Dateizugriff mit new I/O 2

Wolfgang Nast
MT AG
Ratingen

Schlüsselworte
Java, nio2, Zugriffsrechte.

Einleitung

Zugriff mit der neuen Schnittstelle. Es gibt jetzt Klassen für die unterschiedlichen DateiSysteme. Zeigen der Möglichkeiten von NTFS, UnixFs und FAT. Bei NTFS Beispiel von Hardlinks, Benutzer Ermittlung und einschränken der Berechtigung. Bei UnixFs Beispiele für Hard- und Softlinks. Verkürzen der Pfade mit beachten der Softlinks. Suchen von Dateien mit der VisitorKlasse des nio2. Überwachen der Ereignisse im Dateisystem mit der neuen API. Damit ist kein Pollen eines Verzeichnisses mehr notwendig um das Anlegen und Löschen von Dateien mit zu bekommen.

NIO 2 in Java 7

Mit new I/O 2 wird dem Programmierer die Möglichkeit gegeben auf die unterschiedlichen Fähigkeiten der Dateisysteme zuzugreifen. Neu sind die Unterstützung von Softlinks in UnixFs und die Berechtigungssysteme von NTFS und UnixFs. Dafür wurden neue Strukturen in Java eingeführt.

Das Standarddateisystem kann man mit der Methode holen:

```
FileSystem fs = FileSystems.getDefault();
```

Mit folgender Methode werden die root- Verzeichnisse geholt:

```
Iterable<Path> pList = fs.getRootDirectories();
```

Die Ordner, Dateien und Links können über die Path Klasse angesprochen werden.

Die Datei „home\unterordner\Datei.txt“ kann mit folgender Methode angesprochen werden.

```
Path p1 = fs.getPath("home”, “unterordner”, “Datei.txt”);
```

Mit dem Path Objekt kann man über die Verzeichnisse/Links iterieren.

```
Iterator<Path> i = p1.iterator();
```

Neu ist, das man auch die Links im Pfad auflösen lassen kann. Dafür gibt es die Methode toRealPath.

```
Path p2 = p1.toRealPath();
```

Dabei ist zu beachten, das nicht jeder Pfad aufgelöst werden kann, was zu einer Exception führt.

Bei NTFS-Dateisystemen, kann man den Besitzer einer Datei mit folgenden Kommandos ermitteln:

```
AclFileAttributeView acl = Files.getFileAttributeView(p2,
AclFileAttributeView.class);
    System.out.println("Besitzer = " + acl.getOwner().getName());
```

Um einen Benutzer ohne weiter Eigenschaften in einem DateiSystem zu bestimmen, gibt es folgende Methoden:

```
UserPrincipal userP = p1.getFileSystem().getUserPrincipalLookupService().
lookupPrincipalByName("Benutzer");
```

Dabei kann man nur nach dem Namen eines Benutzers suchen, hier war es „Benutzer“.

Um einer Gruppe mit einem Namen(„admin“) zu suchen gibt es folgende Methoden:

```
GroupPrincipal grP = p1.getFileSystem().getUserPrincipalLookupService()
.lookupPrincipalByGroupName("admin");
```

Die Rechte kann man sich mit folgenden Befehlen abrufen:

```
for(AclEntry aclEntry : acl.getAcl()){
   System.out.println("rechte: " + aclEntry.toString());
}
```

Ein neues Recht kann man folgendermaßen bauen:

```
AclEntry aclEnt1 = AclEntry.newBuilder()
   .setType(AclEntryType.ALLOW)
   .setPrincipal(userP)
   .setPermissions(AclEntryPermission.READ_DATA,
AclEntryPermission.WRITE_DATA)
   .build();
```

Das Einfügen der Rechte sollte abhängig von “Type” geschehen.
Bei ALLOW am Anfang, vor allen DENY-Einträgen:

```
acl.getAcl().add(0, aclEnt1);
```

Bei DENY am Ende, nach allen ALLOW-Einträgen:

```
acl.getAcl().add(aclEnt1);
```

Bei UnixFs, wird der Besitzer mit folgenden Kommandos ermittelt:

```
PosixFileAttributes at =
    Files.readAttributes(file, PosixFileAttributes.class);
System.out.println("Besitzer = " + at.getOwner().getName());
```

Die Berechtigung kann dann folgendermaßen gesetzt werden:

```
at.setPermissions(PosixFilePermissions.fromString("rw-r--r--"));
```

Die Rechte können auch als Set von den einzelnen Rechten gesetzt werden:

```
Set<PosixFilePermission> permSet = new HashSet<>();
permSet.add(PosixFilePermission.OWNER_READ);
permSet.add(PosixFilePermission.GROUP_EXECUTE);

…//weitere Rechte
at.setPermissions(permSet);
```

Zum Suchen von Dateien unter einem Pfad gibt es ein Visitor-Intreface, damit auch die Softlinks beachtet werden können. Ohne dieses Interface kann es durch die Softlinks zu einem geschlossenen Kreis kommen. Ein solcher Kreis führt, beim rekursiven Aufruf aller Kinder eines Verzeichnisses zu einer Endlosschleife, da immer eines der Kinder auf das übergeordnete Verzeichnis zeigt. Das Visitor-Interface besucht jedes Unterelement im Pfad nur einmal, auch wenn es durch Links mehrfach gefunden werden kann.

Mit folgendem Aufruf wird der Visitor „vis“ verwendet.

```
Files.walkFileTree(p1, vis);
```

Alternativ der Aufruf mit aufgelösten Links und einer Maximaltiefe von 10:

```
int tiefe = 20;
EnumSet<FileVisitOption> links = EnumSet.of(FileVisitOption.FOLLOW_LINKS);
Files.walkFileTree(p1, links, tiefe, vis);
```

Im Visitor Interface sind folgende Methoden:

```
FileVisitResult postVisitDirectory(T dir, IOException exc)
FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs)
FileVisitResult visitFile(T file, BasicFileAttributes attrs)
FileVisitResult visitFileFailed(T file, IOException exc)
```

Bei „visitFile“ kann man nach den Dateien und ihren Eigenschaften handeln, wie neue Rechte setzen oder den Besitzer überprüfen.

Bei „visitFileFailed“ kann man die Problembehandlung einbauen, wenn man keine Rechte an einem Verzeichnis oder einer Datei hat. Es ist nicht notwendig die „Suche“ abzubrechen.

Bei „preVisitDirectory“ kann man gut entscheiden, das man ein Verzeichnis auslassen will, weil es einen anderen Besitzer hat oder der Name bekannt ist, durch Rückgabe von „SKIP_SUBTREE“.

Bei „postVisitDirectory“ kann man gut auf das Verlassen eines Verzeichnisses reagieren oder einen Fehler im Durchlaufen des Verzeichnisses. Auch hier kann das Abbrechen („TERMINATE“) unerwünscht sein.

Zum Suchen in einem Pfad gibt es passend zu jedem Dateisystem einen PathMathcher. Dabei kann man die Bedingung als regulären Ausdruck(„regex“) angeben oder als Glob(„glob“) an geben.

```
PathMatcher mat = p1.getFileSystem().getPathMatcher(
      "glob: *.{java,class}");
if(mat.matches(p2){
```

Mit der „matches“ Methode wird überprüft, ob die Bedingung erfüllt ist.

Zum Überwachen eines Verzeichnisses gibt es die den WatcherService. Dieser wird passend zum Dateisystem geholt mit :

```
WatchService watcher = p1.getFileSystem().newWatchService();
p1.register(watcher, StandardWatchEventKinds.ENTRY_CREATE,
StandardWatchEventKinds.ENTRY_DELETE);
```

Nach dem man den Watcher registriert hat, muss man nur noch den Watcher abfragen, ob ein Ereignis stattgefunden hat. Dafür kann man folgenden Code verwenden:

```
for (;;) {
         WatchKey key;
         try {
            key = watcher.take();
         }
         catch (InterruptedException x) {
            return;
         }
         for (WatchEvent<?> event : key.pollEvents()) {
            @SuppressWarnings("unchecked")
            WatchEvent<Path> ev = (WatchEvent<Path>) event;
            Kind<?> kind = ev.kind();
            if (kind == OVERFLOW) {
               continue;
            }
            Path name = ev.context();
            if (ev.kind() == ENTRY_CREATE)  {
               //Datei angelegt
            }  else if (ev.kind() == ENTRY_DELETE) {
               //Datei gelöscht
            }
         }
         boolean valid = key.reset();
         if (!valid) {
            break;
         }
      }
```

Zu beachten ist, dass das Overflow-Ereignis immer kommen kann, es kann nicht herausgefiltert werden. Dies muss man selber ignorieren.

Es ist jetzt möglich auch Links zu erzeugen, wenn das Dateisystem dies unterstützt, wie UnixFs.

```
Files.createSymbolicLink(neuerLink, linkZiel);
```

Dabei ist der Pfad „neuerLink" der Name des Link und der Pfad „linkZiel" das Ziel des Links.

Es können auch Hardlinks angelegt werden, dies wird auch von NTFS unterstützt.

```
Files.createLink(neuerLink, linkZiel);
```

Dabei zeigen beide Pfade(neuerLink, linkZiel) auf die selbe Datei. Erst wenn alle Hardlinks gelöscht wurden, wird die Datei wirklich gelöscht.

Es können jetzt Dateien, Verzeichnisse und Links kopiert werden:

```
Files.copy(original, kopi, REPLACE_EXISTING, COPY_ATTRIBUTES,
NOFOLLOW_LINKS);
```

Dabei ermöglich die Option NOFOLLOW_LINKS, das der Softlink kopiert wird und nicht das Ziel des Links.

Zum Verschieben von Dateien und Verzeichnissen gibt es folgende Methode:

```
Files.move(ursprung, ziel, REPLACE_EXISTING, ATOMIC_MOVE);
```

Dabei ermöglicht die Option ATOMIC_MOVE die Datei auf einmal zu verschieben. Wenn das Verschieben nicht atomar geht, dann wird eine Exception geworfen.

Es gibt noch mehr Neuerungen, die wie Aktionen(Block von Code) die häufig vorkommen auf eine Methode zusammenfassen, wie z.B. Das Laden einer Datei in den Speicher(Files.readAllBytes(pfad)).

Kontaktadresse:
Wolfgang Nast
MT AG
Balcke-Dürr-Allee 9
D-40882 Ratingen

Telefon:	+49 (0) 2102 30961-0
Fax:	+49 (0) 2102 30961- 101
E-Mail	Wolfgang.Nast@mt-ag.com
Internet:	www.mt-ag.com