Höher, weiter, schneller

java.nio.file: Zeitgemäßes Arbeiten mit Dateien

30. August 2013 Christian Robert

©Shutterstock/Maksim Kabakou

Mit der Version 7 wurde in Java die Art und Weise, Dateien zu verwalten und Dateiinhalte zu bearbeiten, von Grund auf neu implementiert. Zwar existiert auch weiterhin die altbekannte Klasse java.io.File, zusätzlich gibt es nun jedoch im Package java.nio.file ein komplett neues und von java.io.File losgelöstes API zum Zugriff auf das Dateisystem. Ein erster Überblick über dieses neue API und die sich daraus ergebenden neuen Möglichkeiten im Vergleich zu java.io.File.

Um in Java-Versionen vor Java SE 7 mit Dateien zu arbeiten, muss die Klasse java.io.File genutzt werden. Exemplare dieser Klasse bilden jeweils den Java-Repräsentanten einer Datei im lokalen Dateisystem. Das eigentliche Konzept eines Dateisystems hingegen ist im klassischen java.io.File-API so gut wie unbekannt bzw. für den Entwickler nicht zugänglich. Zwar existiert eine Klasse namens java.io.FileSystem, diese ist jedoch package-private innerhalb von java.io und damit für eine Nutzung durch Clientcode nicht vorgesehen.

Implizit ist mit „Dateisystem“ vor Java SE 7 immer das lokale Dateisystem gemeint. Anders ausgedrückt: Ein java.io.File-Objekt repräsentiert immer eine Datei im lokalen Dateisystem. Eine Möglichkeit, den Dateibegriff abstrakter und weiter zu interpretieren, ist nicht vorgesehen. Ein Dateizugriff auf ein entferntes System (beispielsweise per FTP) lässt sich mit java.io.File nicht realisieren und muss zwangsläufig mit externen Frameworks umgesetzt werden.

Java SE 7 hingegen macht mit java.nio.file.FileSystem das Konzept eines Dateisystems auch für den Entwickler explizit verfügbar. Das lokale Dateisystem ist nur eine mögliche Implementierung; beliebige andere lassen sich vom Entwickler jederzeit dem System bekannt machen und nutzen.

Es bietet sich damit eine ganz neue Möglichkeit, Abstraktionen zur Speicherung von Inhalten zu entwickeln und zu nutzen, da nicht mehr auf jeder Ebene zwischen interner Speicherung (auf dem lokalen Dateisystem) und externer Speicherung (auf einem Ziel außerhalb des lokalen Dateisystems) unterschieden werden muss.

Zugegeben: Wirklich revolutionär ist diese Idee nicht – Frameworks wie Apache Commons VFS bieten eine entsprechende Abstraktion bereits seit einigen Jahren an. Mit java.nio.file bietet sich jedoch erstmals die Möglichkeit, dies direkt mit Java-Bordmitteln zu lösen.

Path

Das Pendant zu java.io.File im java.nio.file Package stellt das Interface java.nio.file.Path dar. Im Gegensatz zu File stellt jedoch Path zunächst keinen direkten Zugriff auf die Inhalte (bzw. Metadaten) einer Datei bereit, sondern enthält lediglich Informationen zur Lokalisierung der eigentlichen Datei im Dateisystem.

FileSystemProvider

Wie bereits erwähnt, bietet java.nio.file eine komplette und offene Verwaltung für Dateisysteme. Einstiegspunkt ist hierbei die Klasse FileSystemProvider. Implementierungen dieser Klasse sind für die tatsächliche Ausführung der I/O-Operationen auf Dateien, die durch Path-Objekte eindeutig identifiziert werden können, verantwortlich.

FileSystem

Während FileSystemProvider für die tatsächliche Implementierung der I/O-Operationen verantwortlich ist, ist es die Aufgabe des FileSystem, die dateisystemspezifische Hierarchie zu verwalten und über Path-Objekte zurückzugeben. Ein Path ist daher immer explizit einem FileSystem zugeordnet und wird von diesem (bzw. seinem FileSystemProvider) erstellt und verwaltet.

FileStore

Als Gruppierung innerhalb eines FileSystem liefert FileStore Informationen über Devices, Partitionen oder andere Arten der FileSystem-Aufteilung. Im lokalen Windows-Dateisystem existiert jeweils ein FileStore für jedes lokal eingebundene Laufwerk (C:, D: etc.).

URIs

Bereits Java SE 1.4 erlaubte es erstmals, ein java.io.File-Objekt aus einem URI zu erzeugen. Akzeptiert werden allerdings nur URIs aus dem file-Schema. Mit java.nio.file wird diese Möglichkeit weiter ausgebaut, und es werden weitere Attribute des URI genutzt.

Es kann direkt aus dem Schema des URI auf das zu verwendende Dateisystem geschlossen werden (Beispiele hierfür wären file, ftp oder jar). Ein URI erlaubt daher einen Dateinamen nicht nur relativ zu seinem Dateisystem, sondern absolut zur gesamten Hierarchie aller Dateisysteme eindeutig zu identifizieren.

Erste Zugriffe

Ein direktes Erzeugen eines Path-Interface ist nicht möglich. Wir erinnern uns: Ein Path stellt (anders als ein java.io.File) lediglich eine abstrakte Repräsentation dar, die erst im Zusammenspiel mit dem zugehörigen FileSystem bzw. FileSystemProvider sinnvoll verwendet werden kann. Ein Path benötigt (und bietet) daher immer eine Referenz auf das FileSystem, an das die eigentlichen I/O-Operationen delegiert werden können. Um einen Path einfach zu halten, können wir uns der Utility-Klasse java.nio.file.Paths bedienen:

```
1 Path path = Paths.get(URI.create("file:/C:/test.txt"));
2 System.out.println("1 "+path.getClass());
3 System.out.println("2 "+path.getFileSystem().getClass());
```

Zu vergleichen ist diese Path-Erzeugung mit:

```
1 File file = new File("C:/test.txt");
```

Führen wir die oberen Codezeilen aus, so erhalten wir als Ausgabe:

```
1 1 class sun.nio.fs.WindowsPath
2 2 class sun.nio.fs.WindowsFileSystem
```

Wir sehen, dass als konkrete Ausprägungen des Path-Interface und der abstrakten Klasse FileSystem die entsprechenden Windows-Implementierungen zurückgegeben werden.

Hinter den Kulissen führt der Aufruf von Paths.get dazu, dass alle im System vorhandenen und registrierten Implementierungen von FileSystemProvider durchlaufen werden. Jede dieser Implementierungen ist für genau ein Schema verantwortlich. Das bedeutet, dass eine Implementierung von FileSystemProvider genau dann ausgewählt werden kann, wenn ihre Schemaangaben mit dem Schema des übergebenen URI übereinstimmen. Auf dem so ermittelten FileSystemProvider wird nun die Methode getPath aufgerufen, die die tatsächliche, zum URI passende Path-Implementierung zurückliefert.

Dateiinhalte lesen und verändern

Vor Java SE 7 konnte auf Dateiinhalte mittels FileInputStream bzw. FileOutputStream zugegriffen werden. Mit java.nio.file bietet sich eine Reihe von Möglichkeiten, die je nach Anwendungsfall gewählt werden können.

Der „klassische“ Weg, die Inhalte über Input- bzw. OutputStreams byteweise zu lesen bzw. zu schreiben, lässt sich mit java.nio.file wie folgt abbilden:

Women in Tech – „Macht einen Master in Selbstdarstellung“ (https://jaxenter.de/women-in-tech-saskia-kurz-58866)

Backend meets Frontend: Web-UIs mit Java erstellen (https://jaxenter.de/backend-meets-frontend-vaadin-java-58701)

LETZTE KOMMENTARE

ios/Swift Softwareentwickler Senior 5.j Berufserfahrung... Freelancer --> eigene Firma. 12 Mitarbeiter ,1 Putzfrau 210.000.- Brutto/jahrl.

Die Top-17-Programmiersprachen: Nachfrage und Gehalt (https://jaxenter.de/die-top-17-programmiersprachen-nachfrage-und-gehalt-379#comment-171605)

Robert müller Der Grund für die meist angebrachte Skepsis gegenüber Frauen in MINT-Berufen liegt einzig und allein in dem seit Jahrzehnten wütenden (und in den letzten Jahren...

Women in Tech: Wie sexistisch ist die IT-Branche? (https://jaxenter.de/women-in-tech-sexismus-59017#comment-171453)

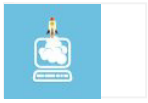
Eduard Was völlig außer Acht gelassen wird, ist die fachliche Ausbildung der entsprechenden Personen. Nach meiner Erfahrung sind viele Frauen in der IT Welt Quereinsteigerinnen. Am...

Women in Tech: Wie sexistisch ist die IT-Branche? (https://jaxenter.de/women-in-tech-sexismus-59017#comment-171394)

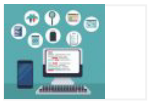
FEATURED POSTS

„Das Highlight von Eclipse Oxygen ist der Java 9 Support“ [Interview mit Mike Milinkovich] (https://jaxenter.de/eclipse-oxygen-mike-milinkovich-java-9-59437)

JAX Innovation Awards 2017 eröffnet: Nominieren Sie die innovativsten Beiträge zu Java & DevOps (https://jaxenter.de/jax-innovation-awards-2017-59335)



Spring Boot Tutorial: In 10 Schritten zur Microservices-Architektur (https://jaxenter.de/spring-boot-tutorial-microservices-cloud-foundry-kubernetes-58695)



Funktional(er) mit Java programmieren: Vavr – der Java-Slang (https://jaxenter.de/vavr-javaslang-funktional-programmieren-59108)

NEWSLETTER

(https://jaxenter.de/newsletter)

```
Path path = Paths.get(URI.create("file:/C:/test.txt"));
InputStream inStream = Files.newInputStream(path);
```

Das hier gezeigte Codebeispiel verhält sich analog zu folgendem Beispiel nach alter Vorgehensweise mit java.io.File:

```
File file = new File(C:/test.txt");
InputStream inStream = new FileInputStream(file);
```

Analog zur bereits oben gezeigten Verwendung von Paths.get führt auch die Verwendung der Methode Files.newInputStream zunächst einen Look-up auf den zum Path (bzw. dem ihm zugeordneten FileSystem) gehörigen FileSystemProvider durch. Die Erzeugung des InputStreams, von dem die Dateiinhalte gelesen werden können, übernimmt dann eben jener FileSystemProvider. Wir hätten daher auch schreiben können:

```
Path path = Paths.get(URI.create("file:/C:/test.txt"));
FileSystem fileSystem = path.getFileSystem();
FileSystemProvider provider = fileSystem.provider();
InputStream inStream = provider.newInputStream(path);
```

Eine weitere Möglichkeit, Dateiinhalte zu bearbeiten, bieten weitere Hilfsmethoden aus java.nio.files.Files. Wollen wir beispielsweise den Inhalt einer Datei als Array von Bytes weiterverwenden, so können wir die Files.copy-Methode nutzen:

```
Path path = Paths.get(URI.create("file:/C:/ test.txt"));
ByteArrayOutputStream out = new ByteArrayOutputStream();
Files.copy(path, out);
byte[] fileBytes = out.toByteArray();
```

Sehen wir uns eine vergleichbare Logik an, die das klassische java.io.File-API verwendet:

```
File file = new File("C:/test.txt");
FileInputStream in = new FileInputStream(file);
ByteArrayOutputStream out = new ByteArrayOutputStream();
for(int b = in.read(); b > -1; b = in.read()) {
  out.write(b);
}
byte[] fileBytes = out.toByteArray();
```

Es finden sich noch weitere Hilfsmethoden zum Lesen und Schreiben von Daten in der Files-Utility-Klasse, die typische I/O Use Cases abbilden.

Erweiterte Funktionalitäten

Bisher haben wir uns hauptsächlich mit den Funktionalitäten von java.nio.file.Path beschäftigt, die so oder ähnlich auch bereits in java.io.File vorhanden sind. Neben der Unterstützung für unterschiedliche Dateisysteme bietet das neue API jedoch auch weitere Funktionalitäten, die in dieser Art und Weise erstmals direkt in Java zur Verfügung gestellt werden.

Links

Links, also Verknüpfungen zwischen Dateien auf Dateisystemebene, können erst mit der Einführung von java.nio.file sinnvoll erkannt und bearbeitet werden. Das API unterstützt hierbei sowohl echte Links (auch hard links genannt) als auch symbolische Links (auch soft links genannt). Erstellt werden beide Arten von Links analog zum Auslesen von Dateien über die Utility-Klasse java.nio.file.Files:

```
Path path = Paths.get(URI.create("file:/C:/test.txt"));

Path slink = Paths.get(URI.create("file:/C:/slink.txt"));
Files.createSymbolicLink(slink, path);

Path hlink = Paths.get(URI.create("file:/C:/hlink.txt"));
Files.createLink(hlink, path);
```

Ebenfalls existiert die Möglichkeit, symbolische Links aufzulösen und auf die „echte“ Datei zu gelangen:

```
Path path = Paths.get(URI.create("file:/C:/test.txt"));
try {
  Path resolvedPath = Files.readSymbolicLink(path);
} catch(NotLinkException e) {
  System.err.println("Path is not a link");
}
```

Um die hier gezeigte Exception-Behandlung zu umgehen, lässt sich auch vor der Auflösung des Links eines Path überprüfen, ob dieser überhaupt ein Link ist:

```
Path path = Paths.get(URI.create("file:/C:/test.txt"));
if(Files.isSymbolicLink(path)) {
  doStuffWithRealFile(Files.readSymbolicLink(path));
} else {
  doStuffWithRealFile(path);
}
```


Benutzerberechtigungen

Unter Betriebssystemen, die POSIX-Dateiberechtigungen verwenden, unterstützt java.nio.file erstmals auch das Abfragen bzw. Setzen von Dateiberechtigungen. Der folgende Code zeigt das Abfragen einer Berechtigung einer Datei:

```
Path path = Paths.get(URI.create("file:/tmp/file.txt"));
Set<PosixFilePermission> permissions = Files.getPosixFilePer
boolean groupHasPermission = permissions.contains(PosixFileP
```

Auch das Setzen von Berechtigungen ist entweder programmatisch oder durch Umwandlung eines Berechtigungsstrings möglich:

```
Path path = Paths.get(URI.create("file:/tmp/file.txt"));
Set<PosixFilePermission> permissions = new HashSet<>();
permissions.add(PosixFilePermission.OWNER_READ);
permissions.add(PosixFilePermission.OWNER_WRITE);
permissions.add(PosixFilePermission.GROUP_READ);
Files.setPosixFilePermissions(path, permissions);
```

Setzen von POSIX-Dateiberechtigungen nach Umwandlung eines Berechtigungsstrings:

```
Path path = Paths.get(URI.create("file:/tmp/file.txt"));
Set<PosixFilePermission> permissions = PosixFilePermissions.
Files.setPosixFilePermissions(path, permissions);
```

Werden POSIX-Dateiberechtigungen vom entsprechenden Dateisystem nicht unterstützt, so wird eine UnsupportedOperationException der getPosixFilePermissions- bzw. setPosixFilePermissions-Methode geworfen.

Notification

Für bestimmte Einsatzzwecke kann es notwendig sein, zu erfahren, wann eine Datei innerhalb eines Verzeichnisses geändert, neu erstellt oder gelöscht wurde. Mit Java-Bordmitteln und ohne java.nio.file lässt sich dies nur durch ständiges Pollen eines Verzeichnisses und Vergleich mit einem vorherigen Zustand erreichen.

Mit java.nio.file bietet uns das API hierfür explizite Unterstützung. Wollen wir beispielsweise erfahren, wenn innerhalb eines Verzeichnisses eine neue Datei erstellt wurde, so können wir dies mit einem einfachen Codeschnipsel (Listing 1) erreichen.

Listing 1: Notification für neue Dateien einrichten

```
Path path = Paths.get(URI.create("file:/C:/Temp/"));
WatchService watchService = path.getFileSystem().newWatchSer
WatchKey watchKey = path.register(watchService, StandardWatc
while(true) {
  for(WatchEvent<?> event : watchKey.pollEvents()) {
    Path newPath = (Path)event.context();
    System.out.println("New file: " + newPath);
  }
}
```

Interoperatibilität

Java 9 für Library-Entwickler: "Die Unsafe-Klasse sollte man einfach nicht mehr nutzen" (https://jaxenter.de/java-9-interview-uwe-schindler-58580)

Auch wenn java.nio.file als kompletter Ersatz für die klassische Dateibehandlung mit java.io.File dienen kann, so wird es immer wieder Situationen geben, wo beide APIs parallel zum Einsatz kommen müssen. Der typische Anwendungsfall hierfür dürfte bestehender Code sein, für den es keinen Grund zum Refactoring gibt oder die Verwendung externer Frameworks, die noch das alte File-API nutzen. Sowohl java.io.File als auch java.nio.file.Path bieten hierzu entsprechende Konvertierungsmethoden an:

```
1 Path path = Paths.get(URI.create("file:/C:/test.txt"));
2 File file = path.toFile();
3 Path pathFromFile = file.toPath();
```

Zu beachten ist hierbei jedoch, dass nicht jedes Path-Objekt automatisch in ein File-Objekt umwandelbar ist (nicht jedes Path-Objekt repräsentiert schließlich eine Datei im lokalen Dateisystem). Im entsprechenden Fall wird beim Aufruf von Path#toFile eine UnsupportedOperationException geworfen.

Fazit

Mit dem neuen java.nio.file-API führt Java SE 7 ein mächtiges und leistungsfähiges neues API zur Dateiverwaltung und Dateibearbeitung ein. Wir haben einige der grundlegenden Funktionalitäten gezeigt und die hierdurch gebotenen Möglichkeiten gesehen. Es existieren allerdings noch eine Reihe weiterer interessanter Optionen (wie Traversierung), auf die wir hier nicht näher eingegangen sind.

Durch die zusätzlichen Abstraktionsschichten wirkt manches auf den ersten Blick im Vergleich zum java.io.File-API noch ungewohnt, aber nach kurzer Einarbeitungszeit wird man die neuen Möglichkeiten schätzen und nicht mehr missen wollen. Ein kleiner Wermutstropfen bleibt die Tatsache, dass nun zwei konkurrierende APIs existieren, die Dateiverwaltung und -bearbeitung erlauben. Da jedoch java.nio.file alle bisherigen Anwendungsfälle – und einiges darüber hinaus – abbilden kann, ist bei neuen Projekten ein Wechsel mehr als anzuraten.

Aufmacherbild: Finance concept: computer keyboard with Folder (http://www.shutterstock.com/cat.mhtml?lang=de&search_source=search_form&search_tracking_id=zsjL_K5LiRlL4SHmSaiMiA&version=llv1&anyorall=all&safesearch=1&searchterm=ordner&s 1-76) von Shutterstock / Urheberrecht: Maksim Kabakou

Links & Literatur

http://docs.oracle.com/javase/tutorial/essential/io/fileio.html
http://docs.oracle.com/javase/tutorial/essential/io/fileio.html
(http://docs.oracle.com/javase/tutorial/essential/io/fileio.html)

GESCHRIEBEN VON

Christian Robert

Christian Robert ist Senior Developer für Mobile Lösungen bei SapientNitro in Köln. Seit über zehn Jahren beschäftigt er sich mit der Konzeption und Entwicklung von Individualsoftware im Java-Umfeld. Seine aktuellen Schwerpunkte liegen in der Entwicklung von pragmatischen und dennoch (oder gerade deswegen) effizienten Softwarelösungen im mobilen Umfeld. Außerdem interessiert er sich intensiv für die Ideen der Software Craftsmanship Bewegung.

Alle Beiträge von Christian Robert (https://jaxenter.de/author/christianrobert)

DAS KÖNNTE SIE AUCH INTERESSIEREN!

Bean-Validierung mit OVal (https://jaxenter.de/bean-validierung-mit-oval-5369)

Der ewige Kampf gegen Redundanzen (https://jaxenter.de/der-ewige-kampf-gegen-redundanzen-2-5465)

Die Git-Revolution (https://jaxenter.de/die-git-revolution-7008)

KOMMENTARE

Schreibe einen Kommentar

Deine E-Mail-Adresse wird nicht veröffentlicht.

Name

E-Mail

Website

Kommentar abschicken