Desenvolvimento Desktop

Aula 13 - Arquivos – Lendo, Escrevendo e Criando - Parte 1

Material Didático do Instituto Metrópole Digital - IMD
Versão 5.0 - Todos os Direitos reservados

Apresentação

Esta aula trata de um assunto introduzido no JDK (Java Development Kit) versão 7 e trata dos novos pacotes de acesso ao sistema de arquivos. O pacote **java.nio.file** é o novo responsável pela entrada e saída de dados em arquivos, assim como para acesso a todo o sistema de arquivos do sistema operacional. Isso inclui operações de cópia, remoção, acesso a arquivos e diretório, entre outras coisas.

Esse pacote, apesar de conter muitas classes, tem poucos pontos importantes que devem ser aprendidos. No geral, é uma API intuitiva e de fácil utilização.

Vídeo 01 - Apresentação

Objetivos

Ao final desta aula, você será capaz de:

- Definir um path.
- Aplicar um path com a classe Path.
- Distinguir caminho absoluto e caminho absoluto.
- Estabelecer como criar e converter um Path.
- Distinguir comparar dois Paths.
- Apagar, copiar e mover arquivos e diretórios etc.

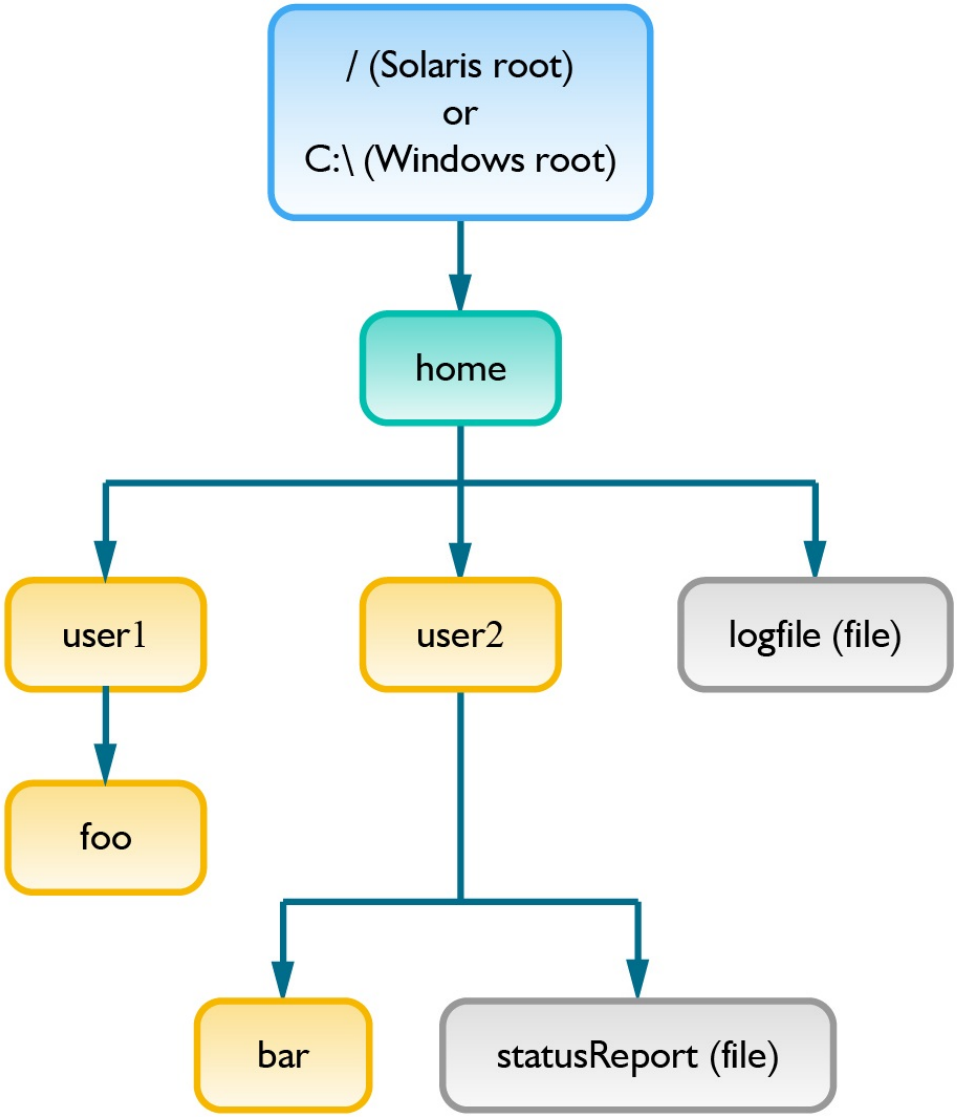
Uma Breve Introdução sobre Arquivos

Instâncias da classe java.nio.file representam caminhos (**paths**) para possíveis locais no sistema operacional. Lembre-se de que ele apenas representa um arquivo ou diretório, isto não quer dizer que esses caminhos existam ou não.

Path (caminho)

O primeiro conceito importante a se aprender é o conceito de **path** (ou caminho), um objeto que pode ser usado para localizar um arquivo em um sistema de arquivos. Um **path** representa um caminho que é hierárquico e composto por uma sequência de elementos de diretório e pelo nome do arquivo, separados por um separador ou delimitador especial. Um caminho pode representar uma raiz (root), uma raiz e uma sequência de nomes, ou simplesmente um ou mais elementos de nomes. A Figura 1 mostra uma árvore de diretórios contendo apenas um nó raiz. O sistema operacional Windows suporta múltiplos nós raiz. Cada nó raiz é mapeado para um volume, C:\ ou D:\. O sistema operacional Solaris permite apenas um nó raiz, que é representado pelo caractere barra (/).

Figura 01 - Amostra de uma estrutura de diretórios



Um arquivo é identificado pelo seu **path** (ou caminho) em um sistema de arquivos, começando pelo nó raiz. Por exemplo, o arquivo **statusReport**, na Figura 1, é representado pela seguinte notação no Solaris OS:

/home/user2/statusReport.

O mesmo arquivo no Windows OS tem a seguinte representação:

C:\home\user2\statusReport.

O caractere utilizado para separar os nomes dos diretórios (delimitador) é específico do sistema operacional. O Solaris OS, por exemplo, usa a barra (/), enquanto que o Windows usa a contrabarra (\).

Caminho Relativo ou Absoluto?

O **path** pode ser relativo ou absoluto. Um **path** absoluto sempre contém o elemento root e a lista completa de diretórios necessários para localizar o arquivo. Por exemplo, o caminho **/home/user2/statusReport** é um caminho absoluto. Ou seja, toda a informação necessária para localizar o arquivo está contida no caminho.

Um caminho relativo precisa ser combinado com outro caminho para que seja possível acessar um arquivo. Por exemplo, **jose/fotos** é um caminho relativo. Sem informação adicional não é possível localizar esse diretório no sistema de arquivos.

A classe Path

Como seu próprio nome implica, a classe **Path** é uma representação de um caminho no sistema de arquivos. Um objeto **Path** contém o nome do arquivo e a lista de diretórios usados para construir o caminho, e é usado para localizar e manipular arquivos. Uma instância da classe **Path** deve refletir as necessidades da plataforma na qual está executando. No Solaris OS, por exemplo, um objeto da classe **Path** usa a sintaxe do Solaris **(/home/joe/foo)**, enquanto que no Windows o mesmo objeto usa a sintaxe do Windows (**C:\home\joe\foo**). É importante perceber que a classe **Path** não é independente de plataforma. Não se deve esperar que um objeto da classe **Path** que está em execução no Solaris OS funcione no Windows, por exemplo; mesmo que a estrutura de diretórios seja idêntica e nas duas instâncias ela aponte para o mesmo arquivo.

O arquivo ou diretório correspondente ao **Path** pode até nem existir. Você pode criar uma instância da classe **Path** e manipulá-la de várias maneiras: pode acrescentá-la, extrair partes dela ou compará-la com outro caminho, por exemplo. No momento apropriado, você poderá utilizar os métodos da classe **Files** para verificar a existência do arquivo correspondente ao **Path**, criar o arquivo, abri-lo, excluí-lo, alterar suas permissões, e assim por diante.

Criando um Path

Um objeto da classe **Path** pode ser facilmente criado com o auxílio da classe **Paths** (no plural) através do método estático **get**, como nos exemplos abaixo:

Path p1 = Paths.get("/tmp/foo/myfile.txt");

Path p2 = Paths.get("c:\\data\\myfile.txt");

O exemplo a seguir cria um caminho **/home/jose/logs/teste.log**, assumindo que seu diretório padrão é **/home/jose**, ou **C:\jose\logd\teste.log**, se você estiver no Windows.

Path p3 = Paths.get(System.getProperty("user.home"),"logs", "teste.log");

Vídeo 02 - Arquivos

Obtendo Informações sobre um Path

Um objeto da classe **Path** também dispõe de métodos, de forma que você possa obter informações acerca do caminho especificado. Os diretórios, por exemplo, são armazenados em sequência. O mais alto diretório na estrutura é referenciado pelo índice 0 (zero), enquanto que o mais baixo elemento é referenciado pelo índice (n – 1), onde n é o número de elementos no caminho (**Path**). Os métodos fornecem tanto elementos individuais quanto subsequências do **Path**, utilizando esses índices.

Os exemplos desta aula usarão a estrutura de diretórios mostrada na Figura 1. O trecho de código a seguir define uma instância de **Path** e, então, invoca vários métodos para obter informação sobre o path.

```
Path path = Paths.get("C:\\Users\\Diego\\Documents\\NetBeansProjects\\MD Java Nio2\\arquivo.txt
System.out.format("toString: %s%n", path.toString());
System.out.format("getFileName: %s%n", path.getFileName());
System.out.format("getName(0): %s%n", path.getName(0));
System.out.format("getNameCount: %d%n", path.getNameCount());
System.out.format("subpath(0,2): %s%n", path.subpath(0, 2));
System.out.format("getParent: %s%n", path.getParent());
System.out.format("getRoot: %s%n", path.getRoot());
```

A saída desta execução deve ser:

```
toString: C:\Users\Diego\Documents\NetBeansProjects\MD Java Nio2\arquivo.txt
getFileName: arquivo.txt
getName(0): Users
getNameCount: 6
subpath(0,2): Users\Diego
getParent: C:\Users\Diego\Documents\NetBeansProjects\MD Java Nio2
getRoot: C:\
```

A seguir, um pequeno comentário sobre cada método:

- **toString** – retorna a representação do Path em string.
- **getFileName** – retorna o nome do arquivo ou o último elemento da sequência de elementos.
- **getName(0)** – retorna o elemento do Path correspondente ao índice especificado. O primeiro elemento (índice zero) é o elemento mais perto da raiz.
- **getNameCount** – retorna o número de elementos do Path.
- subpath(0, 2) – retorna a subsequência do Path (não incluindo o elemento raiz) como especificado pelos índices inicial e final.
- **getParent** – retorna o Path do diretório pai.
- **getRoot** – retorna a raiz do Path.

Convertendo um Path

Existem três métodos à disposição para se converter um **Path**. Se é necessário converter um **Path** para uma string que o determinado arquivo possa ser aberto em um navegador, deve-se usar o método **toUri**. Por exemplo:

```
Path p1 = Paths.get("/home/logfile");
System.out.format("%s%n", p1.toUri());
```

Cujo resultado da conversão é:

file:///home/logfile

O método **toAbsolutePath** converte um **Path** para seu correspondente caminho absoluto (a partir da raiz). Se o **Path** já for absoluto, ele retorna o mesmo objeto. Esse método é particularmente útil para tratar nomes de arquivos digitados por usuários. Por exemplo:

```
if (args.length < 1) {
    System.out.println("usage: FileTest file");
    System.exit(-1);
}
// Converte a string de entrada para um objeto Path.
Path inputPath = Paths.get(args[0]);
Path fullPath = inputPath.toAbsolutePath();
```

O método **toAbsolutePath** converte a entrada do usuário e retorna um **Path** que pode ser consultado, como o exemplo anterior, e retornar valores bastante úteis. O arquivo não precisa existir para que esse método funcione. O método **toRealPath** retorna o caminho real de um arquivo existente e método executa várias operações em uma:

- Se **true** é passado para esse método e o sistema de arquivos suporta links simbólicos, o próprio método os resolve e retorna o endereço real;
- Se o caminho (**Path**) é relativo, ele retorna um caminho absoluto;
- Se o caminho (**Path**) contém algum elemento redundante, ele os remove e retorna um Path limpo.

Esse método dispara uma exceção se o arquivo não existe ou não pode ser acessado. Pode-se capturar a exceção quando se deseja manipular esses casos. Por exemplo:

```
try {
    Path fp = path.toRealPath(true);
} catch (NoSuchFileException x) {
    System.err.format("%s: no such" + " file or directory%n", path);
    // Caso o arquivo não exista.
} catch (IOException x) {
    System.err.format("%s%n", x);
    // Outro erro qualquer.
}
```

Vídeo 03 - Arquivos: Copiar e Mover

Comparando Dois Paths

A classe **Path** suporta o método equals, possibilitando testar a igualdade entre dois **Paths**. Os métodos **startWith** e **endsWith** permitem saber se um **Path** começa ou termina com uma string em particular. Por exemplo:

```
Path path1 = Paths.get("arquivo.txt");
Path path = path1.toAbsolutePath();
if (path.equals(path1)) {
    System.out.println("Os caminhos são iguais!");
} else if (path.startsWith(beginning)) {
    System.out.println("Path começa com C:\\Users");
} else if (path.endsWith(ending)) {
    System.out.println("Path termina com Nio2");
}
```

A classe **Path** implementa a interface **Iterable**. O método **iterator** retorna um objeto que possibilita iteração pelos nomes dos elementos em um **Path**. O primeiro elemento retornado é o mais perto da raiz na árvore de diretórios. O seguinte trecho de código itera sobre um **Path**, imprimindo cada nome de elemento:

```
for (Path name: path) {
    System.out.println(name);
}
```

A classe **Path** também implementa a interface **Comparable**. Ou seja, pode-se comparar objetos dessa classe através do método **compareTo**. Pode-se ainda querer saber se dois caminhos (**Paths**) se referem ao mesmo arquivo. O método **isSameFile** pode ser usado em situações como essa. Ele é descrito em *Checking Whether Two Paths Locate the Same File*, cujo endereço eletrônico se encontra nas referências.

Atividade 01

1. O que vem a ser um path?
2. Que pacote é responsável pela entrada e saída de dados em arquivos e também acesso a todo o sistema de arquivos do Sistema Operacional?
3. O que você entende por caminho absoluto e caminho relativo?
4. Que método devemos utilizar para retornar o número de elementos do Path?

Verificando a Existência de um Arquivo ou Diretório

Vimos que uma instância da classe **Path** representa um arquivo ou um diretório, mas será que esse arquivo existe no sistema de arquivos? Ele pode ser lido? Escrito? Executado?

Isso pode ser feito através dos métodos **exists(Path, LinkOption...)** e **notExists(Path, LinkOption...)**. É importante perceber que **!Files.exists(path)** não é equivalente a **Files.notExists(path)**. Quando se está testando a existência de um arquivo, três resultados são possíveis:

- é verificada a existência do arquivo;
- é verificada a não existência do arquivo;
- o status do arquivo é não conhecido. Esse resultado pode ocorrer quando o programa não tem acesso ao arquivo.

Se ambos, **exists** e **notExists**, retornarem falso, a existência do arquivo não pode ser verificada.

Checando a Acessibilidade do Arquivo

Para verificar se um programa pode acessar um arquivo, pode-se usar os métodos **isReadable(Path)**, **isWritable(Path)** e **isExecutable(Path)**. O seguinte trecho de código verifica se um arquivo em particular existe e se o programa tem permissão para executá-lo.

```
Path path = Paths.get(""arquivo.txt"");
boolean arquivoExecutavelComum = Files.isRegularFile(path)
    && Files.isReadable(path) && Files.isExecutable(path);
System.out.println((arquivoExecutavelComum ? ""É um arquivo executável comum"" : ""Não é um a
```

Checando se Dois Paths Apontam para o Mesmo Arquivo

Quando se tem um sistema de arquivos que usa links simbólicos, é possível ter dois diferentes caminhos (**paths**) que representam o mesmo arquivo. O método **isSameFile(Path, Path)** compara se dois caminhos representam o mesmo arquivo, por exemplo:

```
Path p1 = ...;
Path p2 = ...;

if (Files.isSameFile(p1, p2)) {
    System.out.println(“Os caminhos apontam para o mesmo arquivo!”);
}
```

Vídeo 04 - Arquivos: Remoção

Apagando um Arquivo ou Diretório

Pode-se apagar arquivos, diretórios ou links. Com links simbólicos, apenas o link é apagado e não o arquivo cujo link aponta. Em se tratando de diretórios, o mesmo deve estar vazio, ou a operação falha.

A classe **Files** provê dois métodos de remoção. O **delete(Path)** apaga o arquivo ou dispara uma exceção se a operação falhar. Por exemplo, se o arquivo não existir, uma exceção **NoSuchFileException** é disparada. De qualquer forma, é possível capturar a exceção e saber por que a operação de remoção falhou, como mostra o trecho de código a seguir:

```
try {
    Files.delete(path);
} catch (NoSuchFileException x) {
    System.err.format("%s: no such" + " file or directory%n", path);
} catch (DirectoryNotEmptyException x) {
    System.err.format("%s not empty%n", path);
} catch (IOException x) {
    // File permission problems are caught here.
    System.err.println(x);
}
```

O método **deleteIfExists(Path)** também apaga o arquivo, mas, se o arquivo não existir, nenhuma exceção é disparada.

Copiando um Arquivo ou Diretório

Pode-se copiar um arquivo ou diretório através do método **copy(Path, Path, CopyOption...)**. A cópia falha se o arquivo destino já existir, a menos que a opção **REPLACE_EXISTING** seja especificada.

Diretórios podem ser copiados, mas os arquivos dentro do diretório não o serão. Dessa forma, os novos diretórios serão vazios mesmo que os originais contenham arquivos.

Quando se copia um link simbólico, o arquivo apontado pelo link é copiado. Se a intenção for copiar o link em si e não o conteúdo do link, deve-se especificar a opção **NOFOLLOW_LINKS** ou **REPLACE_EXISTING**.

Esse método (**copy(Path, Path, CopyOption...))** recebe parâmetros onde os seguintes argumentos são suportados:

- **REPLACE_EXISTING** — executa a cópia mesmo se o arquivo destino já existe.
- **COPY_ATTRIBUTES** — copia os atributos associados ao arquivo para o arquivo destino. Os atributos suportados dependem da plataforma e do sistema de arquivos, mas o atributo last-modified-time é suportado entre plataformas e é copiado ao arquivo destino.
- **NOFOLLOW_LINKS** — indica que links simbólicos não devem ser seguidos. Se o arquivo a ser copiado é um link, o próprio link será copiado (e não arquivo cujo link aponta).

O trecho de código a seguir mostra como o método **copy** deve ser utilizado:

Files.copy(source, target, REPLACE_EXISTING);

Onde os dois primeiros argumentos são o arquivo de origem e o de destino, respectivamente, ambos representados por objetos da classe **Path**. O terceiro argumento é um atributo que caracteriza a maneira como a operação vai ser realizada, nesse caso, executando a cópia mesmo que o arquivo de destino exista.

Para que esse método funcione com os atributos, é necessário importar o pacote **java.nio.file.StandardCopyOption.*.**

Movendo um Arquivo ou Diretório

Para mover arquivos ou diretórios, deve-se utilizar o método **move(Path, Path, CopyOptions...)**. Perceba a similaridade com o método **copy**, no qual os argumentos têm funcionalidades parecidas.

Diretórios vazios podem ser movidos e, caso eles não estejam vazios, o conteúdo não é movido junto com o diretório. Em sistemas UNIX, mover um diretório para a mesma partição significa renomeá-lo. Nessas situações, o método funciona mesmo quando o diretório contém arquivos.

Entre os atributos da operação, os suportados são os seguintes:

- **REPLACE_EXISTING** — executa a operação mesmo se o destino existe. Se o destino é um link simbólico, o link é substituído, mas o que ele aponta não é afetado.
- **ATOMIC_MOVE** — executa a operação atomicamente. Se o sistema de arquivos não suporta a atomicidade, uma exceção é disparada.

O trecho de código a seguir mostra como usar o método move:

Files.move(source, target, REPLACE_EXISTING);

Da mesma forma que no método **copy**, o pacote **java.nio.file.StandardCopyOption.*** precisa ser importado.

Atividade 02

---

1. Qual dos métodos abaixo retorna o nome do arquivo ou o último elemento da sequência de elementos?
    a. getName
    b. getFileName
    c. getNameCount
    d. getParent
2. Que método é utilizado para saber se dois caminhos representam o mesmo arquivo?
3. Os diretórios podem ser copiados, mas os arquivos dentro desse diretório não o serão. Eles serão vazios, mesmo que os originais contenham arquivos. Essa afirmativa é verdadeira ou falsa?

Conclusão

Por enquanto, é só. Na próxima aula, daremos continuidade ao estudo dos arquivos e você aprenderá a manipulá-los de forma correta e conhecerá mais alguns detalhes sobre eles. Até lá.

Resumo

Nesta aula, você aprendeu o que vem a ser um path (ou caminho), assim como criar um path nos sistemas Windows e Solaris utilizando a classe Path. Aprendeu também como funcionam os caminhos absolutos e relativos e conheceu o pacote do Java, responsável pela entrada e saída de arquivos. Aprendeu também a utilizar os métodos para converter e comparar Paths, como também métodos específicos para apagar, copiar e mover arquivos e diretórios. Na próxima aula, concluiremos o assunto sobre arquivos abordando mais alguns recursos, como a criação, leitura e escrita, utilizando suas respectivas classes e métodos. Até lá.

Autoavaliação

1. Que método da classe **Paths** é utilizado para criar um caminho?
2. Qual a forma correta para se criar um caminho usando a classe Path?
    a. Path p1 = Paths.get("/tmp/foo");
    b. Path p1 = PathsGet("/tmp/foo");
    c. Path p1 = Path.get("/tmp/foo");
    d. Path p1 = Paths.gets("/tmp/foo");

Referências

THE JAVA tutorials. **What Is a Path? (And Other File System Facts)**. Disponível em: <http://docs.oracle.com/javase/tutorial/essential/io/path.html>. Acesso em: 28 abr. 2012a.

_____. **The Path Class.** Disponível em: <http://docs.oracle.com/javase/tutorial/essential/io/pathClass.html>. Acesso em: 28 abr. 2012b.

_____. **Parth Operations** Disponível em: <http://docs.oracle.com/javase/tutorial/essential/io/pathOps.html>. Acesso em: 28 abr. 2012c.

_____. **Checking a File or Directory.** Disponível em: <http://docs.oracle.com/javase/tutorial/essential/io/check.html#same>. Acesso em: 28 abr. 2012d.