**NIO2 Introducere**

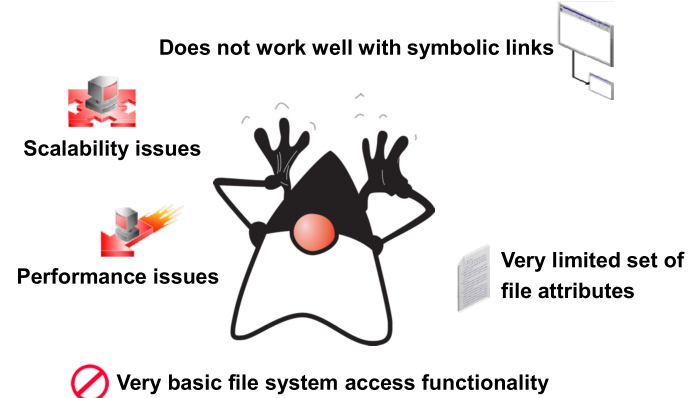
API-ul nio2 aduce urmatoarele imbunatatiri:

* Imbunatateste interfata File System
* Functionalitate completa Socket-Channel
* I/O asincrone scalabile

JSR 51 care sta la baza NIO API este orientat spre buffer-e, channel-uri si charset-uri. In plus aduce socket-uri scalabile furnizand un API neblocabil si multiplexat, care permite dezvoltarea serverelor scalabile.

Noul API:

* Functioneaza mai consistent peste platforme
* Face mai usoara scrierea programelor, care sunt in stare sa manipuleze operatiile esuate ale sistemului de fisiere
* Furnizeaza acces mai eficient la o multime mult mai mare de atribute ale fisierului
* Permite dezvoltarea aplicatiilor sofisticate, facand posibila folosirea facilitatilor dependente de platforma, acolo unde este absolut necesar
* Furnizeaza suport pentru ca fisierele sistem non-native sa fie „plugged in” platforma



API-ul java.io.File a prezentat de-a lungul timpului numeroase provocari pentru dezvoltatori:

* Multe metode nu aruncau exceptii atunci cand esuau, astfel incat era imposibil de obtinut un mesaj de eroare folositor
* Multe operatii lipseau (copierea, mutarea fisierelor, etc)
* Metoda rename nu functiona consistent pe diverse platforme
* Nu exista suport pentru link-urile simbolice
* Lipsa de suport pentru metadate precum: permisiune la fisier, proprietar de fisier, atribute de securitate
* Accesul metadatelor de fisier era ineficient (orice apel pentru metadate rezulta intr-un apel de sistem)
* Multe dintre metode nu se scalau (cererea unei liste mari dintr-un director aflat pe server putea duce la intreruperea conexiunii)
* Nu era posibil scrierea de cod corect ce traversa recursiv o arborescenta de fisiere daca erau link-uri simbolice circulare

Mai mult, IO nu a fost scris pentru a fi extins. Dezvoltatorii cereau posibilitatea de a scrie propriile implementari de fisiere. Spre exemplu, pastrarea unui pseudofisier in memorie sau formatarea fisierelor ca arhive.

In NIO2 fisierele si directoarele sunt reprezentate de o cale, care este o locatie absoluta sau relativa a fisierului sau directorului.

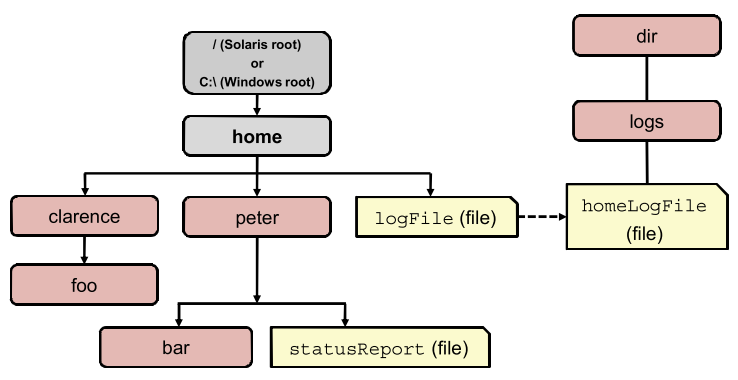
Asa dupa cum am mai afirmat, inainte de NIO2 fisierele erau reprezentate de clasa java.io.File. In NIO2 instante ale obiectelor java.nio.file.Path sunt folosite pentru a reprezenta locatii relative sau absolute ale unui fisier sau director.

Fisierele sunt structuri ierarhice. Ele pot avea unul sau mai multe directoare radacina.

Calea absoluta intotdeauna contine elementul radacina si lista completa a directoarelor pentru a localiza fisierul: /home/peter/statusReport. Toate informatiile necesare a localiza fisierul sunt continute in string-ul de cale.

Calea relativa trebuie combinata cu o alta cale pentru a accesa un fisier: clarence/foo. Fara alte informatii programul nu poate localiza directorul.

Obiectele sistemului fisier sunt directoare si fisiere. Unele dintre ele suporta notatia de link-uri simbolice, notatie care mai este cunoscuta ca „symlink” sau „soft link”.



Un link simbolic este un fisier special ce serveste ca referinta pentru un alt fisier. Un link simbolic este de obicei transparent pentru utilizator. Citirea sau scrierea unui link simbolic este aceeasi cu citirea sau scrierea unui alt fisier sau director.

In imaginea anterioara logFile pare a fi un fisier obisnuit, dar, in realitate, este un link simbolic catre dir/logs/homeLogFile. homeLogFile este target-ul link-ului.

In NIO2 avem urmatoarele pachete si clase:

* java.nio.file.Path, care localizeaza un fisier sau director folosind o cale dependenta de sistem
* java.nio.file.Files, care folosind un Path executa operatii pe fisiere sau directoare
* java.nio.file.FileSystem, care furnizeaza o interfata catre un sistem de fisiere si un factory pentru crearea unui Path si a altor obiecte ce acceseaza sistemul de fisiere
* toate metodele ce acceseaza sistemul de fisiere arunca o IOException sau o subclasa a sa, ce furnizeaza detalii despre cauza exceptiei

Diferenta semnificativa intre NIO2 si java.io.File este arhitectura de acces la sistemul de fisiere. In clasa File metodele folosite pentru a manipula informatii despre cai sunt in aceeasi clasa cu metodele utilizate pentru a scrie si citi fisiere si directoare.

In NIO2 cele doua concepte sunt separate. Caile sunt create si manipulate folosind interfata Path, in timp ce operatiile pe fisiere si directoare sunt in responsabilitatea clasei Files, ce opereaza doar pe obiecte Path.

**Interfata Path**

Furnizeaza punctul de intrare pentru manipularea fisierelor si directoarelor prin NIO2. Pentru a obtine un obiect Path:

* obtinem o instanta a sistemului de fisiere predefinit si apoi invocam metoda getPath().

FileSystem fs = FileSystems.*getDefault*();

Path p44 = fs.getPath("C:\\teste\\Steeams\\out.txt");

* Apelam metoda get() din helper-ul Paths

Path p1 = Paths.*get*("/home/student/./bob/file");

Obiectele Path sunt nemodificabile, o data ce au fost create nu mai pot fi schimbate.

Observatie: daca dorim sa lucram cu sistemul de fisiere predefinit, care este sistemul de fisiere pe care JVM ruleaza, utilitarul Paths ofera metoda cea mai scurta de obtinere a lui Path. Altfel, cand dorim sa efectuam operatii pe Path, pe un alt sistem de fisiere decat cel default, va trebui sa obtinem mai intai instanta sistemului si apoi sa construim obiectul Path.

Observatie: sistemul de fisiere windows utilizeaza un back slash in mod predefinit. Windows accepta ambele delimitatoare: slash si back slash. Back slash-urile in Java sunt caractere escape, de aceea pentru a reprezenta un back slash va trebui sa-l scriem dublu.

Metodele lui Path cuprind:

* Accesul componentelor unei cai:
  + Path getFileName(); returneaza end pointul acestei cai
  + Path getParent(); returneaza calea parinte sau null (fisier sau director)
  + getNameCount()
* Operare pe o cale: normalize(), toUri(), toAbsolutePath(), subpath(), resolve(), relativize()
* Compararea cailor: startsWith(), endsWith(), equals()

Este bine sa gandim obiectele Path in acelasi fel ca pe obiectele String. Obiectele Path pot fi create dintr-un singur text sau dintr-o multime de componente:

* O componenta root, ce identifica ierarhia de sistem
* Un element cu nume, cel mai indepartat de radacina, ce defineste fisierul sau directorul catre care indica calea
* Alte elemente, separate de un caracter special sau un delimitator ce identifica numele directoarelor ce sunt parte a ierarhiei

Exemplu:

**public** **class** PathOperationsTest {

**public** **static** **void** main(String[] args) {

// Normalize

System.*out*.println("Normalize");

Path p1 = Paths.*get*("/home/student/./bob/file");

Path pN1 = p1.normalize();

Path p2 = Paths.*get*("/home/student/bob/../sally/file");

Path pN2 = p2.normalize();

System.*out*.format("Normalized paths: %s%n%s%n", pN1, pN2);

// Subpath

System.*out*.println("Subpath");

Path p3 = Paths.*get*("D:\\Temp\\foo\\bar");

Path p4 = p3.subpath(0, 2);

System.*out*.println("Count: " + p3.getNameCount());

System.*out*.println(p3.getName(0));

System.*out*.println(p4);

// Resolve

System.*out*.println("Resolve");

Path p5 = Paths.*get*("/home/clarence/foo");

Path p6 = p5.resolve("bar");

System.*out*.println(p6);

Path p7 = Paths.*get*("foo").resolve("/home/clarence");

System.*out*.println(p7);

// Relativize

System.*out*.println("Relativize");

Path p8 = Paths.*get*("peter");

Path p9 = Paths.*get*("clarence");

Path p8Top9 = p8.relativize(p9);

Path p9Top8 = p9.relativize(p8);

System.*out*.println(p8Top9);

System.*out*.println(p9Top8);

}

}

Descrierea metodelor din exemplu:

* int getNameCount();, returneaza numarul de elemente nume ce formeaza aceasta cale
* Path getRoot();, returneaza componenta radacina a caii
* boolean isAbsolute();, returneaza true daca calea contine un element radacina. In Windows elementul radacina contine o litera si doua puncte. Pentru Unix true este returnat pentru orice cale ce incepe cu slash
* Path toAbsolutePath();, returneaza calea absoluta a caii
* java.net.URI toUri();, returneaza URI-ul absolut

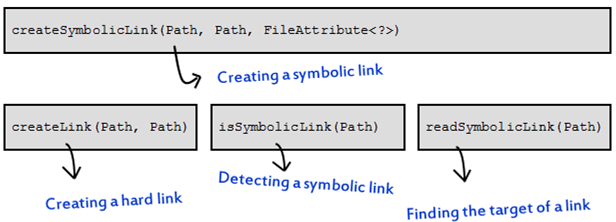
Multe sisteme de fisiere utilizeaza notatia . pentru a marca directorul curent si .. pentru directorul parinte. Putem intalni astfel situatii in care Path contine informatii redundante. Metoda normalize() elimina orice elemente redundante. Este de retinut ca metoda nu verifica sistemul in momentul in care modifica calea, este o operatie pur sintactica.

O portiune a unei cai poate constitui o subcale si se poate obtine prin metoda: Path subpath(int beginIndex, int endIndex);. Elementul cel mai apropiat de radacina are indexul 0. Elementul cel mai indepartat de radacina are indexul count-1. Rezultatul este o cale ce cuprinde elementele ce incep de la beginIndex pina la endIndex-1.

Metoda resolve() este utilizata pentru a combina doua cai. Ea accepta o cale partiala, care este calea ce nu include elementul radacina, cale ce este adaugata caii originale. Daca trimitem ca parametru o cale absoluta metoda returnea calea trimisa.

Metoda relativize() ne permite sa construim o cale de la o locatie din sistem la o alta locatie. Noua cale este relativa la cea originala.

Pachetul java.nio.file si interfata Path sunt „link aware”. Orice metoda poate detecta ce are de facut atunci cand se intalneste un link simbolic sau ofera posibilitatea de configurare a comportamentului metodei atunci cand se intalneste un link simbolic.



Unele sisteme de fisiere suporta link-urile hard. Link-urile hard sunt mult mai restrictive decat link-urile simbolice:

* target-ul link-ului trebuie sa existe
* link-urile hard nu sunt in general permise directoarelor
* hard link-urile nu sunt permise pe partitii sau volume
* un link hard arata si se comporta ca un fisier regulat, de aceea este greu de gasit
* un link hard este aceeasi entitate ca un fisier original. Toate atributele sunt identice

Datorita acestor restrictii hard link-urile nu sunt folosite la fel de des precum link-urile simbolice.

**Clasa Files**

Este entry point-ul primar pentru operatii pe obiecte Path. Metodele statice ale clasei citesc, scriu sau manipuleaza fisiere si directoare reprezentate de obiecte Path. Clasa este, de asemenea, link aware.

Obiectele Path reprezinta conceptul de locatie e unui fisier sau director. Inainte de a accesa un fisier sau director va trebui sa accesam sistemul de fisiere pentru a determina daca acesta exista:

* exists(Path p, LinkOption...option), care testeaza daca fisierul exista. In mod implicit sunt urmate link-urile simbolice
* notExists(Path p, LinkOption...option), care testeaza daca fisierul nu exista. In mod implicit sunt urmate link-urile simbolice

Exemplu:

Path p = Paths.*get*("D:\\Temp\\foo\\bar");

System.*out*.format("Path %s exists: %b%n", p,

Files.*exists*(p, LinkOption.*NOFOLLOW\_LINKS*));

Cand testam existenta unui fisier avem urmatoarele posibilitati:

* fisierul este verificat daca exista
* fisierul este verificat ca nu exista
* statutul fisierului este necunoscut. Acest rezultat poate apare atunci cand programul nu are acces la fisier

Observatie: !Files.*exists*(path) nu este echivalent cu Files.*notExists*(path). Daca ambele returneaza false, existenta fisierului sau directorului nu poate fi determinata. In Windows, spre exemplu, aceasta poate fi obtinuta atunci cand cerem statutul unui drive off-line, precum CD-ROM-ul.

Pentru a verifica ca un fisier poate fi accesat avem urmatoarele metode booleene: isReadable(Path), isWritable(Path), isExecutable(Path). Aceste operatii nu sunt atomice, cu respectarea altor operatii pe sistemul de fisiere. De aceea rezultatul acestor metode nu este sigur pentru ca poate fi modificat imediat dupa incheierea metodei.

Metoda isSameFile(Path, Path) testeaza daca doua cai indica catre acelasi fisier. Aceasta este folositor in sisteme ce suporta link-uri simbolice.

Fisierele si directoarele pot fi create folosind una dintre metodele: Files.createFile(Path), Files.createDirectory(Path). Metoda Files.createDirectories(Path) poate fi utilizata pentru a crea directoare ce nu exista, de sus in jos.

Putem sterge fisiere, directoare sau link-uri. Metoda Files.delete(Path) sterge fisierul sau arunca o exceptie in caz de esec. Spre exemplu, daca fisierul nu exista arunca o NoSuchFileException. Metoda Files.deleteIfExists(Path) sterge fisierul, dar nu arunca exceptii. Aceasta metoda este utila in cazul firelor de executie.

Putem copia un fisier sau director prin Files.copy(Path, Path, CopyOption...). Cand directoarele sunt copiate, fisierele continute nu sunt copiate. Copierea esueaza daca fisierul destinatie exista, mai putin atunci cand specificam optiunea REPLACE\_EXISTING. Cand copiem un link simbolic destinatia link-ului este copiata. Daca dorim sa copiem link-ul insusi, si nu continutul sau, specificam optiunea NOFOLLOW\_LINKS sau REPLACE\_EXISTING. In cazul folosirii optiunii REPLACE\_EXISTING in cazul copierii unui director nevid, copierea esueaza cu aruncarea FileAlreadyExistsException. Optiunea COPY\_ATTRIBUTES determina copierea atributelor de fisier asociate, fisierului destinatie.

Avem la dispozitie doua metode de copiere dintr-un flux intr-un fisier: copy(InputStream source, Path target, CopyOption...options), respeciv dintr-un fisier intr-un stream: copy(Path source, OutputStream out).

Exemplu de folosire a metodei de copiere dintr-o pagina web intr-un fisier:

Path path = Paths.*get*("C:/oracle.html");

URI u = URI.*create*("http://www.oracle.com/");

**try** (InputStream in = u.toURL().openStream()) {

Files.*copy*(in, path, StandardCopyOption.*REPLACE\_EXISTING*);

} **catch** (IOException e) {

e.printStackTrace();

}

Exemplul de dinainte poate fi extins pentru a scrie un fisier intr-un socket sau alte tipuri de fluxuri.

Putem muta un fisier sau director prin utilizarea metodei: Files.move(Path, Path, CopyOption...). La mutare trebuie sa tinem cont de urmatoarele:

* daca destinatia este un director si directorul este vid, mutarea reuseste daca este setat REPLACE\_EXISTING
* daca directorul destinatie nu exista, mutarea reuseste. Aceasta este de fapt o redenumire a directorului
* daca directorul exista si nu este vid este aruncata o DirectoryNotEmptyException
* daca sursa este un fisier si destinatia este un director care exista, REPLACE\_EXISTING este setat, mutarea va redenumi fisierul la numele directorului

Pentru a muta un director ce contine fisiere intr-un alt director trebuie sa copiem recursiv continutul directorului si apoi sa stergem vechiul director.

Putem efectua mutarea ca pe o operatie atomica folosind ATOMIC\_MOVE. Daca sistemul de fisiere nu suporta mutarea atomica se va arunca o exceptie. Cu mutarea atomica putem muta un fisier intr-un director si avem garantia ca un proces ce urmareste directorul acceseaza fisierul intreg.

Clasa DirectoryStream furnizeaza un mecanism pentru a itera peste toate intrarile dintr-un director. Clasa se scaleaza pentru a suporta foarte multe directoare. Exceptia DirectoryIteratorException este aruncata daca apar eori I/O pe timpul iterarii. Exceptia PatternSyntaxException este aruncata daca pattern-ul furnizat ca al doilea argument este invalid.

**public** **class** ListDirectory {

**public** **static** **void** main(String[] args) {

Path dir = Paths.*get*("c:/chestii");

String filter = "\*";

**try** (DirectoryStream<Path> stream = Files.*newDirectoryStream*(dir, filter)) {

**for** (Path file : stream) {

System.*out*.println(file.getFileName());

}

} **catch** (PatternSyntaxException | DirectoryIteratorException | IOException x) {

//IOException can never be thrown by the iteration.

//In this snippet, it can only be thrown by newDirectoryStream.

System.*err*.println(x);

}

}

}

Daca avem un fisier de dimensiuni mici si dorim sa-i citim intregul continut intr-un singur pas putem folosi metodele Files.readAllBytes(Path) sau Files.readAllLines(Path, Charset). Aceste metode gestioneaza intreaga operatiune precum inchiderea si deschiderea stream-ului.

Putem folosi una dintre metodele de scriere pentru a scrie octeti sau linii intr-un fisier: write(Path, byte[], OpenOption...), respectiv write(Path, Iterable<? extends CharSequence>, Charset, OpenOption...).

**Channel-uri si byte buffer-i**

Streamurile citesc un caracter o data, in timp ce channel-urile citesc o data un buffer. Interfata ByteChannel furnizeaza functionalitatea de baza pentru scriere si citire. Un SeekableByteChannel este un ByteChannel ce are capabilitatea sa mentina o pozitie in canal si sa o schimbe.

Capabilitatea de miscare la diferite puncte ale fisierului si apoi citirea de la sau scrierea la acea locatie face posibil accesul random la acel fisier. Avem doua metode de scriere si citire in canal:

* newByteChannel(Path, OpenOperation...), ce returneaza un SeekableByteChannel. Putem converti acest rezultat la un FileChannel, ce furnizeaza acces la facilitati mai avansate precum: maparea unei regiuni a fisierului direct in memorie pentru acces rapid, blocarea unei regiuni a fisieruluiastfel incat alte procese sa nu o poata accesa, citirea sau scrierea octetilor de la o pozitie absoluta fara a afecta pozitia curenta a canalului
* newByteChannel(Path, Set<? Extends OpenOption>, FileAttribute<?>...)

Exemplu:

**public** **class** ByteChannelCopyTest {

**public** **static** **void** main(String[] args) {

**try** (FileChannel fcIn = **new** FileInputStream("file.in").getChannel();

FileChannel fcOut = **new** FileOutputStream("file.out")

.getChannel()) {

System.*out*.println("File size: " + fcIn.size());

// Create a buffer to read into

ByteBuffer buff = ByteBuffer.*allocate*((**int**) fcIn.size());

System.*out*.println("Bytes remaining: " + buff.remaining());

System.*out*.println("Bytes read: " + fcIn.read(buff));

buff.position(0);

System.*out*.println("Buffer: " + buff);

System.*out*.println("Bytes remaining: " + buff.remaining());

System.*out*.println("Wrote: " + fcOut.write(buff) + " bytes");

} **catch** (FileNotFoundException f) {

System.*out*.println("File not found: " + f);

} **catch** (IOException e) {

System.*out*.println("IOException: " + e);

}

}

}

Fisierele cu acces aleator permit acces non-secvential la continutul lor. Pentru a accesa un fisier aleator trebuie sa deschidem fisierul, sa ne pozitionam pe o anumita locatie si sa citim sau scrie acolo. Toata aceasta functionalitate este furnizata de interfata SeekableByteChannel.

Dupa cum am mai spus aceasta innterfata extinde canalul cu notiunea de pozitie curenta. Metodele ne permit sa setam sau sa interogam pozitia:

* position(), returneaza pozitia curenta a canalului
* position(long), seteaza pozitia canalului
* read(ByteBuffer), citeste octeti in buffer de la canal
* write(ByteBuffer), scrie octeti din buffer la canal
* truncate(long), truncheaza fisierul conectat la canal

Metoda Files.newBufferedReader(Path, Charset) deschide un fisier pentru citire, returnand un BufferedReader, utilizat pentru citirea textului dintr-un fisier intr-o maniera eficienta.

Metoda Files.newBufferedWriter(Path, Charset) deschide un fisier pentru scriere, returnand un BufferedWriter, utilizat pentru scrierea textului intr-un fisier intr-o maniera eficienta.

NIO2 suporta si urmatoarele metode pentru a deschide stream-uri de octeti:

InputStream in = Files.*newInputStream*(file);

BufferedReader reader = **new** BufferedReader(**new** InputStreamReader(in));

line = reader.readLine();

Pentru a crea un fisier, adauga intr-un fisier putem utiliza urmatoarea secventa de cod:

Path logfile = ...;

String s = ...;

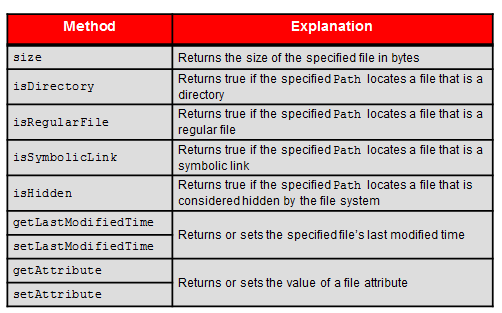
**byte** data[] = s.getBytes();

OutputStream out =

**new** BufferedOutputStream(file.newOutputStream(CREATE, APPEND);

out.write(data, 0, data.length);

**Gestiunea Metadatelor**



Daca un program necesita mai multe atribute ale fisierului in acelasi timp este ineficient sa folosim metode ce returneaza un singur atribut. Accesul repetat la sistemul de fisiere pentru a obtine un singur atribut poate afecta performanta. De aceea clasa Files furnizeaza doua metode de obtinere a atributelor unui fisier intr-o singura operatie:

* readAttributes(Path, String, LinkOption...)
* readAttributes(Path, Class<A>, LinkOption...)

Exemplu:

**public** **class** FileAttributes {

**public** **static** **void** main(String[] args) {

DosFileAttributes attrs = **null**;

Path file = Paths.*get*("file.in");

**try** {

attrs = Files.*readAttributes*(file, DosFileAttributes.**class**);

} **catch** (IOException e) {

System.*out*.println("Exception reading attributes of the file: " + e);

}

System.*out*.println(file);

System.*out*.println("Creation time: " + *toDate*(attrs.creationTime()));

System.*out*

.println("Last Modified: " + *toDate*(attrs.lastModifiedTime()));

System.*out*.println("Last Access: " + *toDate*(attrs.lastAccessTime()));

**if** (!attrs.isDirectory()) {

System.*out*.println("Size (K Bytes):" + (attrs.size() / 1024));

}

System.*out*.println("DOS File information:");

System.*out*.format(

"Archive: %b Hidden: %b ReadOnly: %b System File: %b%n",

attrs.isArchive(), attrs.isHidden(), attrs.isReadOnly(),

attrs.isSystem());

}

// Utility method to print a better formatted time stamp

**public** **static** String toDate(FileTime ft) {

**return** DateFormat.*getInstance*().format(**new** Date(ft.toMillis()))

.toString();

}

}

DOS poate modifica atribute dupa ce fisierul a fost creat prin metoda setAttribute().

Cu NIO2 putem crea fisiere si directoare pe sisteme POSIX (Portable Operating System Interface) cu setari initiale de permisiune. Aceasta rezolva o problema in programare atunci este creat un fisier, si anume ca permisiunea pe fisier poate fi modificata inainte ca urmatoarea executie sa seteze permisiunea.

Permisiunea poate fi setata doar in sisteme ce implementeaza POSIX precum MacOS, Linux si Solaris. Windows bazat pe DOS nu implementeaza POSIX. Fisierele si directoarele DOS nu au permisiuni ci atribute de fisier.

Putem determina daca sistemul suporta POSIX programat prin cautarea atributelor suportate de fisier.

**public** **class** PosixFileTest {

**public** **static** **void** main(String[] args) {

**boolean** unixFS = **false**;

Set<String> views = FileSystems.*getDefault*()

.supportedFileAttributeViews();

**for** (String s : views) {

System.*out*.println(s);

**if** (s.equals("posix")) {

unixFS = **true**;

}

}

**if** (!unixFS) {

System.*out*

.println("This filesystem does not support Posix permissions.");

System.*exit*(-1);

}

}

}

**Operatii recursive**

Clasa Files furnizeaza o metoda de parcurgere a arborelui de fisiere pentru operatii recursive precum copierea si stergerea.

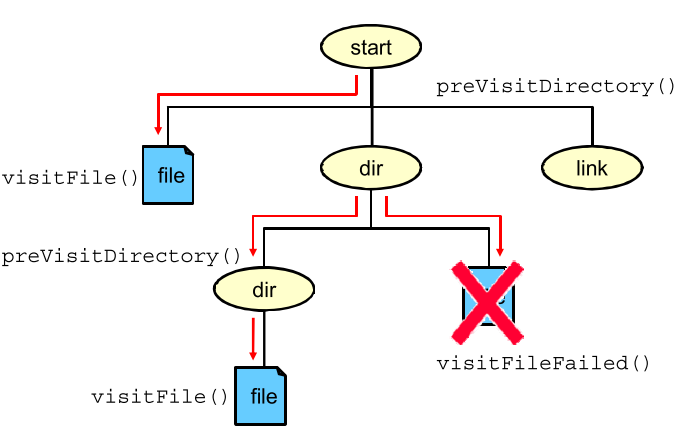
Interfata FileVisitor contine metode ce sunt invocate pe fiecare nod vizitat dintr-o arborescenta de fisiere:

* preVisitDirectory(Path, BasicFileAttributes), invocata pe un director inainte ca intrarile in director sa fie vizitate
* visitFile(Path, BasicFileAttributes), invocata pentru un fisier intr-un director
* postVisitDirectory(Path, BasicFileAttributes), invocata dupa ce toate intrarile din director si descendentii lor au fost vizitati
* visitFileFailed(Path, BasicFileAttributes), invocata pentru un fisier ce nu poate fi vizitat

Rezultatul returnat de fiecare dintre metode determina actiunile ce urmeaza a se lua dupa ce un nod a fost atins (pre sau post). Aceste actiuni sunt enumerate in clasa FileVisitResult:

* CONTINUE, continua cu nodul urmator
* SKIP\_SIBLINGS, continua fara vizitarea fratilor acestui fisier sau director
* SKIP\_SUBTREE, continua fara vizitarea intrarilor din acest director
* TERMINATE

Exista o clasa, SimpleFileVisitor, ce implementeaza metodele din FileVisitor cu tipul returnat FileVisitResult.CONTINUE sau aruncarea unei IOException.



Pornind de la primul nod director si continuand cu fiecare subdirector intalnit metoda preVisitDirectory() este invocata pe clasa trimisa metodei walkFileTree(). Presupunand ca rezultatul returnat este FileVisitResult.CONTINUE, urmatorul nod este explorat.

Observatie: traversarea arborescentei de fisiere este in adancime cu FileVisitor invocat pentru fiecare fisier intalnit. Traversarea arborescentei se incheie cand toate fisierele accesibile din arborescenta au fost vizitate sau o metode de vizitare returneaza TERMINATE. Cand o metoda de vizitare se termina din cauza unei IOException, o eroare sau o exceptie runtime, traversarea se termina si eroarea sau exceptia este propagata apelentului metodei.

Cand este intalnit un fisier in arborescenta, metoda walkFileTree() incearca sa-i citeasca BasicFileAttributes. Daca fisierul nu este director metoda visitFile() este invocata cu atributele de fisier. Daca atributele de fisier nu pot fi citite metoda visitFileFailed() este invocata cu o exceptie I/O.

Dupa vizitarea tuturor descendentilor unui nod metoda postVisitDirectory() este invocata pe fiecare director.

In exemplul urmator, clasa PrintTree implementeaza fiecare metoda din FileVisitor si afiseaza tipul, numele si marimea directorului si fisierului la fiecare nod.

**public** **class** PrintTree **implements** FileVisitor<Path> {

// Print information about the root path

@Override

**public** FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attr) {

System.*out*.print("preVisitDirectory: ");

**if** (attr.isDirectory()) {

System.*out*.format("Directory: %s ", dir);

} **else** {

System.*out*.format("Other: %s ", dir);

}

System.*out*.println("(" + attr.size() + " bytes)");

**return** *CONTINUE*;

}

// Print information about each type of file.

@Override

**public** FileVisitResult visitFile(Path file, BasicFileAttributes attr) {

System.*out*.print("visitFile: ");

**if** (attr.isSymbolicLink()) {

System.*out*.format("Symbolic link: %s ", file);

} **else** **if** (attr.isRegularFile()) {

System.*out*.format("Regular file: %s ", file);

} **else** {

System.*out*.format("Other: %s ", file);

}

System.*out*.println("(" + attr.size() + " bytes)");

**return** *CONTINUE*;

}

// Print each directory visited.

@Override

**public** FileVisitResult postVisitDirectory(Path dir, IOException exc) {

System.*out*.print("postVisitDirectory: ");

System.*out*.format("Directory: %s%n", dir);

**return** *CONTINUE*;

}

// If there is some error accessing the file, let the user know.

// If you don't override this method and an error occurs, an IOException

// is thrown.

@Override

**public** FileVisitResult visitFileFailed(Path file, IOException exc) {

System.*out*.print("vistiFileFailed: ");

System.*err*.println(exc);

**return** *CONTINUE*;

}

}

**public** **class** WalkFileTreeTest {

**public** **static** **void** main(String[] args) {

Path path = Paths.*get*("c:/chestii");

**if** (!Files.*isDirectory*(path)) {

System.*out*.println("c:/chestii" + " must be a directory!");

System.*exit*(-1);

}

**try** {

Files.*walkFileTree*(path, **new** PrintTree());

} **catch** (IOException e) {

System.*out*.println("Exception: " + e);

}

}

}

**Gasirea fisierelor**

Pentru a gasi un fisier, in mod normal, cautam un director. Putem utiliza un instrument de cautare sau o comanda precum: dir /s \*.java. Aceasta comanda va cauta recursiv in arborescenta de directoare, incepand cu elementul in care ne aflam, toate fisierele ce contin extensia java.

Interfata java.nio.file.PathMatcher include o metoda de cautare pentru a determina daca obiectul Path se potriveste unui string de cautare. Orice sistem de fisiere furnizeaza un PathMatcher ce poate fi obtinut din factory-ul FileSystems. (FileSystems.getDefault().getPathMatcher(String syntaxAndPattern))

String-ul argument este de forma syntax:pattern, unde syntax poate fi glob sau regex.

Sintaxa glob este similara expresiilor regulate, dar mai simpla. Urmatoarele reguli sunt utilizate pentru a interpreta pattern-ii glob:

* Caracterul \* potriveste zero sau mai multe caractere unui nume fara a depasi limitele directorului
* Caracterele \*\* potriveste zero sau mai multe caractere depasind limitele directorului
* ? potriveste exact un caracter al numelui
* \ este utilizat pentru caractere escape
* [] formeaza o expresie ce potriveste un singur caracter al numelui dintr-o multime de caractere. Spre exemplu[abc] potriveste a, b sauc. – poate fi utilizat pentru a specifica un doemniu. Spre exemplu [a-z] specifica domeniul, inclusiv capetele, de la a la z. Putem avea si [abce-g] insemnand a, b, c, e, f sau g. ! este folosit pentru negare. Spre exemplu [!a-c] potriveste orice caracter mai putin a, b sau c
* Intr-o expresie cu paranteze caracterele \*, ? si \ se potrivesc lor insele. – se potriveste lui insusi daca este primul caracter sau imediat dupa !
* {} formeaza un grup de subpattern-uri, unde grupul se potriveste daca orice subpattern din grup se potriveste. , este folosita pentru a separa subpattern-urile. Grupurile nu pot fi incuibate
* . in numele unui fisier este tratat ca un caracter obisnuit in expresia de potrivire

Cand syntax este regex componenta pattern este o expresie regulata asa cum a fost definita de clasa Pattern.

Exemplu:

**public** **class** Finder **extends** SimpleFileVisitor<Path> {

**private** Path file;

**private** PathMatcher matcher;

**private** **int** numMatches;

**public** Finder(Path file, PathMatcher matcher) {

**this**.file = file;

**this**.matcher = matcher;

}

//Compares the glob pattern against the file or directory name.

**private** **void** find(Path file) {

Path name = file.getFileName();

**if** (name != **null** && matcher.matches(name)) {

numMatches++;

System.*out*.println(file);

}

}

//Prints the total number of matches to standard out.

**public** **void** done() {

System.*out*.println("Matched: " + numMatches);

}

//Invoke the pattern matching method on each file.

@Override

**public** FileVisitResult visitFile(Path file, BasicFileAttributes attrs) {

find(file);

**return** *CONTINUE*;

}

//Invoke the pattern matching method on each directory.

@Override

**public** FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs) {

find(dir);

**return** *CONTINUE*;

}

}

**public** **class** FindFileTest {

**public** **static** **void** main(String[] args) {

String r = "c:/chestii";

Path root = Paths.*get*(r);

**if** (!Files.*isDirectory*(root)) {

System.*out*.println(r + " must be a directory!");

System.*exit*(-1);

}

PathMatcher matcher = FileSystems.*getDefault*().getPathMatcher(

"glob:" + "\*.doc");

Finder finder = **new** Finder(root, matcher);

**try** {

Files.*walkFileTree*(root, finder);

} **catch** (IOException e) {

System.*out*.println("Exception: " + e);

}

finder.done();

}

}

In clasa ce contine metoda main() avem doua argumente necesare cautarii. Primul este radacina arborescentei in care se face cautarea si este testat pentru a fi director. Al doilea este utilizat pentru a crea un PathMatcher cu o expresie regulata utilizand un factory FileSystems.

Clasa Finder implementeaza interfata FileVisitor, astfel incat sa poata fi trimisa metodei walkFileTree(). Aceasta clasa este utilizata pentru a apela metoda de potrivire pe fiecare dintre fisierele vizitate in arborescenta.

**Alte clase in NIO2**

Clasa FileStore este folosita pentru a furniza operatii de utilizare ale sistemului de fisiere, precum spatiul de disk total, utilizabil sau alocat.

O instanta a interfetei WatchService poate fi utilizata pentru a raporta modificari de inregistrat obiectelor Path. WatchService poate fi utilizata pentru a identifica cand fisierele sunt adaugate, sterse sau modificate intr-un director.

Exemplu:

**public** **class** DiskUsage {

**static** **final** **long** *K* = 1024;

**static** **void** printFileStore(FileStore store) **throws** IOException {

**long** total = store.getTotalSpace() / *K*;

**long** used = (store.getTotalSpace() - store.getUnallocatedSpace()) / *K*;

**long** avail = store.getUsableSpace() / *K*;

String s = store.toString();

**if** (s.length() > 20) {

System.*out*.println(s);

s = "";

}

System.*out*.format("%-20s %12d %12d %12d\n", s, total, used, avail);

}

**public** **static** **void** main(String[] args) **throws** IOException {

System.*out*.format("%-20s %12s %12s %12s\n", "Filesystem", "kbytes",

"used", "avail");

FileSystem fs = FileSystems.*getDefault*();

**for** (FileStore store : fs.getFileStores()) {

*printFileStore*(store);

}

}

}

Si un exemplu de folosire a observatorilor:

**public** **class** WatchDir {

**private** **final** WatchService watcher;

**private** **final** Map<WatchKey, Path> keys;

**private** **final** **boolean** recursive;

**private** **boolean** trace = **false**;

// An example of a Generic method

@SuppressWarnings("unchecked")

**static** <T> WatchEvent<T> cast(WatchEvent<?> event) {

**return** (WatchEvent<T>) event;

}

/\*\*

\* Register the given directory with the WatchService

\*/

**private** **void** register(Path dir) **throws** IOException {

WatchKey key = dir.register(watcher, *ENTRY\_CREATE*, *ENTRY\_DELETE*,

*ENTRY\_MODIFY*);

**if** (trace) {

Path prev = keys.get(key);

**if** (prev == **null**) {

System.*out*.format("register: %s\n", dir);

} **else** {

**if** (!dir.equals(prev)) {

System.*out*.format("update: %s -> %s\n", prev, dir);

}

}

}

keys.put(key, dir);

}

/\*\*

\* Register the given directory, and all its sub-directories, with the

\* WatchService.

\*/

**private** **void** registerAll(**final** Path start) **throws** IOException {

// register directory and sub-directories

Files.*walkFileTree*(start, **new** SimpleFileVisitor<Path>() {

@Override

**public** FileVisitResult preVisitDirectory(Path dir,

BasicFileAttributes attrs) **throws** IOException {

register(dir);

**return** FileVisitResult.*CONTINUE*;

}

});

}

/\*\*

\* Creates a WatchService and registers the given directory

\*/

WatchDir(Path dir, **boolean** recursive) **throws** IOException {

**this**.watcher = FileSystems.*getDefault*().newWatchService();

**this**.keys = **new** HashMap<>();

**this**.recursive = recursive;

**if** (recursive) {

System.*out*.format("Scanning %s ...\n", dir);

registerAll(dir);

System.*out*.println("Done.");

} **else** {

register(dir);

}

// enable trace after initial registration

**this**.trace = **true**;

}

/\*\*

\* Process all events for keys queued to the watcher

\*/

**void** processEvents() {

**for** (;;) {

// wait for key to be signalled

WatchKey key;

**try** {

key = watcher.take();

} **catch** (InterruptedException x) {

**return**;

}

Path dir = keys.get(key);

**if** (dir == **null**) {

System.*err*.println("WatchKey not recognized!!");

**continue**;

}

**for** (WatchEvent<?> event : key.pollEvents()) {

Kind<?> kind = event.kind();

// TBD - provide example of how OVERFLOW event is handled

**if** (kind == *OVERFLOW*) {

**continue**;

}

// Context for directory entry event is the file name of entry

WatchEvent<Path> ev = *cast*(event);

Path name = ev.context();

Path child = dir.resolve(name);

// print out event

System.*out*.format("%s: %s\n", event.kind().name(), child);

// if directory is created, and watching recursively, then

// register it and its sub-directories

**if** (recursive && (kind == *ENTRY\_CREATE*)) {

**try** {

**if** (Files.*isDirectory*(child, *NOFOLLOW\_LINKS*)) {

registerAll(child);

}

} **catch** (IOException x) {

// ignore to keep sample readbale

}

}

}

// reset key and remove from set if directory no longer accessible

**boolean** valid = key.reset();

**if** (!valid) {

keys.remove(key);

// all directories are inaccessible

**if** (keys.isEmpty()) {

**break**;

}

}

}

}

**public** **static** **void** main(String[] args) **throws** IOException {

**boolean** recursive = **false**;

Path dir = Paths.*get*("c:/chestii");

**new** WatchDir(dir, recursive).processEvents();

}

}

In java.io.File a fost adaugata o metoda toPath() pentru a oferi compatibilitatea cu NIO2 (Path path = file.toPath()). Mai mult, putem inlocui codul existent pentru a avea o mentenanta imbunatatita: file.delete() va fi inlocuit cu:

Path path = file.toPath();

Files.delete(path);

Invers, Path furnizeaza metoda pentru a construi un File: File file = path.toFile();