de seu sistema operacional, conforme mostra a segunda linha da **Listagem 2**.

Para verificar se o Grails foi corretamente instalado, utilize, no terminal (ou no *prompt* de comando do Windows) a instrução demonstrada a seguir:

```
grails -version
```

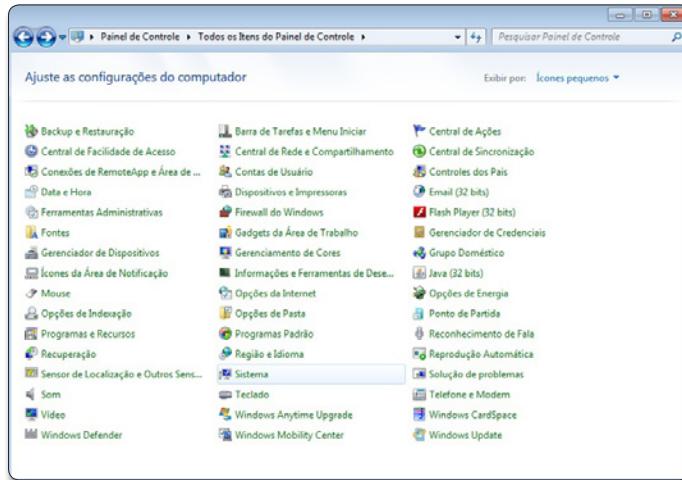


Figura 1. Painel de controle no Windows 7

Caso a instalação tenha sido realizada corretamente – e supondo que a versão instalada tenha sido a 2.3.3, a qual será utilizada na parte prática desse artigo – a seguinte resposta será obtida:

```
Grails version: 2.3.3
```

Feito isso, o Grails está instalado e pronto para o desenvolvimento.

Desenvolvendo a primeira aplicação Grails

O Grails tem como objetivo tornar o desenvolvimento Web mais simples e produtivo. Para isso, oculta detalhes de configuração da aplicação e faz com que a implementação de convenções e padrões seja mais transparente para o desenvolvedor.

Sendo assim, sua curva de aprendizado tende a ser menor, fazendo com que os recursos disponibilizados pela plataforma Java, que por sua complexidade apresentam certa barreira de entrada, sejam acessíveis a um conjunto maior de desenvolvedores.

Um exemplo simples: cadastrando Docentes e Especialidades

Para demonstrar a criação de uma aplicação simples em Grails, o exemplo demonstrado nesse artigo é composto de duas classes, *Docente* e *Especialidade*, conforme a **Figura 4**, e só serão implementadas quatro operações: Criação (ou inserção), Leitura (ou exibição), Exclusão e Atualização no banco de dados.

Listagem 2. Configuração da variável de ambiente *GRAILS_HOME* no Linux.

```
export GRAILS_HOME=/caminho/do/grails
export PATH="$PATH:$GRAILS_HOME/bin"
```

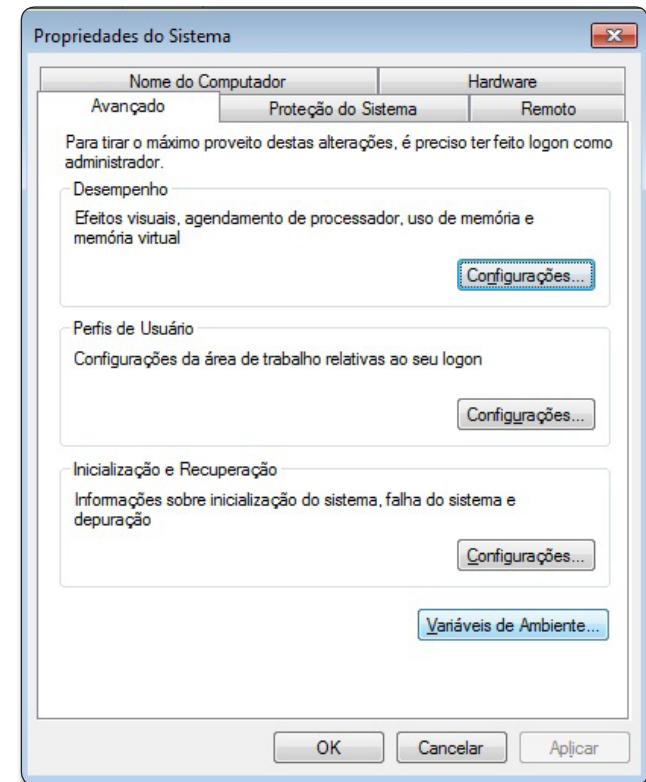


Figura 2. Janela “Propriedades do Sistema”

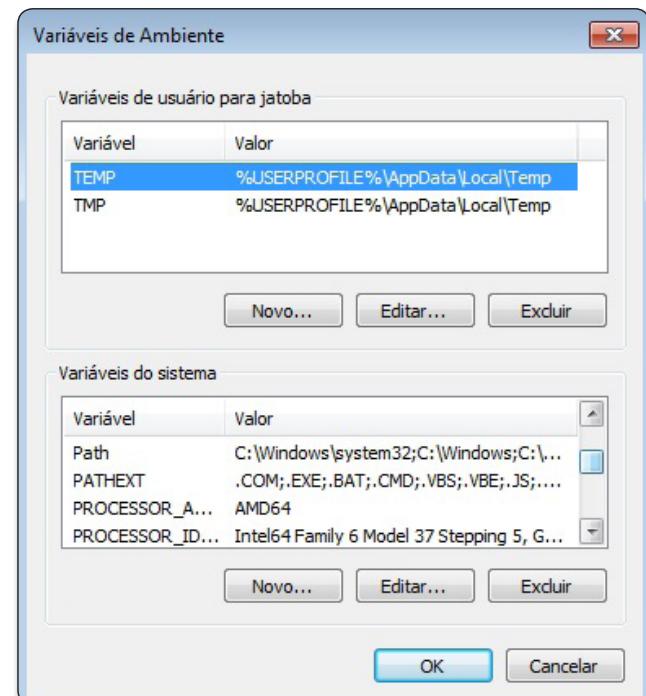


Figura 3. Janela “Variáveis de Ambiente”

Grails Framework: Criando aplicações web com Grails

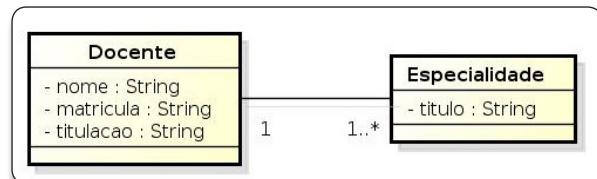


Figura 4. Diagrama de Classes de Domínio do exemplo

Repare no diagrama que, entre as classes, existe uma associação em que 1(um) objeto do tipo **Docente** se relaciona com 1 (um) ou muitos objetos do tipo **Especialidade**. Para que o artigo não fique longo demais e, ao mesmo tempo, para simplificar o exemplo, não será abordada a implementação da associação entre as classes.

Desenvolvendo aplicações web com Grails no NetBeans

Não é difícil utilizar os comandos do Grails. No entanto, uma vez que iniciamos o artigo falando sobre desenvolvimento ágil e ganho de produtividade, é útil que o desenvolvedor utilize uma IDE.

Não houve nenhum critério específico para a escolha do NetBeans, exceto pelo fato da versão utilizada ter suporte nativo ao desenvolvimento com Grails, ao contrário do Eclipse, que demanda a instalação de alguns plug-ins. Embora o NetBeans seja a IDE demonstrada nesse artigo, também serão demonstrados os comandos do Grails equivalentes aos recursos utilizados na IDE.

Para instalar o NetBeans, tanto no Linux quanto no Windows, basta executar o instalador. Seu download pode ser feito no site do fabricante (veja o endereço na seção **Links**) e a versão utilizada foi a 7.4. É importante, no momento do download, confirmar se a distribuição que você está baixando possui suporte à tecnologia Groovy.

Com essa compatibilidade garantida, ao criar um novo projeto estará disponível a opção *Aplicação Grails*, conforme expõe a Figura 5.

Criando um projeto Grails

Para começar o desenvolvimento de sua aplicação Grails, tenha em mente que, caso você opte pela linha de comando ao invés de utilizar uma IDE, precisará ter sempre à mão o comando *grails*. Seu uso segue a sintaxe *grails [instrução]*, onde a instrução vai variar de acordo com o que você deseja fazer na etapa do desenvolvimento em que estiver.

Nesse momento estamos criando nosso projeto, criando nossa aplicação e, portanto, utilizaremos o comando *grails create-app [nome do projeto]*. Para fazer isso, pelo terminal, entre no diretório que corresponda ao seu *workspace* e execute o seguinte comando:

```
grails create-app gestaoacademica
```

De acordo com a sintaxe desse comando, o nome do projeto que está sendo criado é “gestaoacademica”.

No NetBeans essa tarefa é ainda mais simples. A opção de menu *Arquivo > Novo Projeto* exibirá a janela demonstrada na Figura 5, onde você deve selecionar na aba *Categorias* a opção *Groovy* e,

na aba *Projetos, Aplicação Grails*. Na janela que é apresentada em seguida, clique em *Próximo*, digite o nome do projeto e selecione *Finalizar*.

Com isso, um novo diretório com o nome do projeto será criado e, dentro dele, a estrutura que, por convenção, compõe um projeto Grails. Na raiz do projeto, além do arquivo *application.properties*, que armazena propriedades gerais da aplicação, a estrutura de diretórios demonstrada na Figura 6 é criada.

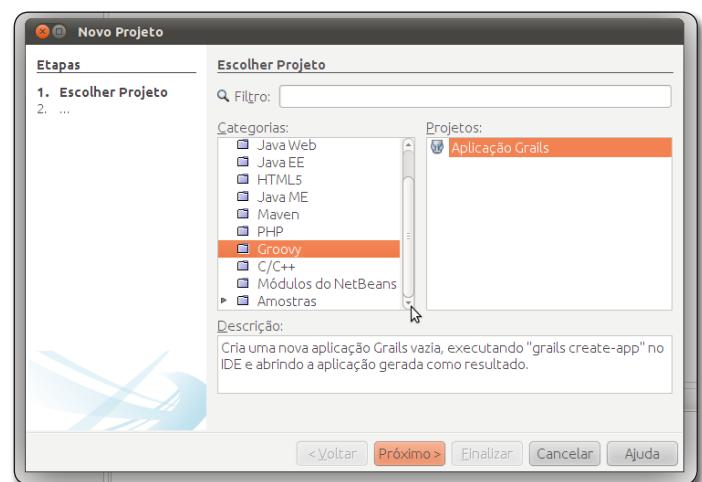


Figura 5. Criando um projeto Grails no NetBeans

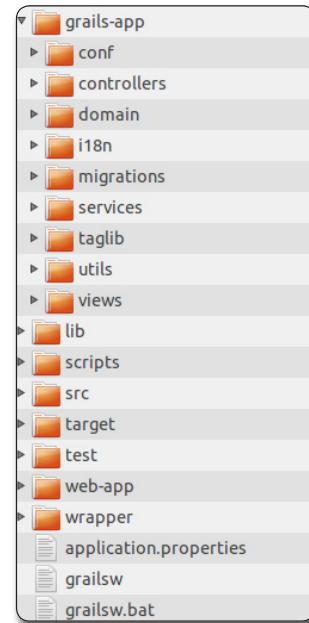


Figura 6. Estrutura de diretórios de um projeto Grails

Esse artigo dará destaque à estrutura que é gerada a partir do diretório *grails-app*. É nele que fica armazenado o código-fonte da aplicação e sua estrutura ilustra de forma muito clara o uso de convenções promovido pelo Grails, onde o nome e a localização dos arquivos são mais importantes do que o uso de arquivos de configuração. A estrutura desse diretório também pode ser vista na Figura 6.

Já é possível, nesse momento, ver como a adoção do padrão MVC se dá de forma natural no Grails. Os diretórios *domain*, *views* e *controllers* receberão o código-fonte de classes e páginas GSP que correspondem às camadas de **Modelo, Visão e Controle**.

No diretório *conf* ficam os arquivos de configuração da aplicação e os diretórios restantes estarão vazios ou não serão utilizados nesse exemplo.

Criando as classes de domínio para a camada de modelo

A construção do exemplo começa pela criação das classes de domínio da aplicação, de acordo com o diagrama demonstrado na **Figura 1**. Para fazer isso no terminal, utilize a linha de comando *grails create-domain-class [nome da classe]*. A linha de comando a seguir demonstra a criação da classe **Docente**:

```
grails create-domain-class docente
```

O resultado dessa operação é a criação, no diretório *grails-app/domain*, da classe de domínio **Docente**, no arquivo *Docente.groovy*, cujo código-fonte autogerado é demonstrado na **Listagem 3**. Modifique o código-fonte incluindo as declarações dos atributos da classe (neste caso: **nome**, **matricula** e **titulacao**).

Repare que no comando de criação da classe são usadas apenas letras em caixa-baixa e, ainda assim, o Grails respeita o uso de convenções e cria tanto o arquivo *Docente.groovy* quanto a declaração da classe em caixa-alta.

Listagem 3. Código-fonte da classe Docente.

```
package gestaoacademica

class Docente {

    String nome;
    String matricula;
    String titulacao;

    static constraints = {
    }
}
```

A sintaxe do Groovy apresenta diversas semelhanças com a do Java. Mais do que isso, é possível escrever código Java dentro de uma classe Groovy. Observe a declaração dos atributos na **Listagem 3**. No Grails, os atributos são, por *default*, privados. Os *setters* e *getters* dos atributos são implícitos e públicos, não havendo necessidade de escrevê-los, desde que o desenvolvedor não deseje alterar qualquer característica do encapsulamento.

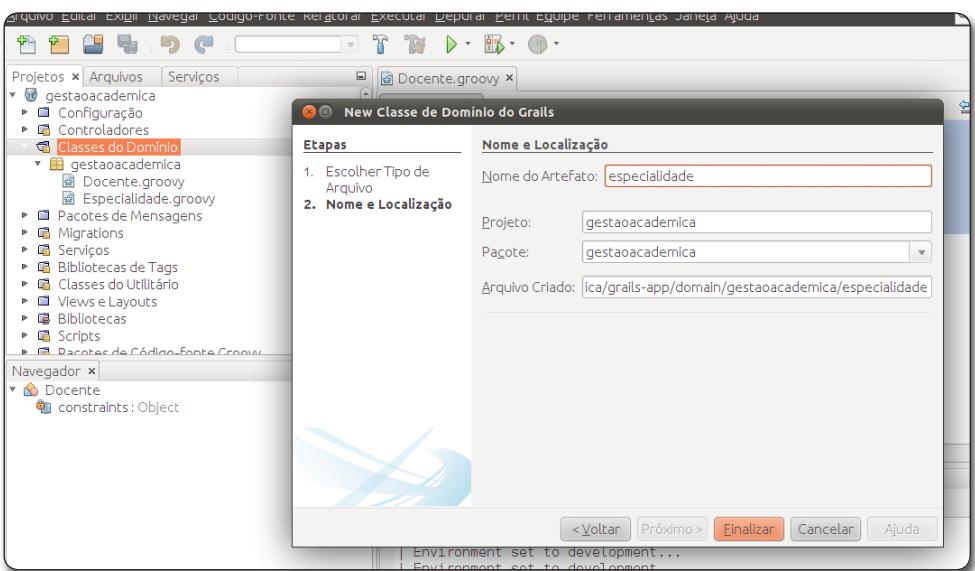


Figura 7. Criação de classe de domínio no NetBeans

No NetBeans, a tarefa de criação das classes de domínio é um pouco mais simples. Basta, com o botão direito sobre a opção *Classes do Domínio*, na aba *Projetos*, selecionar as opções *Novo > Classes de Domínio do Grails*. Na janela que é exibida, escreva o nome da classe – no nosso exemplo, agora criaremos **Especialidade** – e clique em *Finalizar*, como mostra a **Figura 7**.

Assim como na classe **Docente**, o Grails se encarregará de criar o arquivo *Especialidade.groovy* e, nesse arquivo, gerar o código correspondente à declaração da classe.

Com isso, todas as classes de domínio referentes ao projeto foram criadas. Podemos, então, partir para a criação das camadas subsequentes. Para isso, faremos uso de um recurso bastante popular nos *frameworks* de alta produtividade disponíveis no mercado e que o Grails não deixa de fora: *Scaffolding*.

Criando as demais camadas da aplicação com Scaffolding

O termo em inglês “*Scaffolding*” vem originalmente da Engenharia Civil. Significa dar suporte temporário para estruturas que estão sendo construídas, como, por exemplo, as escoras de madeira utilizadas para segurar lajes que estão secando.

Em programação, essa analogia é perfeitamente aplicada. O que se obtém após um processo de *Scaffolding* é a estrutura essencial das camadas cujos conteúdos, eventualmente, precisarão ser complementados com os algoritmos finais que atenderão ao negócio, na medida em que o processo de desenvolvimento progredir.

Como conceito, *Scaffolding* se popularizou com as ferramentas CASE como forma de ganhar produtividade no desenvolvimento. Sendo assim, *frameworks* com esse propósito baseados no padrão MVC se apropriaram dessa técnica, tendo como resultado a criação automática de uma estrutura básica e do código-fonte que torna possível inserir, exibir, atualizar e excluir (ou seja, o CRUD) registros em um banco de dados.

No Grails é possível realizar *Scaffolding* de dois modos: o primeiro, chamado de **dinâmico**, ocorre em tempo de execução.

Grails Framework: Criando aplicações web com Grails

No segundo modo, chamado de **estático**, as camadas são geradas durante o desenvolvimento, por meio do uso de comandos específicos do Grails. No exemplo apresentado nesse artigo, será utilizado o *scaffolding* estático.

O *scaffolding* estático é feito pelo comando *grails generate-all [Classe de Domínio]*, que gera de uma só vez as *views* necessárias para o CRUD, os *controllers* correspondentes e, nesses *controllers*, as *actions* (métodos) para cada uma das operações. Nesse caso, é mandatório o uso explícito do pacote e do nome da classe de domínio exatamente da forma como foi declarada, respeitando, por exemplo, a caixa-alta convencionada para o nome de classes.

Vale ressaltar que é possível fazer o *scaffolding* em partes, ou seja, criando as *views* e os *controllers* separadamente. Para gerar o *controller* para uma determinada classe de domínio, basta utilizar o comando *grails generate-controller [Classe de Domínio]*. Da mesma forma, para gerar as *views* de uma determinada classe de domínio, utilize o comando *grails generate-views [Classe de Domínio]*.

A seguir, é demonstrada a linha de comando a ser utilizada para realizar o *Scaffolding* para a classe de domínio **Docente**:

```
grails generate-all gestaoacademica.Docente
```

O resultado desse comando é a criação, dentro do diretório *grails-app/views*, da pasta *docente* e, dentro dela, dos arquivos: *create.gsp*, *edit.gsp*, *_form.gsp*, *index.gsp* e *show.gsp*, já com os códigos-fonte correspondentes. Por exemplo, a página *index.gsp* exibe uma listagem de registros, e até mesmo o código-fonte para a paginação é criado automaticamente.

Também é criado, no diretório *controllers*, o subdiretório *gestaoacademica* e, dentro dele, o arquivo *DocenteController.groovy*, com as *actions* *index*, *show*, *create*, *save*, *edit*, *update* e *delete*, referentes às operações básicas com o banco de dados.

Parte do código da classe *DocenteController.groovy* pode ser visto na **Listagem 4**. Nessa listagem, destacamos a operação **save()**, que é responsável por inserir um registro.

Para realizar o *Scaffolding* no NetBeans, na aba *Projetos*, selecione a classe de domínio desejada, clique com o botão direito do mouse e selecione *Gerar Tudo*, conforme mostra a **Figura 8**.

Assim como ocorreu com a classe de domínio **Docente**, todas as *views* e o *controller* correspondente foram criados para a classe **Especialidade**, bem como as *actions* para cada operação.

Modificando as views e utilizando templates

No Grails, as páginas podem ser construídas a partir de *templates*, para promover o maior reuso possível e facilitar não só a criação de páginas relacionadas, mas também sua manutenção. Os *templates* são “módulos” que podem ser incluídos em diversas páginas. No nosso exemplo, dois *templates* foram criados: um quando realizamos o *scaffolding* da classe **Docente** e outro quando realizamos o *scaffolding* da classe **Especialidade**. Ambos se chamam *_form.gsp* – por convenção, um *template* tem seu nome físico precedido de um underscore (“_”) – e correspondem aos formulários com os campos de cada classe. Como o mesmo formulário será utilizado

tanto na inserção quanto na atualização de um registro, é bastante conveniente que seu código-fonte esteja num arquivo que pode ser incluído em diversas páginas.

O aspecto visual padrão do *site*, como sua estrutura, a posição do cabeçalho e do rodapé, a posição dos menus e outros elementos que devem ser recorrentes nas páginas, é definido no que o Grails chama de *Layouts*, que são arquivos GSP – armazenados no diretório *views/layouts* – cujo código HTML define a aparência das páginas do projeto.

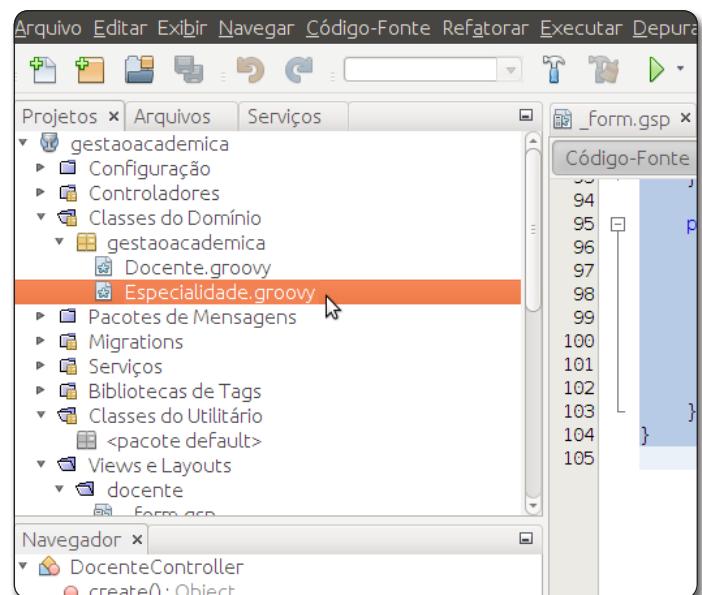


Figura 8. Realizando o Scaffolding estático no NetBeans

Listagem 4. Action save() no código fonte do controller de Docente.

```
...
@Transactional
def save(Docente docenteInstance) {
    if (docenteInstance == null) {
        notFound()
        return
    }

    if (docenteInstance.hasErrors()) {
        respond docenteInstance.errors, view:'create'
        return
    }

    docenteInstance.save flush:true

    request.withFormat {
        form {
            flash.message = message(code: 'default.created.message', args:
                [message(code: 'docenteInstance.label', default: 'Docente'),
                 docenteInstance.id])
            redirect docenteInstance
        }
        '*' { respond docenteInstance, [status: CREATED] }
    }
}
```

O código-fonte do *layout* padrão de cada projeto fica no arquivo *main.gsp*, sendo possível modificar o visual de sua aplicação alterando esse arquivo ou criando novos arquivos de *layout*. É possível, por exemplo, definir um *layout* específico para cada *controller*, atribuindo um valor à sua propriedade *layout*, como mostra a **Listagem 5**.

Nessa listagem é atribuído o layout “novo_layout” ao controller de **Docente**. Isso determina que as páginas referentes a objetos do tipo **Docente** terão sua aparência de acordo com o código HTML do arquivo *novo_layout.gsp*.

A **Figura 9** mostra o formulário de inserção de um novo docente, criado automaticamente pelo *scaffolding*, utilizando o *layout* padrão do Grails.

Listagem 5. Alterando o layout de um controller.

```
class DocenteController {
    static layout = 'novo_layout'
    ...
}
```

Figura 9. Formulário gerado pelo scaffolding

Embora ainda não tenhamos implementado qualquer interação com o banco de dados, o código gerado permite a utilização da aplicação perfeitamente, armazenando os objetos na memória, utilizando o H2 Database Engine, que acompanha a instalação do Grails de forma nativa.

Persistindo dados no MySQL

Por padrão, a aplicação Grails vem configurada para armazenar os dados apenas em memória. Com isso, a cada reinício da aplicação os dados são perdidos. Sendo assim, para dar maior consistência ao exemplo apresentado nesse artigo, alteraremos a configuração padrão para persistir os dados em um banco MySQL.

Para facilitar a implementação do acesso ao banco, o Grails usa de maneira transparente o mecanismo de persistência do Hibernate, de forma que seja possível realizar ao menos as operações

básicas de persistência sem a necessidade de qualquer intervenção do desenvolvedor.

Para fazer as configurações necessárias para acessar o MySQL, apenas um arquivo precisa ser modificado, o *DataSource.groovy*, presente no diretório *grails-app/conf/*.

Esse arquivo possui três seções importantes para o *deployment* do projeto, uma para cada ambiente em que a aplicação deve funcionar: *development* – para o ambiente de desenvolvimento; *test* – para o ambiente de testes; e *production* – para o ambiente de produção. Neste artigo será utilizado somente o ambiente de desenvolvimento. Logo, apenas a seção *development* será alterada.

Na **Listagem 6** é demonstrada a parte do arquivo *DataSource.groovy* que contém a configuração para que a aplicação utilize o banco de dados MySQL “gestaoacademica” no ambiente de desenvolvimento.

Neste mesmo arquivo, há uma seção chamada *DataSource*, onde é configurado o *driver JDBC* que será utilizado, bem como o usuário e a senha de acesso ao banco, conforme pode ser visto na **Listagem 7**. Certifique-se também de que o *jar* correspondente ao driver JDBC esteja presente no diretório *lib* do seu projeto.

Listagem 6. Arquivo de configuração do ambiente.

```
...
environments {
    development {
        dataSource {
            dbCreate = "create-drop"
            url = "jdbc:mysql://localhost/gestaoacademica?useUnicode=yes&characterEncoding=UTF-8"
        }
    }
    ...
}
```

Listagem 7. Configuração do data source.

```
...
dataSource {
    pooled = true
    driverClassName = "com.mysql.jdbc.Driver"
    username = "gestaoacademica"
    password = "gestaoacademica"
}
...
```

Nesse momento nada mais precisa ser feito e, uma vez que toda a estrutura necessária para o funcionamento da aplicação já foi construída, é possível executá-la. Para isso, utilize o comando *grails run-app*. No NetBeans, basta clicar no botão *Play*, como mostra a **Figura 10**.

Durante a inicialização da aplicação, as tabelas do banco de dados serão criadas a partir da estrutura definida nas classes de domínio e os dados passarão então a ser armazenados no MySQL que foi configurado. A **Figura 11** mostra a listagem dos docentes cadastrados, já realizando uma consulta no banco de dados.

Ao clicar na matrícula, o usuário é direcionado para o formulário de alteração, e para acessar o formulário de inserção, basta clicar em *Novo Docente*.

Grails Framework: Criando aplicações web com Grails

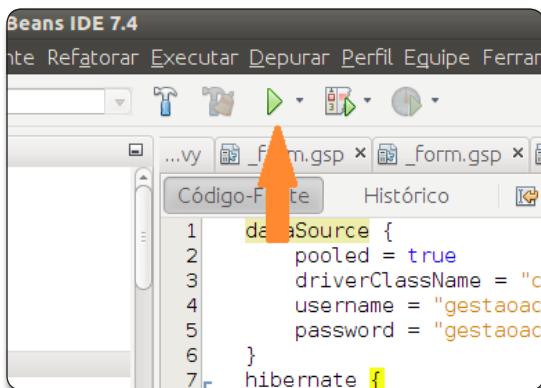


Figura 10. Executando a aplicação Grails no NetBeans

Figura 11. Listagem de docentes, exibindo dados do MySQL

Neste artigo foi apresentado um passo a passo para o desenvolvimento de uma aplicação Grails que persiste dados em um banco de dados MySQL. Foi apresentado também como utilizar o NetBeans para aprimorar a experiência de programação nesse framework.

O objetivo do artigo é possibilitar um primeiro contato do desenvolvedor com esse tipo de conceito, que prima pelo ganho de produtividade. Sendo assim, cabem futuros aprofundamentos na medida em que o desenvolvedor se depare com problemas mais complexos.

Não foi abordado nesse artigo, por exemplo, a implementação de associações e suas multiplicidades e muito menos a forma pela qual se dá o mapeamento objeto-relacional no Grails. Mais do que isso, vários outros recursos e conceitos em torno Grails precisam ser explorados mais detalhadamente, uma vez que podem representar ganhos significativos de produtividade – vale ressaltar sua integração com Ajax, a possibilidade de usar

classes Java livremente, DRY, a realização do deployment da aplicação, etc.

A web apresenta um horizonte imprevisível de oportunidades e de problemas. O Grails se posiciona nesse contexto para deixar à mão do desenvolvedor todo o ferramental disponível na plataforma Java para ser aplicado da maneira mais simples e flexível possível com alta produtividade. Logo, ferramentas utilizadas recorrentemente no desenvolvimento Java são integradas de forma transparente no Grails, e o desenvolvedor deve se sentir à vontade para usá-las.

É óbvio que o Grails não resolve todos os problemas. Como sabiamente disse Larry Flon, da Universidade Carnegie Mellon e autor do livro clássico *Fundamental Structures of Computer Science* (Estruturas Fundamentais da Ciência da Computação), “Não há linguagem de programação que impeça os programadores de fazerem maus programas, não importa quão estruturada ela seja”.

No entanto, na busca eterna pelo aumento da produtividade no desenvolvimento, Grails tem um papel importante, pois traz para o campo do desenvolvimento ágil toda a robustez característica e mundialmente estabelecida da plataforma Java, e os resultados dessa união são bastante animadores.

Autor



Alessandro Jatobá

jatoba@jatoba.org

É Mestre em Ciência da Computação pela UFRJ, Doutorando em Engenharia também pela UFRJ e, durante 2014 e 2015, fará um Doutorado-Sanduíche pelo Departamento de Engenharia e Projeto de Sistemas da Universidade de Waterloo, no Canadá. Trabalha com Java há mais de 15 anos, a maior parte deles liderando equipes de desenvolvedores em projetos Web. Também é professor universitário lecionando temas ligados ao desenvolvimento orientado a objetos.



Links:

Site oficial do Grails.

grails.org

Site oficial do NetBeans

Netbeans.org

Site da linguagem Groovy

<http://groovy.codehaus.org>

Novidades do Groovy 2.2

Pequenas melhorias que aumentam a produtividade do desenvolvedor

Groovy é uma das linguagens mais populares voltadas para a JVM. A primeira versão do projeto, lançada em 2003, conquistou diversos programadores Java ao apresentar uma sintaxe familiar à que estavam acostumados, ao mesmo tempo em que removia a verbosidade de diversas tarefas comuns ao seu dia a dia. Groovy fazia mais do que simplesmente reduzir a burocracia. Por ser dinâmica, acabou propiciando a criação de um dos frameworks mais populares para desenvolvimento web, o Grails, que se tornou um dos principais impulsionadores da popularidade da linguagem.

Um fator que limitava sua introdução em diversos projetos era o seu desempenho, que realmente era bastante inferior ao do Java devido à natureza dinâmica da linguagem. Esta realidade começou a mudar rapidamente com o lançamento do release 1.8 e, a partir de julho de 2012, com o lançamento da versão 2.0, que trouxe a checagem estática de tipos, um recurso opcional que melhorou bastante a situação. Até este momento era comum o Groovy ser dezenas de vezes mais lento que o Java. Hoje, felizmente, o cenário é bem diferente, com a linguagem estando em um patamar de performance bem próximo ao do Java. Para comprovar esta melhoria, diversos benchmarks foram feitos. Como exemplo, confira o que se encontra na seção [Links](#).

A versão 2.0 também trouxe uma série de melhorias para a linguagem no que diz respeito à produtividade dos desenvolvedores. Algumas delas, inclusive, inspiradas no Java 7, como instruções do tipo multi catch, melhorias na integração com o Java, uso da instrução invokedynamic, e diversas outras funcionalidades. A tendência na melhoria da produtividade se confirmou na versão 2.1 e, finalmente, na versão que trata este artigo, a 2.2, que, apesar de ser um release com poucas novas funcionalidades, traz algumas que tornarão ainda mais simples a vida do desenvolvedor.

Com base nisso, neste artigo veremos alguns dos novos recursos presentes na versão 2.2, lançada em novembro de 2013 e que já possui seu primeiro bug fix, a versão 2.2.1, lançada duas semanas após o lançamento do release oficial.

Fique por dentro

Em novembro de 2013 foi lançada a versão 2.2 da linguagem de programação Groovy, que veio com pequenas melhorias com o objetivo de aumentar a produtividade do desenvolvedor e mais uma vez melhorar o desempenho da linguagem. Neste artigo veremos as principais novidades deste último release e de que forma você poderá aplicá-las em seus projetos.

Dentre as linguagens voltadas para a JVM, Groovy ainda é a mais popular. Esta popularidade se justifica por ela apresentar uma sintaxe muito próxima à do Java, porém removendo grande parte da sua verbosidade, e também por ser uma linguagem dinâmica. A partir da versão 2.0 muitas melhorias vêm sendo incluídas no projeto com o objetivo de aprimorar seu desempenho e aumentar a produtividade do desenvolvedor.

Coerção implícita de closures

Uma das novidades mais comentadas e previstas para o Java 8 é a inclusão das expressões lambda, recurso bastante similar às closures, com as quais estamos acostumados a trabalhar no Groovy. Na implementação deste recurso em Java, há um aspecto que chamou a atenção da equipe de desenvolvimento do Groovy: o modo como as “closures do Java” podem ser convertidas de forma transparente em interfaces ou classes abstratas que possuem apenas um método abstrato. Soa estranho este recurso, não é mesmo? Então vamos clarificá-lo através da apresentação de um conceito e também alguns exemplos, mostrando como esta característica do Java ajudou os projetistas do Groovy a melhorar ainda mais a linguagem.

Quando uma proposta de melhoria é feita à linguagem Groovy, primeiro ela é documentada sob a forma de uma GEP (*Groovy Enhancement Proposal* – Proposta de Melhoria ao Groovy). Na GEP que deu origem à coerção implícita de closures (a qual você pode conferir na seção [Links](#)), foi introduzido o conceito de SAM, sigla para *Single Abstract Method*, cuja tradução para o português seria algo como: “Um único método abstrato”. Usamos este nome complexo para representar qualquer interface ou classe abstrata que possua um único método abstrato em sua definição. Na [Listagem 1](#) podemos ver alguns exemplos de SAM, que pode ser aplicado tanto em Java quanto em Groovy.

Novidades do Groovy 2.2

Listagem 1. Exemplos de SAM.

```
public interface Calculo {  
    public int soma(int x, int y);  
}  
  
public abstract class Integracao {  
    public abstract void integre();  
}  
  
// Nota: repare que a regra se mantém. Temos um único método abstrato  
public abstract class Compras {  
    public abstract double total();  
    public void adicionaItem(Item item);  
}
```

É importante mencionar que a herança não invalida o conceito de SAM. Assim, uma classe abstrata que possui uma superclasse diferente de **Object** ainda entra nesta categoria se em sua classe pai é definido um único método abstrato, tal como na **Listagem 2**.

Agora que o conceito de SAM está explicado, cabe o analisarmos na prática. Para isto, iniciaremos com um exemplo bastante simples. Em Groovy, toda coleção (classe pertencente à API Collections do Java) recebe métodos que têm como parâmetro uma closure, como é o caso do método **findAll()**. Este retorna todos os itens de uma lista que satisfaçam a uma determinada condição expressa por uma closure, como exposto na **Listagem 3**.

Listagem 2. Exemplo de SAM em um caso que envolve herança.

```
// Naturalmente SAM  
public abstract class Integracao {  
    public abstract void integre();  
}  
  
// Continua sendo SAM. Observe que o método abstrato único fica na classe pai  
public abstract class IntegracaoBD extends Integracao {  
    public void persiste() {  
        // faça algo  
    }  
}
```

Listagem 3. Chamada tradicional do método findAll() em Groovy.

```
def lista = [1,2,3,4,5,6,7,8]  
lista.findAll { it > 5}  
// retorna [6,7,8]
```

É comum nos depararmos com código como o demonstrado na **Listagem 4**, que tira proveito dessa característica da linguagem. Nesta listagem, a função **busca()** recebe como parâmetro uma closure e faz exatamente o que foi exposto na **Listagem 3**.

Nota-se que não há nada de errado com o código da **Listagem 4**, mas há situações nas quais o desenvolvedor pode querer mais controle a respeito de qual tipo de closure deve ser executada, como por exemplo, quando possui código legado que já esteja seguindo o padrão SAM e que possa ser aplicado em um projeto Groovy. Neste cenário, imagine que tenhamos uma interface chamada **Filtro**, exposta na **Listagem 5**, que se enquadra na categoria SAM.

Listagem 4. Tirando proveito de closures.

```
def lista = [1,2,3,4,5,6,7,8]  
def closureTeste = {it > 5}  
  
def busca (closure) {  
    lista.findAll closure  
}  
busca closureTeste  
// retorna [6,7,8]
```

Listagem 5. Código da interface Filtro (um SAM).

```
public interface Filtro {  
    boolean testarItem(Object obj);  
}
```

Como poderíamos usar essa interface de forma transparente em Groovy como se fosse uma closure? Entra em cena a coerção implícita de closures, que é, na realidade, a tipificação do uso de closures, tal como no exemplo exposto da **Listagem 6**, na qual reescrevemos a função **busca()** exposta na **Listagem 4**. Nossa função não mais espera receber apenas uma closure; agora pode trabalhar também com qualquer objeto que implemente a interface **Filtro** exposta na **Listagem 5**.

Listagem 6. Aplicando a coerção implícita de closures.

```
1.def lista = [1,2,3,4,5,6,7,8]  
2.def closureTeste = {it > 5}  
3.// FiltroImpl é uma classe concreta que implementa a interface Filtro  
4.def implementacaoFiltro = new FiltroImpl();  
5.def busca(Filtro filtro) {  
6.    lista.findAll filtro  
7.}  
8.busca closureTeste // funciona conforme vimos nas listagens 3 e 4  
9./*  
10. Um objeto é tratado como se fosse uma closure de forma  
11. totalmente transparente  
12. */  
13..busca implementacaoFiltro
```

A **Listagem 6** expõe bem a coerção implícita de closures. O modo tradicional com o qual estamos acostumados a lidar com closures continua intacto, tal como podemos ver na linha 8, na qual passamos a closure definida na linha 2. Na linha 4 instanciamos uma implementação da interface **Filtro**, que poderia muito bem ter sido escrita em Java, por exemplo. Finalmente, na linha 13 vemos a coerção implícita de closures aplicada ao passarmos como parâmetro para a função **busca** um objeto que implementa a interface **Filtro**.

Cacheando resultados com @Memoized

Um recurso muito interessante incluído no Groovy 2.2 é a anotação **@Memoized**. Ela é usada para cachear resultados de funções sobre as quais é aplicada. A ideia é bastante simples: se temos em nossas classes uma função que sempre retorna o mesmo valor para o mesmo conjunto de parâmetros e que não depende do estado interno de um ou mais objetos, por que repetir o custo computacional se podemos cachear o resultado para uso posterior?

Este é um recurso interessante, mas que deve ser usado com cuidado: lembre-se de apenas aplicá-lo em pontos do código nos quais o resultado final do processamento não dependa de estados externos e que, de preferência, sejam executados repetidas vezes durante a execução do sistema. De nada adianta cachear resultados que raríssimas vezes serão utilizados (seria mero desperdício de memória).

O código da **Listagem 7** expõe um bom uso deste recurso.

Listagem 7. Exemplo de uso de @Memoized.

```
// Lembre-se de incluir esta instrução de import em seu código
import groovy.transform.Memoized

class CalculoImpostos {
    @Memoized
    public double impostoAPagar(double imposto, double porcentagem) {
        //efetua o cálculo e retorna o valor
    }
}
```

Na **Listagem 7**, quando executamos a função `impostoAPagar()` com os parâmetros 1.000 e 20 pela primeira vez, é feito o processamento e logo em seguida nos é retornado quanto devemos pagar em impostos. Na segunda execução do método, caso os mesmos parâmetros sejam passados, ao invés do método `impostoAPagar()` ser executado, o que realmente ocorrerá será o retorno do último valor computado para aqueles argumentos. Quando bem aplicado, este recurso tende a aumentar significativamente a performance do sistema como um todo.

Mudando a classe básica de execução de scripts

Para muitos programadores, Groovy é basicamente uma linguagem de script usada para agregar dinamismo ao código Java, cujo uso consiste em uma estratégia composta por três passos:

1. O script Groovy é persistido em um arquivo ou em um banco de dados;
2. O conteúdo do script é carregado em memória e em seguida compilado;
3. O código compilado será uma classe que estende `groovy.lang.Script` que será manipulada pelo nosso código Java.

Mas o que acontece se não quisermos trabalhar com uma classe que estenda `groovy.lang.Script`, mas sim com alguma internamente desenvolvida que estenda esta classe e adicione alguma funcionalidade específica para o nosso projeto, tal como a que expomos na **Listagem 8**?

Até a versão anterior do Groovy esta poderia ser uma tarefa bem complicada de ser cumprida. A partir da versão 2.2, no entanto, ficou bastante simples. Para isto, basta usarmos a anotação `@BaseScript` em nossos scripts, como apresentado na **Listagem 9**. É importante mencionar que o script deve possuir um atributo chamado `baseScript`, que corresponderá à classe customizada que será usada pelo desenvolvedor.

Listagem 8. Nossa customização de `groovy.lang.Script`.

```
public abstract ScriptCustomizado extends groovy.lang.Script {
    public String getNode() {
        // retorna o nó do cluster no qual o código será executado
    }
}
```

Listagem 9. Exemplo de uso de `@BaseScript`.

```
// Lembre-se de incluir este import
import groovy.transform.BaseScript

// Definimos qual o script básico. Atenção para o nome do atributo: baseScript
@BaseScript ScriptCustomizado baseScript

// Conteúdo do script como normalmente o escreveríamos

// executando no script o método da classe ScriptCustomizado
// que definimos com a anotação @BaseScript
println getNode()
```

Suporte ao Log4j2 usando a anotação `@Log4j2`

A anotação `@Log` é uma antiga conhecida dos programadores Groovy. Seu uso é bastante simples: aplique-a sobre uma classe e uma nova propriedade, chamada `log`, será “automagicamente” incluída nesta, possibilitando assim usar de forma transparente e bastante elegante seu framework de logging favorito. Até a versão 2.2 da linguagem, havia suporte para os seguintes frameworks de log: Log4j (1.x), Apache Commons Logging, SLF4J e a API padrão do próprio Java. A grande novidade agora é o suporte à versão 2.x do Log4j.

Curiosamente este recurso não é tão empregado quanto deveria. Sendo assim, nesta seção iremos descrever seu uso de uma forma geral para que você, programador Groovy que ainda não tirou proveito deste recurso, possa usá-lo de uma forma bem simples. Nossa foco será no suporte ao Log4j2. Para as demais bibliotecas, sugerimos a leitura da documentação oficial presente no site da linguagem (veja a seção **Links**). Na **Listagem 10** podemos ver um exemplo do seu uso.

Listagem 10. Exemplo de uso da anotação `@Log4j2`

```
// anotação de logging que você deve importar para o Log4j2
import groovy.util.logging.Log4j2

// A anotação que deve ser aplicada à sua classe
@Log4j2
class Integracao {
    public void integre() {
        // atributo log inserido de forma transparente em sua classe pelo Groovy
        log.info "Iniciando integração"
        // Código da função integre()
        log.info "Integração finalizada"
    }
}
```

Quando aplicamos a anotação tal como exposto na **Listagem 10**, o logger terá como nome o mesmo que o da classe que anota. Caso queira customizar esse nome, você deve passá-lo como parâmetro para a anotação, como demonstra a **Listagem 11**.

Novidades do Groovy 2.2

Listagem 11. Customizando o nome do logger.

```
import groovy.util.logging.Log4j2
@Log4j2("IntegracaoNomeCustomizado")
class Integracao {
    // Código fonte da classe
}
```

Ao usar a anotação `@Log4j2`, lembre-se de incluir as dependências do Log4j 2.x no classpath do projeto. O restante da configuração desse framework de logging é exatamente a mesma que seria realizada em um projeto Java convencional.

Conforme mencionado anteriormente, o Groovy já oferecia suporte a frameworks de logging antes da versão 2.2. Para tirar proveito da sua opção de logging favorita, basta usar uma das anotações presentes no pacote `groovy.util.logging`, que se encontram listadas na Tabela 1.

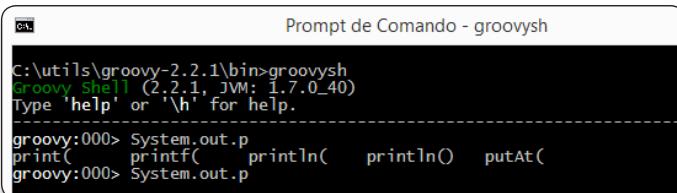
Anotação	Framework de logging
<code>@Commons</code>	Apache Commons Logging
<code>@Log</code>	Java Util Logging
<code>@Log4j</code>	Log4j 1.x
<code>@Log4j2</code>	Log4j 2.x
<code>@Slf4j</code>	LogBack

Tabela 1. As diferentes anotações de logging do Groovy

Melhorias no Groovysh

Groovysh é uma interface de linha de comando que fornece aos desenvolvedores Groovy um REPL. REPL é a sigla para *Read Eval Print Loop* e trata-se de uma ferramenta bastante útil em momentos nos quais o programador deseja experimentar aspectos da linguagem e suas bibliotecas de uma forma simples.

Ao executar o comando `groovysh` no diretório `bin` da sua instalação do Groovy, será exposta a interface de linha de comando que recebeu melhorias interessantes neste release da linguagem. Talvez a mais importante seja o recurso de autocompletar. Com ele, enquanto digita seu código, pressione a tecla `tab` e serão apresentadas opções tal como exposto na Figura 1.



```
Prompt de Comando - groovysh
C:\utils\groovy-2.2.1\bin>groovysh
Groovy sh! (2.2.1, JVM: 1.7.0_40)
Type 'help' or '\h' for help.

groovy:000> System.out.p
print( printf( println( println() putAt(
groovy:000> System.out.p
```

Figura 1. Exemplo de uso do recurso auto completar no Groovysh

Outra melhoria bastante interessante no groovysh foi a inclusão do comando `doc`. Se o usuário digitar este comando, seguido do nome de uma classe ou método desta, por exemplo, será aberta uma nova janela do browser padrão do seu sistema operacional

expondo o javadoc da mesma, o que na prática transforma o groovysh em uma excelente ferramenta de ajuda para programadores que estejam começando na linguagem.

Concatenação de strings mais rápida

Há também uma melhoria sutil neste release que pode trazer ganhos de performance para aplicações que façam uso intenso da concatenação de strings com o operador `+`. Na Listagem 12 podemos ver um exemplo de uso deste novo recurso.

Listagem 12. Exemplos de concatenação de strings.

```
String s = ""
for (int i = 0; i < 10000; i++) {
    s = "a" + "b" // era lento
    // s = "a".concat("b") (mais rápido, porém incômodo de escrever)
}
```

O modo tradicional de concatenação (usando o operador `+`) é muito mais lento que a outra forma exposta na mesma listagem, baseada no método `concat()` da classe `String`. A novidade é que agora o compilador internamente substitui o uso do operador `+` pela função `concat` da classe `String`, obtendo assim um ganho de desempenho interessante para aplicações que façam uso intenso de concatenação de strings.

Para comprovar estes ganhos de desempenho, fizemos um benchmark simples comparando as versões 2.1.6 e 2.2.1 da linguagem. O código exposto na Listagem 13 expõe o teste realizado. Concatenamos strings das duas formas expostas na Listagem 12 100.000 vezes em ambas as versões da linguagem, e em seguida, comparamos os tempos em nanosegundos.

Listagem 13. Código do benchmark de concatenação de strings.

```
String str = ""
long inicio = System.nanoTime()
for (int i = 0; i < 100000; i++) {
    // str = "a" + "b"
    str = "a".concat("b")
}
long fim = System.nanoTime()
println "Tempo total: ${fim - inicio}"
```

O computador usado para os testes foi um notebook com processador Intel i5 com clock de 1.7 GHz e 6Gb de memória RAM. Para cada versão da linguagem o teste foi executado seis vezes. Na Tabela 2 podemos ver o resultado obtido e que comprova uma redução próxima de 20% no tempo de execução. É interessante observar que, como esperado, o tempo na execução usando a instrução `concat()` se mostrou quase constante.

Previsões para o Groovy 3.0

A versão 2.2.1 muito provavelmente foi o último release da linguagem antes da próxima grande versão do Groovy, a 3.0, cujo lançamento está previsto para março de 2014, tal como publicado no roadmap da linguagem (veja a seção [Links](#)).

Versão do Groovy	Usando o operador “+”	Usando a função concat() de String
2.1.6	72780949	18959447
2.2.1	59632515	19986980

Tabela 2. Resultado do benchmark

Apesar de sempre ter sido uma linguagem muito bem integrada ao ambiente de execução Java, esta integração se tornará ainda melhor com a reescrita do protocolo de meta-objeto do Groovy (*meta-object protocol, MOP*). Esta é a parte da linguagem responsável pelo seu aspecto dinâmico. O objetivo da refatoração de um componente tão importante é tirar o máximo proveito da instrução *invokedynamic* do Java.

Invokedynamic é, na realidade, nada mais do que mais uma instrução de bytecode Java que é executada pela JVM, porém bastante importante para aqueles que desenvolvem novas linguagens para a JVM. Até então as equipes de desenvolvimento de linguagens alternativas (especialmente as dinâmicas) como Groovy e Scala precisavam fazer verdadeiros malabarismos na escrita de seus compiladores. Malabarismos estes que muitas vezes não tiravam o máximo proveito de otimizações que a JVM poderia lhes fornecer e, com isto, perdia-se bastante em performance de um código que já é complexo por natureza. Com *invokedynamic* a integração de novas linguagens à JVM tornou-se mais simples, visto que menos artifícios são necessários, facilitando a escrita de compiladores.

Além da reescrita do MOP, este release será fundamental para garantir a compatibilidade do Groovy com o Java 8. Como mencionado neste artigo, uma das principais novidades deste novo release do Java será a inclusão de closures. Com isto a equipe de desenvolvimento do Groovy tem concentrado grandes esforços no objetivo de garantir que closures Groovy possam ser usadas de forma transparente por código Java e vice-versa.

Neste artigo foram apresentadas as principais melhorias presentes na versão 2.2 do Groovy no que diz respeito a funcionalidades que aumentam a produtividade do desenvolvedor. Como pode ser observada, a novidade que aparentemente parece ser mais complexa, a coerção implícita de closures, na realidade é um recurso bastante simples que basicamente tipifica o modo como lidamos com closures em nosso código. É interessante notar o fato de termos aqui um recurso diretamente inspirado no Java 8, o que mostra um indício de que a próxima versão do Java realmente trará ganhos importantes de produtividade para o desenvolvedor.

Uma dica importante para aqueles que acompanham de perto a evolução do Groovy é sempre comparar o release notes oficial disponibilizado pelo site oficial, com o gerado de forma automática pelo Jira, ferramenta de gestão de bugs e melhorias usada pela equipe de desenvolvimento do projeto. Isto nos permite entender

melhor pontos deixados em aberto como, por exemplo, qual foi a melhoria de performance da linguagem. Além disto, podemos também ver quais foram os bugs resolvidos, bugs estes que muitas vezes podem ter sido a causa de diversas horas extras gastas por desenvolvedores sem que estes soubessem.

Ao atualizar o Groovy é importante levar em consideração dois pontos. A última versão do Grails (2.3.3) ainda está lidando com a versão 2.1.9 dessa linguagem. Portanto, caso esteja ansioso para tirar proveito das novidades descritas neste artigo em seu projeto Grails, recomendamos que espere pelo próximo release do framework, que muito provavelmente deverá incluir a versão 2.2 da linguagem. O segundo ponto é, provavelmente, o mais importante, diz respeito a qual versão da linguagem baixar. Opte pela 2.2.1, que já traz uma série de correções de bugs encontrados na versão 2.2.

Esperamos com este artigo ter incentivado o leitor a experimentar as novidades desta linguagem que não para de evoluir e que, em 2014, trará ainda mais novidades com o release 3.0.

Autor



Henrique Lobo Weissmann

É arquiteto com vasta experiência em projeto e desenvolvimento de aplicações corporativas. É especialista nas plataformas Java EE, Spring e Groovy/Grails, sendo o fundador do Grails Brasil e da itexto, onde presta consultoria para pequenas e médias empresas.



É também autor do livro “Vire o Jogo com Spring Framework”, publicado recentemente pela editora “Casa do Código”.

Links:

Release notes do Groovy 2.2.

<http://groovy.codehaus.org/Groovy+2.2+release+notes>

API de Logging do Groovy: documentação oficial.

<http://groovy.codehaus.org/Logging>

GEP 12: SAM Coercion (deu origem à coerção implícita de closures).

<http://docs.codehaus.org/display/GroovyJSR/GEP+12+-+SAM+coercion>

Benchmark do Groovy 2.0 contra o Java.

<http://objectscape.blogspot.com.br/2012/08/groovy-20-performance-compared-to-java.html>

Página oficial da linguagem Groovy.

<http://groovy.codehaus.org>

Melhoria na performance da concatenação de strings em Groovy.

<http://jira.codehaus.org/browse/GROOVY-5956>

Roadmap do Groovy.

<http://groovy.codehaus.org/Roadmap>

Analisando o Big Data na teoria e na prática

Entenda o Big Data e as principais tecnologias que fazem parte de seu domínio

Há vários anos vem acontecendo uma progressão do modo como os dados estão sendo acumulados. Antes da Internet, a maioria dos dados eram aqueles que eram gerados e acumulados pelos sistemas de computados das empresas. Porém, após o surgimento da rede mundial de computadores, os usuários tornaram-se capazes de gerar seus próprios dados. Essa causa teve como efeito um aumento exponencial na geração de informação.

Essa explosão de dados já faz parte do nosso cotidiano. Um dos maiores contribuintes para isso foi o surgimento das redes sociais. No Facebook, atualmente a maior rede social do mundo, em 24 horas são compartilhados bilhões de conteúdos entre seus usuários. Somado a isso, em apenas um minuto aproximadamente 100.000 mensagens são disparados pelo Twitter, 700.000 buscas são realizadas no Google e 600 vídeos são adicionados no Youtube.

Além de todas essas fontes de informação, máquinas também estão acumulando dados. Existem trilhões de sensores espalhados nos diversos tipos de dispositivos, os quais coletam bilhões de dados diariamente. Aviões, por exemplo, geram aproximadamente 2,5 bilhões de terabytes de dados anualmente por sensores instalados em seus motores.

Nos conjuntos de dados de alguns departamentos policiais, por exemplo, existem milhões de registros de crimes que foram armazenados durante o decorrer de dezenas de anos, contendo informações de como e onde os crimes aconteceram. Hoje em dia esses dados já são utilizados para entender a natureza de um crime e identificar potenciais lugares de possíveis ocorrências. Com estas informações valiosas, estudos estão sendo realizados com a intenção de tornar possível predizer o local e modo de prováveis incidências de crimes.

Como demonstra o exemplo anterior, a habilidade de se trabalhar com Big Data abriram inúmeras possibilidades. A partir do Big Data diversos casos de fraudes foram

Fique por dentro

Este artigo tem por objetivo desmistificar a revolução que está sendo o Big Data, e como ele está transformando o modo como vivemos, trabalhamos e pensamos. Também será apresentado ao leitor algumas das principais tecnologias relacionadas a esta palavra tão repercutida nos dias de hoje, especialmente na área de tecnologia da informação. Sendo assim, o tema abordado é útil para todos aqueles que desejam conhecer e aprender sobre o que é Big Data, e entender como ele está influenciando a evolução das ferramentas de armazenamento e processamento de dados.

descobertos por algoritmos que analisavam os e-mails corporativos de algumas empresas. Esta tecnologia também possibilitou evitar catástrofes naturais e humanas. Através de ferramentas poderosas com capacidade suficiente para analisar o código genoma, foi possível também descobrir padrões, encontrar curas de doenças, novas estrelas, entre muitas outras informações.

Diante disso, a capacidade de armazenar essa colossal quantidade de dados foi apenas uma das preocupações das companhias. Para transformar esses dados em informações “palpáveis”, foi necessário evoluir também as ferramentas e tecnologias de processamento e análise de dados. Neste cenário, a possibilidade de minerar todas essas informações teve um papel importante na previsão de acontecimentos e no estudo de comportamentos.

Quando tratados da maneira correta, os dados podem fornecer informações valiosas para as empresas. E já não é de hoje que as empresas enxergam o quanto valiosos são os dados contidos em seus bancos. Tecnologias modernas de BI (*Business Intelligence*) como data warehouse conseguem lidar com vastas quantidades de informações para ajudar a identificar e desenvolver oportunidades de negócio.

Entretanto, esse grande aumento no acúmulo e na diversidade de dados que está ocorrendo nos tempos atuais acaba gerando muita dificuldade para as ferramentas tradicionais serem capazes de lidar. Com isso, a tecnologia que tínhamos acabou se tornando insuficiente para atender o negócio, forçando-nos a

buscar outras maneiras, processos e ferramentas, capazes de possibilitar o trabalho com tanta e variada informação. Como pode ser notado, vivemos em uma era onde geramos mais informações do que somos capazes de armazenar, processar, analisar e gerar algum valor.

Tendo isso em vista, empresas pioneiras no Big Data, como Google, Facebook, Twitter e Amazon, estão utilizando estes conjuntos de dados colossais para descobrir diversas percepções escondidas dentro de toda essa informação. Através do Big Data, a sociedade adquiriu a oportunidade de aproveitar essa vasta quantidade de informação de uma maneira extraordinária, com a finalidade de produzir ideias ou produtos e serviços de valores significativos.

Com base nisso, este artigo irá introduzir o leitor no domínio do Big Data, apresentando um pouco da história por trás dessa palavra tão repercutida atualmente. Também iremos analisar algumas das principais tecnologias que foram criadas com o principal objetivo de fornecer recursos para se trabalhar com os conjuntos de dados que podem ser caracterizados como sendo Big Data.

Big Data

Big Data refere-se às tecnologias e iniciativas que envolvem conjuntos gigantescos de dados que são diversificados, complexos de armazenar, transportar e/ou analisar a partir do uso de ferramentas ou métodos de processamento de dados tradicionais. O volume desses conjuntos de dados é tão grande que desafia as tecnologias atuais e nos encoraja a construir a próxima geração de sistemas de armazenamento e processamento de dados.

A partir do Big Data as organizações estão se tornando cada vez mais capazes de criar novos produtos e com isso ultrapassar seus concorrentes. Além disso, seu domínio também ajuda as empresas a evitar gastos desnecessários, provendo-lhes a habilidade de analisar a grande quantidade de informações que foram coletadas durante toda a sua existência.

O modo como o Big Data está transformando o negócio das organizações foi descrito em um *white paper* publicado pela empresa 10gen, responsável pelo banco de dados NoSQL MongoDB. Este artigo relata as principais contribuições que estão sendo ocasionadas pelo Big Data. São elas:

- Construção de aplicações que até então eram impossíveis de serem desenvolvidas;
- Adaptar e desenvolver vantagens competitivas;
- Aumentar a satisfação dos clientes;
- Reduzir custos.

Mesmo que ainda hoje não haja uma definição concreta do que é Big Data, a mais conhecida é a publicada pelo grupo *Gartner*, que descreve em seu artigo que um conjunto de dados classificado como Big Data possui no mínimo uma das seguintes características:

- É formado por grandes volumes de dados;
- Lida com dados bastante variados quanto à sua forma e origem;
- Requer um rápido processamento de informações para atender a demanda.

Esta classificação ficou conhecida como os 3Vs: Volume, Velocidade e Variedade. Além destes, alguns pesquisadores especulam a existência de um quarto V, que seria o Valor para o negócio que a capacidade de lidar com Big Data pode trazer para as empresas.

Variedade

Quando lidamos com Big Data, geralmente os dados são provenientes de diversas fontes, como textos, sensores, arquivos de áudio, vídeo, logs, entradas de usuários, entre outras. Muitas vezes, essa multiplicidade na origem das informações acaba gerando dados com diversas estruturas e formatos.

Pensando nisso, quando trabalhamos com Big Data, devemos aceitar a entrada de dados estruturados, semiestruturados ou até mesmo não estruturados. Para poder manipular esta variedade, é necessário ter em mãos ferramentas capazes de processar dados de origens diversas e em qualquer formato.

Com toda essa variedade de dados, tornou-se foco para grandes empresas possuir aplicações capazes de explorar, de forma eficiente, estes dados diversificados, para que, independentemente de sua estrutura ou origem, elas consigam extrair algum valor que as auxilie em tomadas de decisões estratégicas.

Velocidade

No Big Data, velocidade refere-se tanto ao aumento significativo da taxa na qual os dados são produzidos em uma organização, quanto na rapidez com que os dados devem ser processados para atender a demanda.

Possuir uma informação precisa no momento errado não é vantagem para o negócio. Às vezes uma questão de segundos pode fazer com que o cliente procure outra maneira de obter o que estava procurando. Por exemplo, se o site de e-commerce de uma empresa A traz o resultado das buscas de seus produtos 10 segundos mais rápido do que o site de e-commerce de uma empresa B, o cliente pode preferir utilizar o site A apenas pela questão da velocidade da transação.

Quando lidamos com Big Data e precisamos de extrema rapidez na busca e processamento dos dados, podemos contar com diversas ferramentas que fazem parte do domínio do Big Data, como soluções MapReduce e bancos de dados NoSQL, que focam na rapidez do resultado de pesquisas sobre dados pré-processados, deixando de lado muitas características dos tradicionais SGBDRs (Sistema Gerenciador de Banco de Dados Relacional), como transação e schema pré-definido, para obter ganho de velocidade.

Volume

Big Data também envolve enormes volumes de dados. Estima-se que aproximadamente 2,3 trilhões de gigabytes de dados sejam gerados todos os dias por máquinas, sensores, usuários, entre outros. Em meados de 2020, espera-se que este valor passe para 40 zettabytes (43 trilhões de gigabytes) de informação criada diariamente, aumentando em 300 vezes o que tínhamos em 2005. Para mais informações sobre estes resultados veja a seção **Links**.

Entretanto, os sistemas de armazenamento convencionais, como os bancos de dados relacionais, não conseguem lidar muito bem com conjuntos de dados volumosos. Escalar os dados nesta arquitetura também ocasionaria gastos extensivos para a empresa, o que inviabilizaria o negócio.

Além do modo como é feito o armazenamento, também precisamos nos preocupar com a estratégia que será usada para processar e analisar estes grandes conjuntos de dados. Algumas opções que temos são: data warehouses, arquiteturas MPP (*Massively Parallel Processing*), bancos de dados como o Greenplum, ou soluções como o Apache Hadoop. No entanto, adotar soluções de data warehouse requer *schemas* pré-definidos, enquanto que com soluções de Big Data, como Apache Hadoop e bancos NoSQL, não é necessário impor condições na estrutura dos dados que serão processados.

Com toda essa necessidade ocasionada pelo Big Data, foram surgindo projetos que permitiam o manuseio de tanta informação. Um dos primeiros deles foi o BigTable, que após seu enorme sucesso se tornou uma das principais alavancas para a popularidade de bancos NoSQL que têm como base seu modelo de dados.

BigTable

Criado pelo Google e usado pelo GFS (*Google File System*) para gerenciar *petabytes* de informações, o BigTable é um sistema de armazenamento de dados distribuído que é estruturado como uma enorme tabela.

Atualmente esse projeto é utilizado em diversas aplicações do Google, principalmente naquelas que demandam alta capacidade de armazenamento e baixa latência de rede, como o processo de indexação de web sites e os produtos: Google Earth, Maps, Adwords, Analytics, Adsense, Youtube e Gmail. Para atender essas demandas, o BigTable provê uma solução flexível e de alta performance.

Extremamente escalável e tolerante a falhas, o BigTable foi projetado para facilitar a adição de novas máquinas ao cluster, sendo este um acontecimento comum em empresas de porte do tamanho do Google.

	usuario:nome	usuario:email	contato:telefone
5fbe8280-6850-11e3-949a-0800200c9a66	Andrew Martin T1	andre@martin.com T3	(11) 11223344 T1 (11) 22229900 T2

Figura 1. Pedaço de uma tabela que armazena dados de clientes

Esta ferramenta também possui uma arquitetura que permite escalar *petabytes* de dados separados em um ambiente de centenas ou milhares de máquinas, de forma automática, sem que seja necessária alguma reconfiguração.

O BigTable possui uma estrutura tabular com colunas dispersas que funciona como uma enorme tabela. Entretanto, diferente de uma tabela de um banco relacional, as linhas do BigTable não possuem esquema. Este tipo de modelo de dados é comumente referenciado como **Column-Family**, ou família de colunas, e serviu de influência para alguns dos bancos de dados NoSQL que surgiram algum tempo depois, como HBase e Cassandra.

Seu modelo de dados consiste em um mapa ordenado e multidimensional de pares chave/valor, onde a *chave* identifica a linha e o *valor* é um conjunto de colunas. Cada item do mapa é indexado por três colunas principais, sendo elas: **row key**, **column key** e **timestamp**. São as *row keys* que mantêm os dados ordenados, e é apenas através delas que pode ser feito o acesso aos dados. Já as *column keys* representam os identificadores das colunas, e são agrupadas em conjuntos denominados *Column Families*.

Além dessas características, as células de cada uma das linhas dentro do BigTable podem possuir múltiplas versões de um mesmo dado. Isto possibilita que a versão desejada seja especificada na recuperação dos dados. O responsável por controlar esse versionamento é a coluna denominada *timestamp*.

A Figura 1 ilustra um exemplo de um pedaço de uma tabela que armazena dados de clientes. A primeira coluna é a **row**

key, onde utilizamos um UUID como id. Nessa figura também podemos observar três outras colunas, onde cada uma possui uma **column key** e são agrupadas em duas famílias de colunas, “*usuario:*” e “*contato:*”. Por fim, cada valor contido nas colunas está marcado com um T junto com um número. Esta marcação representa o **timestamp** daquele conteúdo, ou seja, no caso do e-mail, por exemplo, o valor já está em sua terceira revisão.

Por este ser um banco de dados sem esquema (*schemaless*), os dados armazenados na parte *value* do mapa são arrays de bytes não interpretados pelo BigTable. Desta forma, a aplicação que for acessar esse sistema de armazenamento fica encarregada de ter a inteligência de como os dados devem ser tratados. Isto permite aos clientes um controle dinâmico sobre a forma e layout dos dados.

Contudo, apenas uma ferramenta de armazenamento de dados não foi o suficiente para lidar com tanta informação. Pensando nisso, com o objetivo de aproveitar todo o poder de seu ambiente clusterizado, o Google desenvolveu o projeto MapReduce.

MapReduce

Com a propagação do uso de clusters nas empresas, fizeram-se necessárias algumas mudanças no modo como os dados eram armazenados e processados. Empresas como Google e Yahoo! tiveram grande importância no desenvolvimento de técnicas, modelos, ferramentas e algoritmos capazes de aumentar a eficiência no processamento dessa grande quantidade de informação espalhada por diversas máquinas.

O modelo de programação MapReduce, criado pelo Google, combina duas estruturas que já existiam em linguagens de programação funcional: *map* e *reduce*. Esta combinação tem como objetivo oferecer uma maneira de se atingir o máximo de aproveitamento de todas as máquinas do cluster durante o processamento paralelo de grandes quantidades de dados de um sistema distribuído.

A técnica MapReduce funciona separando o processamento dos dados em duas fases: *map* e *reduce*, como expõe o próprio nome. Tomando o framework

Hadoop como exemplo, durante a fase *map*, o processamento distribuído começa no nó principal da rede, denominado *master*. Este recebe pares de chave/valor como entrada, que são divididos em problemas menores durante a execução da implementação do *map*. Por fim, o resultado é distribuído para os outros nós participantes.

O framework Apache Hadoop disponibiliza uma API escrita em Java para trabalharmos com a sua implementação de MapReduce. Nesta API, a função *map* é representada pela classe **Mapper**, a qual

declara o método abstrato *map()*. Este método espera como parâmetros uma chave, que poderia ser, por exemplo, o índice de uma linha de um suposto arquivo de texto CSV, e o valor, que iria conter o texto encontrado na linha daquele índice. A saída gerada por essa função também é composta por pares de informação formados pela dupla chave/valor.

Assim que a fase *map* é finalizada, o Hadoop coleta, classifica e agrupa todas as chaves que são idênticas, e em seguida, a função *reduce*, representada pela implementação do método abstrato *reduce()*, declarado na classe **Reducer**, recebe cada chave com a coleção de todos os valores relacionados a ela. Por fim, os valores são combinados pelas suas chaves em uma única saída de chave/valor, de modo que tenhamos a resposta para o problema original que estávamos tentando resolver, como por exemplo, a contagem de palavras idênticas de um arquivo de texto.

A **Figura 2** ilustra o funcionamento do *map*. Neste exemplo, a função *map* recebe como entrada as ordens de compra de um site de e-commerce no formato JSON. Após sua execução, são gerados como saída pares de chave-valor, onde a chave é o tipo do produto e o valor é composto pelos campos *quantidade* e *total*.

A **Figura 3** ilustra o funcionamento do *reduce*. Esta função recebe como entrada todos os pares chave/valor gerados pela função *map*, que neste cenário seriam as chaves: *jogo* e *livro*. Após a execução do *Reducer*, as entradas são reduzidas a uma única saída de chave-valor para cada chave recebida, resultando no valor total de vendas por cada tipo de produto.

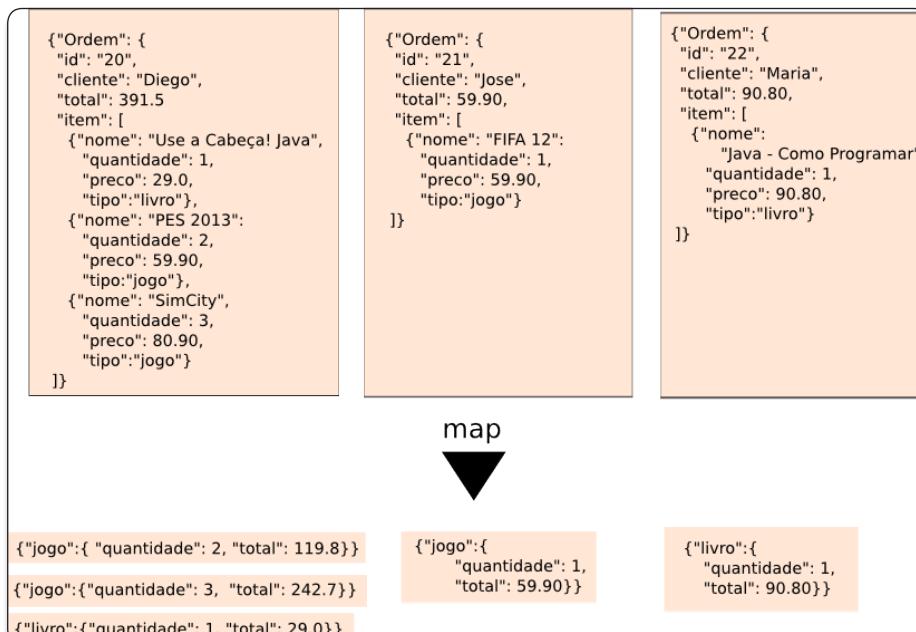


Figura 2. Ilustração da fase map

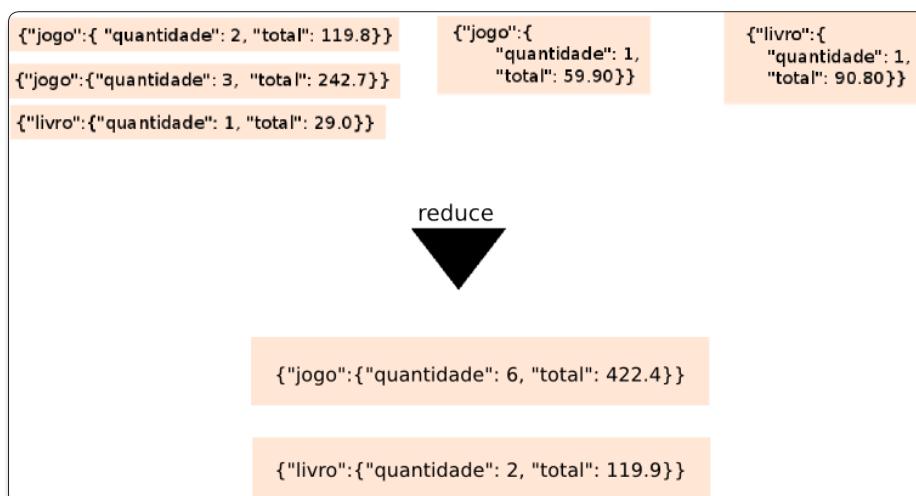


Figura 3. Ilustração da fase reduce

Apache Hadoop

Com a necessidade do desenvolvimento de aplicações capazes de realizar o processamento de grandes volumes de dados em paralelo sobre vários nós de um cluster, foram criadas várias bibliotecas com implementações do MapReduce. Entre elas, a mais popular é a implementada pelo Hadoop, um framework open source da Apache desenvolvido em Java que permite o trabalho com Big Data distribuindo o

processamento dos dados entre os nós do cluster através do uso de técnicas do MapReduce.

O desenvolvimento do Apache Hadoop teve uma forte base em dois projetos do Google, sendo eles o MapReduce e Google File System (GFS). Essa biblioteca foi projetada para entregar um serviço escalável e de alta disponibilidade em um sistema de computadores distribuídos.

Como exemplo, para fazer a contagem do número de ocorrências de cada palavra em uma determinada entrada de dados no Hadoop, além de algumas configurações básicas para o seu funcionamento, bastaria apenas que implementássemos as operações map e reduce. Desta forma, o código ficaria parecido com o da **Listagem 1**, o qual foi retirado do site oficial da Apache.

Listagem 1. Implementação feita em Java das operações map e reduce no Hadoop.

```
public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);
    while (tokenizer.hasMoreTokens()) {
        word.set(tokenizer.nextToken());
        output.collect(word, one);
    }
}

public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
    int sum = 0;
    while (values.hasNext()) {
        sum += values.next().get();
    }
    output.collect(key, new IntWritable(sum));
}
```

Embora o Hadoop seja mais reconhecido pelo MapReduce e por seu sistema de arquivos distribuído (HDFS), seu ecossistema é composto por quatro módulos principais que dividem as funções de armazenamento, processamento e agendamento. São eles:

- **Hadoop Common:** Conjunto de utilitários que dão suporte (serialização, Java RPC, estruturas de dados persistentes) para os demais módulos do projeto;
- **Hadoop Distributed File System (HDFS):** Este é o sistema de arquivos do Hadoop. Baseado no Google File System (GFS), fornece alto desempenho para acesso aos dados da aplicação;
- **Hadoop YARN:** Framework utilizado para agendamento de tarefas e gerenciamento dos recursos do cluster;
- **Hadoop MapReduce:** Baseado no YARN, este framework é responsável pelo processamento paralelo de grandes volumes de dados.

NoSQL

Além de conhecer as poderosas ferramentas de processamento de dados como o Hadoop, também é necessário conhecer os sistemas de armazenamento que foram preparados para lidar com

Big Data. Desta forma, nos tornamos capazes de escolher a opção que trará mais benefício em cada tipo de cenário.

O surgimento do BigTable (Google) e do Dynamo (Amazon) inspirou diversos projetos experimentais como alternativas ao armazenamento de dados voltados para o Big Data. Um destes projetos foi o banco Strozzi NoSQL, que tinha como principais características ser de código aberto e não expor uma interface SQL.

O termo NoSQL, ou Not Only SQL, foi usado em 1998 por Carlo Strozzi para referenciar seu banco de dados, o Strozzi NoSQL. Desde então, este termo é comumente utilizado para mencionar os sistemas de armazenamento que possuem características parecidas com o Strozzi NoSQL, e que por isso não se encaixam nas características compartilhadas pelos bancos de dados relacionais.

Com o objetivo de oferecer alguma ferramenta para o manuseio de seus dados, alguns bancos NoSQL implementam linguagens de consulta bem parecidas com o SQL, como o CQL do Cassandra. Entretanto, muitos destes bancos de dados implementam uma linguagem específica de seu domínio, como no caso da Cypher, uma linguagem de consulta própria para lidar com grafos e que apenas pode ser utilizada dentro do Neo4J.

Outra característica dos bancos de dados NoSQL é que grande parte deles é direcionada para rodar em clusters. Para isto, alguns desses bancos lidam com as transações de seus dados de maneira diferente dos bancos de dados relacionais, onde as transações são feitas de forma ACID, ou seja, Atômica, Consistente, Isolada e Durável.

No universo dos bancos não relacionais, na maioria dos casos o modo como a transação pode ser feita é particular a cada um destes bancos. Por exemplo, enquanto o MongoDB garante a consistência por documento inserido em sua base, o banco Neo4J suporta transações ACID. Por isso, em um ambiente clusterizado, o MongoDB tem um desempenho muito superior ao Neo4J.

Mesmo com tantos casos de sucesso no mercado, ainda hoje muitas companhias possuem bastante receio de colocar em produção bancos de dados que não sejam os tradicionais SGBDR. Geralmente a principal motivação para a escolha da utilização de um banco NoSQL é resolver o problema de escalabilidade dos bancos tradicionais, que na grande maioria dos casos é algo muito caro e/ou complexo. Outras características que influenciam na escolha pelo uso de bancos NoSQL são:

- Lidar com grandes volumes de dados;
- Esquema flexível;
- Escalabilidade;
- Alta disponibilidade e performance;
- Ambiente clusterizado;
- Rapidez no armazenamento e/ou acesso a dados distribuídos.

Além das características supracitadas, os bancos de dados NoSQL são classificados de acordo com o modo como é feito o armazenamento de seus dados. Os principais tipos de bancos NoSQL que temos hoje em dia são: Key-Value, Column-Family, Document-oriented e Graph. A seguir, veremos mais detalhes

sobre cada um destes, informando como os dados são tratados e também alguns cenários nos quais eles seriam bem aplicados.

Bancos de dados Chave/Valor (Key-Value)

Riak, Redis e DynamoDB são alguns dos principais bancos de dados NoSQL do tipo chave-valor, sendo que desses três, apenas o DynamoDB, da Amazon, não é open source. Esse tipo de banco de dados funciona como tabelas de hash, sendo que alguns deles, como o Redis, permitem que os dados sejam armazenados em outros tipos de estruturas de dados, como **Lists** e **Sets**.

Uma **hash table** é um agregado de duas colunas, sendo uma para a chave e outra para o valor. A chave (ou *key*) é um identificador único, a qual o banco utiliza para fazer a indexação. O valor (ou *value*) é armazenado como um BLOB, em outras palavras, uma grande **String**. O banco armazena esse BLOB sem manter nenhum conhecimento de sua estrutura. Desta forma, o entendimento do que está sendo armazenado e o que fazer para lidar com o valor ali contido se tornam de responsabilidade da aplicação que irá consultar esses dados.

No Riak, por exemplo, os dados são armazenados em buckets, que podem ser comparados às tabelas de um banco de dados relacional. Deste modo, se quiséssemos armazenar a sessão de um usuário, as informações do seu carrinho de compras, seus produtos preferidos e o log da aplicação, poderíamos fazer isso adicionando todos esses dados em um único bucket, chamado, por exemplo, de **dadosUsuario**.

Em uma aplicação Java poderíamos simplesmente serializar um objeto contendo todas essas informações, armazenar o valor dessa serialização na coluna *value*, e utilizar o ID da sessão como a chave. Como outro exemplo, poderíamos transformar o objeto em XML ou JSON e armazenar a **String** gerada como valor. A **Listagem 2** apresenta um exemplo de código Java que armazena logs do sistema em um banco Redis. Observe que com apenas duas linhas foi possível abrir uma conexão com o banco, acessar a lista *log* e adicionar a ela uma **String** concatenada com um timestamp, representando um log de uma aplicação qualquer.

Ao fazermos uma consulta nessa lista, temos uma visualização parecida com a apresentada na **Figura 4**. Nela, utilizamos o comando **LRANG** do Redis, passando como argumento a chave *log* e um range de registros de 1 até 10. Como resultado, são exibidos todos os valores encontrados entre os índices 1 e 10 na lista *log*.

Como este tipo de sistema de armazenamento só possibilita o acesso aos dados por meio da chave, eles são usados apenas em cenários nos quais todos os acessos possam ser feitos a partir de uma chave primária. Esta característica faz com que os bancos chave/valor possuam alta performance e escalem com facilidade.

Alguns casos onde esses bancos de dados podem ser muito bem aplicados são em situações em que buscamos uma maneira simples de armazenar informações de sessão, preferências e perfis de usuários, dados do carrinho de compras e logs de sistema, sem ter que lidar com certas complexidades providas pelos modelos de dados dos bancos relacionais.

```
redis 127.0.0.1:6379> LRANGE log 1 10
1) "Log da aplicacao: Wed Jul 10 23:08:41 BRT 2013"
2) "Log da aplicacao: Wed Jul 10 23:08:41 BRT 2013"
3) "Log da aplicacao: Wed Jul 10 23:08:40 BRT 2013"
4) "Log da aplicacao: Wed Jul 10 23:08:39 BRT 2013"
5) "Log da aplicacao: Wed Jul 10 23:08:38 BRT 2013"
6) "Log da aplicacao: Wed Jul 10 23:08:37 BRT 2013"
redis 127.0.0.1:6379>
```

Figura 4. Exemplo de lista de logs armazenada no Redis.

Listagem 2. Exemplo de acesso ao banco Redis com a utilização do framework Jedis.

```
Jedis jedis = new Jedis("localhost");
jedis.lpush("log","Log da aplicacao:" + Calendar.getInstance().getTime());
```

Bancos de dados orientados a Documentos

O principal propósito dos bancos de dados orientados a documentos é armazenar documentos, que podem ser arquivos JSON, XML, BSON, entre outros. Os documentos são guardados na parte *value* de um sistema de armazenamento do tipo chave/valor. Pense neste tipo de banco de dados como um banco do tipo chave/valor onde o valor é examinável, ou seja, podemos realizar consultas sobre o conteúdo que está no *value*. Para fazer as buscas e recuperar as informações, são utilizadas técnicas como MapReduce e frameworks de agregação.

Como os bancos de dados orientados a documentos não possuem esquema, os documentos que são armazenados não precisam possuir uma estrutura comum. Por esta característica, esse tipo de sistema de armazenamento é uma ótima opção para trabalhar com dados semiestruturados.

Ainda por ter um esquema flexível, os documentos arquivados podem ou não possuir a mesma estrutura, mesmo quando armazenados na mesma coleção. Estas coleções podem ser comparadas com as tabelas dos bancos de dados relacionais, e os documentos são como um registro, ou tupla, de uma tabela. No entanto, como informado, diferentemente de um SGBDR, onde cada registro de uma mesma tabela tem a mesma sequência de campos, em um banco de dados orientado a documentos uma mesma coleção pode ter diversos documentos com estruturas diferentes.

O MongoDB, um dos bancos orientados a documentos mais conhecidos, armazena arquivos no formato BSON, que de forma simples pode ser definido como uma versão binária de um arquivo JSON. A **Listagem 3** apresenta o resultado de uma consulta feita no MongoDB, onde é exibida uma coleção contendo dois documentos. Observe que ambos possuem atributos em comum, como *nome* e *revista*. Porém, suas estruturas divergem no atributo *empresa*, contido apenas no primeiro documento.

Bancos de dados Família de Colunas (Column-Family)

Com fortes heranças de características do BigTable, bancos do tipo Família de Colunas, como HBase, Amazon SimpleDB e Cassandra, também são bem parecidos com os bancos do tipo

Analisando o Big Data na teoria e na prática

key-value (chave/valor). A diferença é que na coluna onde são guardados os valores, temos um segundo nível de mapeamento chave-valor. Isto permite que sejam armazenados valores mais complexos.

Listagem 3. Exemplo de uma coleção de documentos JSON.

```
{  
    nome: "Desmistificando o Big Data",  
    revista: "Java Magazine",  
    empresa: "Devmedia",  
    numero_edicao: 100  
}  
  
{  
    nome: "Desmistificando o Big Data",  
    revista: "Java Magazine",  
    numero_edicao: 100  
}
```

Nesta solução, cada família de colunas é um agrupamento de dados relacionados, ou seja, aqueles que são geralmente acessados conjuntamente. As famílias de colunas são análogas às tabelas dos bancos de dados relacionais, e são agrupadas em um container denominado keyspace. Um keyspace é similar a um banco de dados em um SGBDR, e tipicamente é usado apenas um por aplicação.

A **Listagem 4** apresenta uma família de colunas no formato JSON. Nela temos dois registros de pedidos, onde cada um forma uma row (linha) e possui um atributo *name* como identificador único. Tomando o segundo pedido como exemplo, no primeiro nível de chave-valor, encontramos a row com a chave “pedido:99820AAC”. Na coluna *value*, temos um segundo nível de chave/valor, contendo as colunas *cliente* e *endereco_entrega*.

Observe que a coluna *endereco_entrega* possui algumas outras colunas em seu conteúdo. Quando uma coluna consiste de um mapa de colunas, ela é chamada de **super coluna**. Pense nas super colunas como sendo containers de colunas, onde podemos ter, recursivamente, infinitos níveis de chave/valor.

Bancos de dados orientados a Grafos

Baseado na teoria dos grafos, um banco orientado a grafos foca na conexão entre os dados. Estes bancos permitem que sejam armazenadas nós e os relacionamentos entre eles. Desta forma, podemos encontrar padrões interessantes entre os nós que estão interligados.

Os conceitos principais deste tipo de banco de dados são os nós (*nodes*) e as arestas (*edges*). Os nós representam as entidades, como uma pessoa, um produto ou um pedido. Contudo, são suas propriedades, como nome, endereço e telefone, que definem, que dão um significado para o nó no sistema. O relacionamento entre as entidades é feito através das arestas, que além de poder possuir propriedades, possibilitam que padrões sejam encontrados, como por exemplo, as empresas em comum em que dois funcionários trabalharam, ou os filmes em comum que dois amigos gostam.

A **Figura 5**, retirada da rede social LinkedIn, apresenta um grafo que ilustra três relacionamentos entre dois nós. Nesta imagem podemos observar que os nós representam dois perfis de cadastros de pessoas, e as arestas mostram a quantidade de localizações, grupos e companhias que ambos os nós possuem em comum.

Listagem 4. Representação JSON de uma Família de Colunas.

```
//Column-Family  
{  
    //row  
    {  
        name: "pedido:99820AAB",  
        value: {  
            cliente: "Marcos Silva",  
            endereco_entrega: {  
                uf: "SP",  
                rua: "Av. Paulista",  
                numero: 999999999  
            }  
        },  
        timestamp: 12345667891  
    }  
    //row  
    {  
        name: "pedido:99820AAC",  
        value: {  
            cliente: "José do Andrade",  
            endereco_entrega:  
            {  
                uf: "SP",  
                rua: "Av. Paulista",  
                numero: "9999 AB",  
                complemento: "CJ-YXZ"  
            }  
        },  
        timestamp: 12345667895  
    }  
}
```

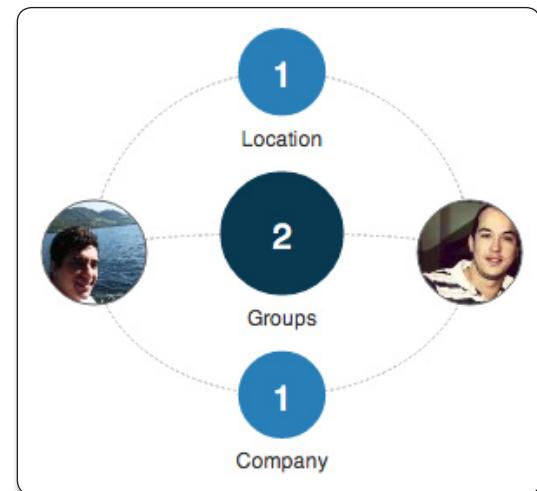


Figura 5. Grafo ilustrando relacionamentos entre duas pessoas

O Neo4J é um dos bancos mais populares entre os orientados a grafos. Ele oferece uma linguagem de consulta própria, chamada Cypher, que possibilita uma fácil manipulação de seus grafos.

Como exemplo, a criação de dois nós e uma relação entre eles pode ser feita de uma maneira bastante simples, conforme expõe a **Listagem 5**. Com a execução deste código teríamos um grafo semelhante ao ilustrado na **Figura 6**.

A maior motivação para a utilização de um banco de dados orientado a grafos é quando queremos encontrar padrões entre os relacionamentos das entidades. Por esta característica, estes bancos são usados principalmente em sistemas de recomendações e em redes sociais.

Listagem 5. Exemplo de criação de dois nós e uma aresta bidirecional com Neo4J.

```
Node diego = graphDb.createNode();
diego.setProperty("nome", "Diego");

Node tiago = graphDb.createNode();
tiago.setProperty("nome", "Tiago");

diego.createRelationshipTo(tiago, AMIGO);
tiago.createRelationshipTo(diego, AMIGO);
```

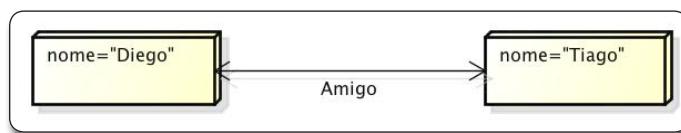


Figura 6. Grafo ilustrando dois nós com uma relação bidirecional

Devido à deficiência no manuseio de grandes volumes de dados das ferramentas convencionais de processamento de dados, foi necessário evoluir o modo como trabalhamos com os dados. Graças à influência de empresas renomadas na área de tecnologia, como Google, Yahoo!, Amazon e Apache, cada vez mais estão surgindo ferramentas mais eficientes para se trabalhar com Big Data, como Hadoop e algumas soluções NoSQL.

Estas ferramentas que já são desenvolvidas orientadas para o Big Data têm auxiliado empresas de diversos ramos a armazenar, processar e olhar para seu vasto conjunto de dados, e também a extrair visões analíticas do negócio e colocá-las à frente de suas concorrentes.

Além disso, a propagação do Big Data também tem gerado mais profissões no mercado, como cientista de dados, e renovado algumas delas, como os arquitetos e os engenheiros de dados. O trabalho realizado por estes profissionais seria, por exemplo, implementar o framework Hadoop, ou então, analisar dados armazenados em bancos NoSQL.

Mesmo com tanta ferramenta preparada para lidar com Big Data no mercado, ainda hoje diversas empresas sofrem com seus gigantescos conjuntos de dados, tendo que investir uma grande

quantidade de dinheiro para poder trazer algum valor para o negócio com a mineração das informações contidas em seus bancos de dados. Isto acontece porque muitas dessas empresas têm certo receio das novas tecnologias e preferem optar por aquilo que já está mais consolidado.

Para concluir, é preciso ressaltar que o Big Data marca o início de uma grande transformação. A possibilidade do manuseio de coisas que poderiam nunca ser medidas, armazenadas, analisadas e compartilhadas está chegando ao fim. Com as ferramentas voltadas para Big Data em mãos, a oportunidade de lidar com vastas quantidades de dados, ao invés de apenas uma pequena porção, abrirá as portas para entendermos melhor o mundo.

Autor



Diego Travassos Balduini

diego.balduini@yahoo.com.br

É MBA em Engenharia de Software pela FIAP, trabalha com Java há quatro anos e possui a certificação SCJP.



Links:

Site oficial do projeto Apache Hadoop.

<http://hadoop.apache.org/>

Site oficial do banco Neo4J.

<http://www.neo4j.org/>

Site oficial do banco Redis.

<http://redis.io/>

Site oficial do banco MongoDB.

<http://www.mongodb.org/>

Publicação do Google sobre BigTable.

<http://research.google.com/archive/bigtable.html>

White paper sobre Big Data.

<http://www.mongodb.com/dl/big-data-front>

Publicação do grupo Gartner sobre os 3Vs.

<http://blogs.gartner.com/doug-laney/files/2012/01/ad949-3D-Data-Management-Controlling-Data-Volume-Velocity-and-Variety.pdf>

Publicação do Google sobre MapReduce.

<http://research.google.com/archive/mapreduce.html>

Fonte de informações sobre os 3Vs.

<http://dashburst.com/infographic/big-data-volume-variety-velocity/>

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

Ele está trabalhando em seu saque e não na sua nova aplicação.



Mais de 95%* dos times de desenvolvimento e testes de aplicativos relatam tempos de espera e atrasos para acessar os sistemas que necessitam para realizar suas atividades. A Virtualização de Serviços elimina estas dependências gerando ambientes simulados similares à realidade, permitindo o desenvolvimento em paralelo. Isto significa que suas aplicações serão lançadas mais rapidamente, com maior qualidade e menor custo. Game over!

Conheça mais em ca.com/br/GoDevOps

ca
technologies

*De acordo com o estudo 2012 North America/Europe Service Virtualization Study, Coleman-Parkes.