


Java Avançado

I/O com a NIO.2 API




Softblue
cursos online

Tópicos Abordados




- NIO.2 API x Legacy I/O API
- A interface *Path*
 - Manipulando objetos *Path*
 - Métodos úteis da classe *Paths*
- A classe *Files*
- Operações em arquivos e diretórios
 - Criar, excluir, copiar, mover, etc.
- Lendo e escrevendo dados
- Iterando sobre elementos de um diretório
 - Filtros glob
 - Filtros customizados
 - A interface *FileVisitor*
- Watch Service API
- Integração entre NIO.2 e Legacy I/O APIs

NIO.2 API x Legacy I/O API



- O Java conta com uma API de I/O desde a sua primeira versão
 - Esta API tem algumas limitações
- A New I/O (NIO.2) API surgiu para resolver os problemas que a API legada possui
- A NIO.2 API surgiu com o Java 7
- A tendência é que ela substitua a API de I/O legada gradualmente

A interface *Path*



- O sistema de arquivos define caminhos para diretórios e arquivos

Windows


C:\Users\Me\MyFile.txt

Unix

/home/me/Myfile.txt

- Um objeto *Path* representa um caminho
 - Pode ser de um arquivo ou diretório
 - O arquivo ou diretório não precisam necessariamente existir

Criando objetos *Path*



- A classe *Paths* possui métodos estáticos, que permitem manipular objetos *Path*


Path p = Paths.get("C:\\Users\\Me\\MyFile.txt");

Cria um objeto *Path* com base no caminho especificado

Path p = Paths.get("C:", "Users", "Me", "MyFile.txt");

Cria um objeto *Path* juntando partes do caminho

Juntando objetos *Path*



- É comum haver situações onde dois ou mais objetos *Path* precisam ser juntados para gerar um *Path* completo

Path dir = Paths.get("/home/me");


Path file = Paths.get("MyFile.txt");

Path path = dir.resolve(file);

Path path = dir.resolve("MyFile.txt");

↓

/home/me/MyFile.txt


Métodos úteis de *Path*


```
Path p = Paths.get("C:\\Users\\Me\\MyFile.txt");
```

Método	Retorno
<code>p.toString()</code>	"C:\\Users\\Me\\MyFile.txt"
<code>p.getFileName()</code>	"MyFile.txt"
<code>p.getParent()</code>	"C:\\Users\\Me"
<code>p.getNameCount()</code>	3
<code>p.getName(1)</code>	"Me"
<code>p.subPath(0, 2)</code>	"Users\\Me"
<code>p.getRoot()</code>	"C:\\"

```
Path p = Paths.get("files\\MyFile.txt");
```

Método	Retorno
<code>p.toAbsolutePath()</code>	"C:\\DiretorioCorrente\\files\\MyFile.txt"

A classe *Files*



- Possui métodos estáticos que permitem manipular arquivos

```
Path path = Paths.get("/home/me/MyFile.txt");
boolean exists = Files.exists(path);
```

Verifica se o arquivo ou diretório existe

- Outros métodos de *Files*

Método	Descrição
<code>isReadable(path)</code>	Permite leitura?
<code>isWritable(path)</code>	Permite escrita?
<code>isExecutable(path)</code>	Pode ser executado?
<code>isDirectory(path)</code>	É um diretório?
<code>isHidden(path)</code>	Está oculto?
<code>size()</code>	Retorna o tamanho em bytes

Operações em arquivos e diretórios


- Excluir

```
Files.delete(path);
Files.deleteIfExists(path);
```


- Copiar

```
Files.copy(sourcePath, targetPath);
Files.copy(sourcePath, targetPath,
StandardCopyOption.REPLACE_EXISTING);
```

- Mover

```
Files.move(sourcePath, targetPath);
Files.move(sourcePath, targetPath,
StandardCopyOption.REPLACE_EXISTING);
```

Lendo dados de arquivos



```
InputStream in = Files.newInputStream(path);
```

Arquivos binários

```
BufferedReader reader = Files.newBufferedReader(path);
```

Arquivos texto


InputStream e BufferedReader são classes que fazem parte da API legada de I/O

```
byte[] bytes = Files.readAllBytes(path);
```

```
List<String> lines = Files.readAllLines(path);
```

Recomendados para arquivos menores

Escrevendo dados em arquivos



```
OutputStream out = Files.newOutputStream(path);
```

Arquivos binários

```
BufferedWriter writer = Files.newBufferedWriter(path);
```

Arquivos texto


OutputStream e BufferedWriter são classes que fazem parte da API legada de I/O

```
Files.write(path, bytes);
```

```
Files.write(path, lines);
```

Recomendados para arquivos menores

O enum *StandardOpenOptions*



- Várias operações de manipulação de arquivos e diretórios solicitam parâmetros que definem o comportamento destas operações

Elemento do enum	Descrição
WRITE	Arquivo para escrita.
APPEND	Faz append de dados.
TRUNCATE_EXISTING	Faz o truncamento do arquivo.
CREATE_NEW	Cria um novo arquivo. Se ele existir, lança exceção.
CREATE	Se o arquivo existir, abre. Se não existir, cria.
DELETE_ON_CLOSE	Exclui o arquivo quando a stream é fechada.

Criando arquivos

- Arquivos regulares


```
Path file = Files.createFile(path);
```
- Arquivos temporários


```
Path file = Files.createTempFile("files_", ".temp");
```

↓

```
C:\Users\Me\AppData\Local\Temp\files_1365648452074156411.temp
```

Criando diretórios

- Diretórios regulares


```
Path dir = Files.createDirectory(path);
```

```
Path dir = Files.createDirectories(path);
```

→ Cria vários níveis de diretórios
- Diretórios temporários


```
Path dir = Files.createTempDirectory("temp_");
```

↓

```
C:\Users\Me\AppData\Local\Temp\temp_6589015846917767979
```

Conteúdo de um diretório

- A interface *DirectoryStream* permite iterar sobre o conteúdo de um diretório


```
try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir)) {
    for (Path path : stream) {
        //...
    }
}
```
- É possível filtrar por padrões no nome


```
try (DirectoryStream<Path> stream =
    Files.newDirectoryStream(dir, "*.txt")) {
    for (Path path : stream) {
        //...
    }
}
```

→ Glob

Para saber mais sobre o glob, consulte o javadoc de `FileSystem.getPathMatcher()`

Criando filtros de conteúdos



- Nem sempre o uso do glob atende às necessidades de filtragem
 - Ele se baseia apenas em padrões de nome
- Nestes casos é preciso implementar um filtro customizado

```
DirectoryStream.Filter<Path> filter =  
file -> Files.isDirectory(file);
```

Expressão lambda para o método `accept()`

```
DirectoryStream<Path> s = Files.newDirectoryStream(dir, filter) {
```

Iterando sobre a árvore de diretórios



- A interface *FileVisitor* possui 4 métodos que devem ser implementados
 - *preVisitDirectory()*
 - *postVisitDirectory()*
 - *visitFile()*
 - *visitFileFailed()*
- É possível também estender a classe *SimpleFileVisitor*
- O método *Files.walkFileTree()* é chamado para iniciar a iteração

Watch Service API



- Permite que a aplicação seja notificada quando um diretório ou arquivo é criado, excluído ou modificado

```
WatchService watcher = FileSystems.getDefault().newWatchService();  
WatchKey key = dir.register(watcher,  
ENTRY_CREATE, ENTRY_DELETE, ENTRY_MODIFY);
```

- Deve existir um loop na aplicação que fica recebendo os eventos

Integrando NIO.2 com a Legacy I/O API

- Aplicações antigas (anteriores ao Java 7) não tiveram acesso à NIO.2 API
- Para que estas aplicações não precisem ser reescritas, existe uma forma de integrar a API legada de I/O com a NIO.2 API



```
Path path = file.toPath();
```

```
File file = path.toFile();
```

