Operações de Entrada e Saída — Java NIO.2

ACH 2003 — Computação Orientada a Objetos

Daniel Cordeiro

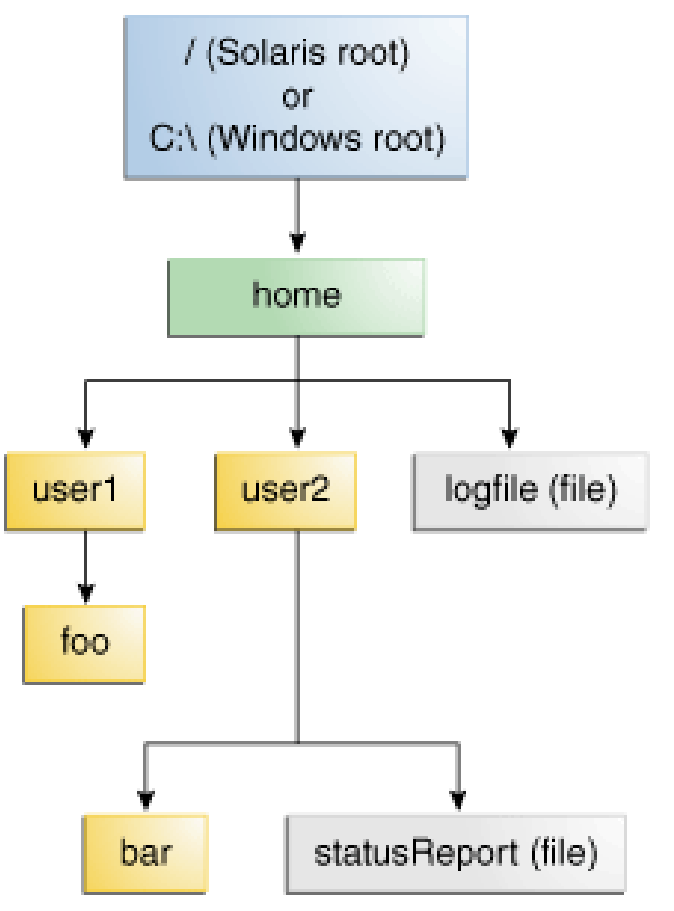
Escola de Artes, Ciências e Humanidades | EACH | USP

Conceito

Um sistema de arquivos organiza arquivos em algum dispositivo de armazenamento (ex: disco rígido) de forma que eles possam ser facilmente recuperados.

- arquivos são organizados em estrutura hierárquica (ex: árvores)
- no topo da árvore há um (ou mais) nós raiz
- em baixo do topo há arquivos e diretórios (pastas), que contém outros arquivos e outros diretórios, que por sua vez contém outros arquivos e outros diretórios, ...

CAMINHO (*PATH*)



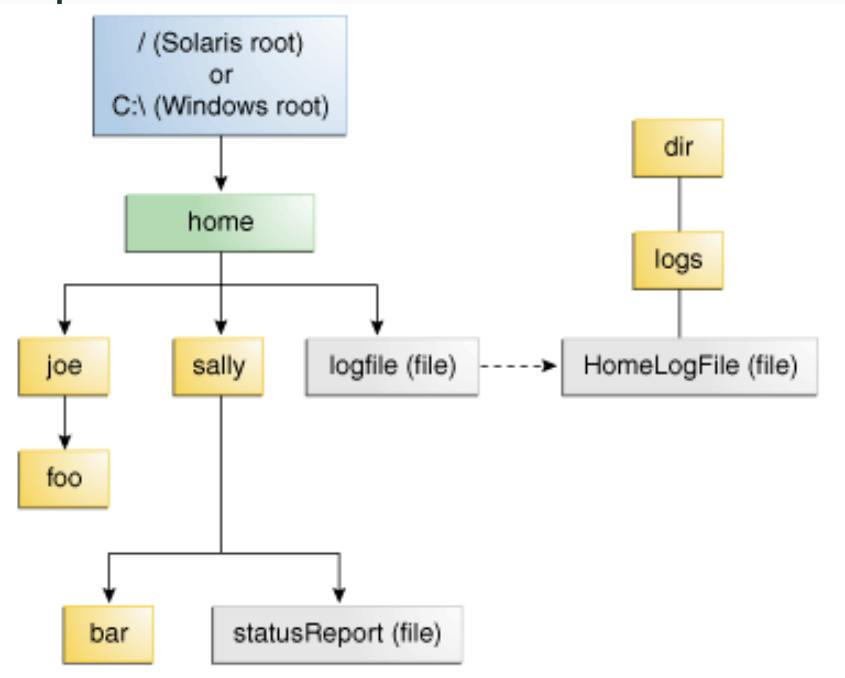
- no Windows cada nó raiz é um volume (`C:\`, `D:\`)
- sistemas Unix só possuem uma raiz, denotada pelo caractere "/"
- o arquivo é identificado pelo caminho a partir da raiz:
  - `/home/sally/statusReport` (Unix)
  - `C:\home\sally\statusReport` (Windows)
- caractere delimitador: separa o nome dos arquivos ("/" ou "\")

CAMINHO (*PATH*)

Absoluto ou relativo

absoluto sempre contém a raiz no caminho. Ex: `/home/sally/statusReport`

relativo precisa ser combinado com outro caminho. Ex: `joe/foo`



Link simbólico

Um nó (que parece um arquivo comum) na verdade aponta para um outro caminho.

CLASSE PATH

- ponto de entrada para manipulação de arquivos
- classe `java.nio.file.Path` é a forma programática de representar um caminho em Java
- contém o nome do arquivo e a lista de diretórios usados para construir o caminho
- permite examinar, localizar e manipular arquivos

CRIAÇÃO DE UM PATH

Instâncias de `Path` são criadas usando o método estático `Path.of()` da classe `Path` para Java $\geq$ 11 ou com o método `Paths.get()` da classe auxiliar `Paths`[1]. Exemplos:

```
Path p1 = Path.of("/tmp/foo");
Path p2 = Path.of(args[0]);
Path p3 = Path.of(URI.create("file:///Users/joe/FileTest.java"));

// Path.of("/users/sally") equivale a:
Path p4 = FileSystems.getDefault().getPath("/users/sally");

// /home/joe/logs/foo.log ou C:\Users\joe\logs\foo.log
Path p5 = Paths.get(System.getProperty("user.home"),"logs","foo.log");
```

[1] `Paths.get()` chama `Path.of()` com os mesmos argumentos, mas pode ser marcado como obsoleto (*deprecated*) no futuro. No tutorial do Java, escrito para Java 8, os exemplos ainda usam `Paths.get()`.

INFORMAÇÕES SOBRE UM CAMINHO

```java
// Nenhum dos métodos exige que o arquivo exista de fato
// sintaxe Windows
Path path = Path.of("C:\\home\\joe\\foo");
// sintaxe Unix
Path path = Path.of("/home/joe/foo");

System.out.format("toString: %s%n", path.toString());
 // /home/joe/foo ou C:\home\joe\foo
System.out.format("getFileName: %s%n", path.getFileName());
 // foo
System.out.format("getName(0): %s%n", path.getName(0));
 // home
System.out.format("getNameCount: %d%n", path.getNameCount());
 // 3
System.out.format("subpath(0,2): %s%n", path.subpath(0,2));
 // home/joe ou home\joe
System.out.format("getParent: %s%n", path.getParent());
 // /home/joe ou \home\joe
System.out.format("getRoot: %s%n", path.getRoot());
 // / ou C:\
```

CONVERSÃO DE CAMINHOS

para URI :

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

para caminho absoluto :

```
// cd $HOME/.config
Path inputPath = Path.of("libreoffice");
Path fullPath  = inputPath.toAbsolutePath();
// /home/danielc/.config/libreoffice
```

para o caminho real :

- resolve links simbólicos
- se o caminho for relativo, devolve absoluto
- se houver elementos redundantes, são removidos

```
// $ ls -l ~/coo
// lrwxrwxrwx 1 danielc danielc 41 fev 15 17:13
//   /home/danielc/coo ->
//                  /home/danielc/aulas/ach2003-objetos/
Path p = Path.of("/home/danielc/coo").toRealPath()
// /home/danielc/aulas/ach2003-objetos
```

Combinação parcial

```
Path p1 = Path.of("/home/joe/foo");
System.out.format("%s%n", p1.resolve("bar"));
 // Resultado é /home/joe/foo/bar

Path.of("foo").resolve("/home/joe");
 // Resultado é /home/joe já que o caminho era absoluto
```

CONCATENAÇÃO DE CAMINHOS

Combinação parcial

```
Path p1 = Path.of("/home/joe/foo");
System.out.format("%s%n", p1.resolve("bar"));
 // Resultado é /home/joe/foo/bar

Path.of("foo").resolve("/home/joe");
 // Resultado é /home/joe já que o caminho era absoluto
```

Combinação relativa

```
Path p1 = Path.of("home");
Path p3 = Path.of("home/sally/bar");

Path p1_to_p3 = p1.relativize(p3);
// Resultado é sally/bar

Path p3_to_p1 = p3.relativize(p1);
// Resultado é ../..
```

OPERAÇÕES EM ARQUIVOS

- Outro ponto de entrada para o uso do pacote `java.nio.file` é a classe `Files`
- `Files` oferece um conjunto de métodos estáticos para leitura, escrita e manipulação de arquivos e diretórios
- Os métodos recebem e/ou devolvem instâncias de `Path`
- Os métodos de `Files` detectam se há *links* simbólicos envolvidos e se adaptam automaticamente ou permitem que o programador especifique o que deve ser feito quando um *link* for encontrado

Tratamento de exceções

- **muitas** coisas podem sair errado em operações de E/S
- para facilitar o tratamento de exceções, muitas classes de fluxos ou canais implementam ou estendem a interface `java.io.Closeable`

```
Charset charset = Charset.forName("US-ASCII");
String s = ...;
try (BufferedWriter writer = Files.newBufferedWriter(file, charset)) {
    writer.write(s, 0, s.length());
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

Varargs

Vários métodos em `Files` foram definidos para receber um número arbitrário de parâmetros[2]. No exemplo abaixo, 0 ou mais instâncias de `CopyOption` podem ser passadas na chamada ao método `move()`:

```
import static java.nio.file.StandardCopyOption.*;

// Path Files.move(Path, Path, CopyOption...)
Path source = ...;
Path target = ...;
Files.move(source,
           target,
           REPLACE_EXISTING,
           ATOMIC_MOVE);
```

[2] Veja https://docs.oracle.com/javase/8/docs/technotes/guides/language/varargs.html.

Operações atômicas

Alguns métodos podem realizar operações atômicas, ou seja, operações que não podem ser interrompidas ou realizadas "parcialmente". Importante em programação concorrente.

Operações atômicas

Alguns métodos podem realizar operações atômicas, ou seja, operações que não podem ser interrompidas ou realizadas "parcialmente". Importante em programação concorrente.

Encadeamento de métodos

Muitos exemplos usarão uma técnica de programação orientada a objetos chamada encadeamento de métodos. Você chama um método e ele devolve um objeto, então você chama o método nesse objeto, e ele devolve outro objeto, etc.

```
String value = Charset.defaultCharset().decode(buf).toString();
UserPrincipal group =
    file.getFileSystem().getUserPrincipalLookupService().
        lookupPrincipalByName("danielc");
```

Glob

Um argumento *glob* é uma string com um padrão de busca que será comparada com outras strings que representam nomes de arquivos e diretórios. Exemplos:

`*`	casa com qualquer quantidade de caracteres
`**`	funciona como `*`, mas percorre vários diretórios
colchetes	conjunto ou faixa de caracteres. Ex: `[aeiou]`, `[0-9]`, `[a-z,A-Z]`, etc.
chaves	uma coleção de consultas. Ex: `{temp*,tmp*}`

Exemplo de uso:

- `*.java`
- `???`
- `*.{html,htm,pdf}`
- `{foo*,*[0-9]*}`

VERIFICAR UM ARQUIVO OU DIRETÓRIO

Verificação de existência

- `Files.exists(Path, LinkOption...)` e `Files.notExists(Path, LinkOption...)`

Três resultados são possíveis:

- o arquivo existe com certeza
- o arquivo não existe com certeza
- o estado do arquivo é desconhecido (ex: o programa não tem acesso ao arquivo); nesse caso tanto `exists()` como `notExists()` devolvem `false`.

Permissão de acesso

O código a seguir verifica se o arquivo existe e se pode ser executado:

```
Path file = ...;
boolean isRegularExecutableFile = Files.isRegularFile(file) &
         Files.isReadable(file) & Files.isExecutable(file);
```

Permissão de acesso

O código a seguir verifica se o arquivo existe e se pode ser executado:

```
Path file = ...;
boolean isRegularExecutableFile = Files.isRegularFile(file) &
         Files.isReadable(file) & Files.isExecutable(file);
```

Dois caminhos podem apontar pro mesmo arquivo

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

if (Files.isSameFile(p1, p2)) {
    // p1 pode ser um link simbólico para p2
}
```

REMOVER UM ARQUIVO OU DIRETÓRIO

- você pode remover arquivos, diretório ou *links*
- quando um *link* é removido, seu alvo não é removido
- um diretório precisa estar vazio para ser removido

A classe `Files` provê dois métodos para remoção:

delete(Path) remove um arquivo ou lança uma `NoSuchFileException` se ele não existir

deleteIfExists(Path) remove um arquivo, mas não faz nada se ele não existir

```
try {
    Files.delete(path);
} catch (NoSuchFileException x) {
    System.err.format("%s: arq. ou diretório não encontrado%n", path);
} catch (DirectoryNotEmptyException x) {
    System.err.format("%s not empty%n", path);
} catch (IOException x) {
    // Problemas com permissões de arquivo são capturados aqui
    System.err.println(x);
}
```

COPIAR UM ARQUIVO OU DIRETÓRIO

- cópias podem ser realizadas com o método `Files.copy(Path, Path, CopyOption...)`
- a cópia de um arquivo falha se o destino existir, exceto se a opção `REPLACE_EXISTING` for usada
- a cópia de um link produz uma cópia de seu alvo. Para copiar o próprio link, use a opção `NOFOLLOW_LINKS`
- para copiar o arquivo e seus atributos (ex: last-modified-time), use a opção `COPY_ATTRIBUTES`

Cópia de/para fluxos

- `Files.copy(InputStream, Path, CopyOption...)`
- `Files.copy(Path, OutputStream)`

MOVER UM ARQUIVO OU DIRETÓRIO

`Files.move(Path, Path, CopyOption...)`

As seguintes opções são válidas:

REPLACE_EXISTING sobrescreve o destino; se o destino for um link simbólico, o link é sobrescrito mas o arquivo para o qual aponta, não

ATOMIC_MOVE se o sistema de arquivos permitir, a operação é garantida como sendo atômica

Exemplo:

```
import static java.nio.file.StandardCopyOption.*;
...
Files.move(source, target, REPLACE_EXISTING);
```

METADADOS

Métodos de `Files` que devolvem metadados sobre arquivos e diretórios:

- `size(Path)`
- `isDirectory(Path, LinkOption)`
- `isRegularFile(Path, LinkOption...)`
- `isSymbolicLink(Path)`
- `isHidden(Path)`
- `getLastModifiedTime(Path, LinkOption...)`
- `setLastModifiedTime(Path, FileTime)`
- `getOwner(Path, LinkOption...)`
- `setOwner(Path, UserPrincipal)`
- `getPosixFilePermissions(Path, LinkOption...)`
- `setPosixFilePermissions(Path, Set<PosixFilePermission>)`

ABERTURA DE ARQUIVOS

Vários métodos de manipulação de arquivo recebem um parâmetro opcional chamado `OpenOptions`, que pode receber um dos seguintes valores definidos pelo enum `StandardOpenOptions`:

WRITE	abre um arquivo para escrita
APPEND	adiciona novos dados ao final do arquivo
TRUNCATE_EXISTING	trunca o arquivo para zero bytes
CREATE_NEW	cria um novo arquivo e lança exceção se o caminho já existir
CREATE	abre um arquivo se existir ou cria um novo
DELETE_ON_CLOSE	remove o arquivo quando o fluxo for fechado. Útil para arquivos temporários
SPARSE	informa que o arquivo será esparso, alguns sistemas de arquivos (quase todos os de Unix, NTFS, etc.) podem usar a informação para economizar espaço
SYNC	mantém o arquivo (conteúdo e metadados) sempre sincronizado com o dispositivo de armazenamento
DSYNC	mantém o conteúdo do arquivo sempre sincronizado

Ler todos os bytes/linhas do arquivo[3]

```
Path file = ...;
byte[] fileArray;
fileArray = Files.readAllBytes(file);

// ou
String[] linhas = Files.readAllLines(file, StandardCharsets.UTF_8)
```

[3] Se o charset não for especificado, assume-se UTF-8.

Ler todos os bytes/linhas do arquivo[3]

```
Path file = ...;
byte[] fileArray;
fileArray = Files.readAllBytes(file);

// ou
String[] linhas = Files.readAllLines(file, StandardCharsets.UTF_8)
```

Escrever todos os bytes/linhas em arquivo

```
Path file = ...;
byte[] buf = ...;
Files.write(file, buf, StandardOpenOption.APPEND);

// ou
Files.write(file, buf, lista_de_strings);
```

[3] Se o charset não for especificado, assume-se UTF-8.

Ler um arquivo usando um fluxo com buffer

```
Charset charset = Charset.forName("US-ASCII");
try (BufferedReader reader = Files.newBufferedReader(file, charset)) {
    String line = null;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

Ler um arquivo usando um fluxo com buffer

```
Charset charset = Charset.forName("US-ASCII");
try (BufferedReader reader = Files.newBufferedReader(file, charset)) {
    String line = null;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

Escrever um arquivo usando um fluxo com buffer

```
Charset charset = Charset.forName("UTF-8");
String s = ...;
try (BufferedWriter writer = Files.newBufferedWriter(file, charset)) {
    writer.write(s, 0, s.length());
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

Ler arquivo usando fluxo de E/S

```
Path file = ...;
try (InputStream in = Files.newInputStream(file);
    BufferedReader reader =
      new BufferedReader(new InputStreamReader(in))) {
    String line = null;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException x) {
    System.err.println(x);
}
```

Criar e escrever em fluxos de E/S

```
import static java.nio.file.StandardOpenOption.*;
import java.nio.file.*;
import java.io.*;

public class LogFileTest {

  public static void main(String[] args) {

    // Converte String em vetor de bytes
    String s = "Hello World! ";
    byte data[] = s.getBytes();
    Path p = Path.of("./logfile.txt");

    try (OutputStream out = new BufferedOutputStream(
                              Files.newOutputStream(p, CREATE, APPEND))) {
      out.write(data, 0, data.length);
    } catch (IOException x) {
        System.err.println(x);
    }
  }
}
```

Arquivos regulares

```
Path file = ...;
try {
    // Cria um arquivo vazio com as permissões padrão
    Files.createFile(file);
} catch (FileAlreadyExistsException x) {
    System.err.format("arquivo chamado %s já existe%n", file);
} catch (IOException x) {
    // Captura erros como problemas de permissão
    System.err.format("erro em createFile: %s%n", x);
}
```

MÉTODOS PARA CRIAÇÃO DE ARQUIVOS

Arquivos regulares

```
Path file = ...;
try {
    // Cria um arquivo vazio com as permissões padrão
    Files.createFile(file);
} catch (FileAlreadyExistsException x) {
    System.err.format("arquivo chamado %s já existe%n", file);
} catch (IOException x) {
    // Captura erros como problemas de permissão
    System.err.format("erro em createFile: %s%n", x);
}
```

Temporários

```
try {
    Path tempFile = Files.createTempFile(null, ".tmp");
    System.out.format("O arquivo temporário foi criado: %s%n", tempFile);
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
// O arquivo temporário foi criado: /tmp/509668702974537184.tmp
```

Listar os diretórios raiz

```
Iterable<Path> dirs = FileSystems.getDefault().getRootDirectories();
for (Path name: dirs) {
    System.err.println(name);
}
```

Listar os diretórios raiz

```
Iterable<Path> dirs = FileSystems.getDefault().getRootDirectories();
for (Path name: dirs) {
    System.err.println(name);
}
```

Criar diretório

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

// Definição de permissões
Set<PosixFilePermission> perms =
    PosixFilePermissions.fromString("rwxr-x---");
FileAttribute<Set<PosixFilePermission>> attr =
    PosixFilePermissions.asFileAttribute(perms);
Files.createDirectory(file, attr);
```

Listar os diretórios raiz

```
Iterable<Path> dirs = FileSystems.getDefault().getRootDirectories();
for (Path name: dirs) {
    System.err.println(name);
}
```

Criar diretório

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

// Definição de permissões
Set<PosixFilePermission> perms =
    PosixFilePermissions.fromString("rwxr-x---");
FileAttribute<Set<PosixFilePermission>> attr =
    PosixFilePermissions.asFileAttribute(perms);
Files.createDirectory(file, attr);
```

Diretórios temporários

- `createTempDirectory(Path, String, FileAttribute<?>...)`
- `createTempDirectory(String, FileAttribute<?>...)`

LISTAGEM DE DIRETÓRIOS

```
Path dir = ...;
try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir)) {
    for (Path file: stream) {
        System.out.println(file.getFileName());
    }
} catch (IOException | DirectoryIteratorException x) {
    System.err.println(x);
}
```

LISTAGEM DE DIRETÓRIOS

```
Path dir = ...;
try (DirectoryStream<Path> stream = Files.newDirectoryStream(dir)) {
    for (Path file: stream) {
        System.out.println(file.getFileName());
    }
} catch (IOException | DirectoryIteratorException x) {
    System.err.println(x);
}
```

Filtragem usando um glob

```
Path dir = ...;
try (DirectoryStream<Path> stream =
     Files.newDirectoryStream(dir, "*.{java,class,jar}")) {
    for (Path entry: stream) {
        System.out.println(entry.getFileName());
    }
} catch (IOException x) {
    System.err.println(x);
}
```

BIBLIOGRAFIA

- The Java™ Tutorials – Basic I/O: `https://docs.oracle.com/javase/tutorial/essential/io/`