Files

Класс: `java.nio.file.Files`

Описание

Класс, состоящий из статических методов для работы с файловой системой, включая обход каталогов, получение информации о файлах и каталогах, операции над файлами и каталогами как элементами ФС (создание, копирование, перемещение, удаление), а также файловый ввод-вывод.

Все методы класса `Files` работают с объектами типа `Path`. Почти все эти методы также объявлены как `throws IOException`.

Информация о файлах и каталогах

Следующие методы возвращают некоторую информацию о переданном пути, позволяя узнать, указывает ли он вообще на какой-либо элемент ФС, а также узнать свойства этого элемента.

`static boolean exists(Path path, LinkOption... options)`
: Возвращает `true`, если путь указывает на существующий файл или каталог.

`static boolean notExists(Path path, LinkOption... options)`
: Возвращает `true`, если путь указывает на несуществующий файл или каталог.

`static boolean isRegularFile(Path path, LinkOption... options)`
: Возвращает `true`, если путь указывает на существующий файл (и это обычный файл, а не каталог или особый псевдофайл).

`static boolean isDirectory(Path path, LinkOption... options)`
: Возвращает `true`, если путь указывает на существующий каталог (и это обычный каталог, а не файл или особый псевдофайл).

`static boolean isReadable(Path path)`
: Возвращает `true`, если файл существует и доступен для чтения.

`static boolean isWritable(Path path)`
: Возвращает `true`, если файл существует и доступен для записи.

`static boolean isExecutable(Path path)`
: Возвращает `true`, если файл существует и доступен для выполнения.

`static long size(Path path) throws IOException`
: Возвращает размер файла в байтах. Предполагает, что файл существует и доступен для чтения.

Необязательный список параметров `LinkOption... options` предназначен для работы с *символическими ссылками* (symbolic links, symlinks), и обычно вместо них можно ничего не указывать.

```
Files.exists(Paths.get("C:\\windows\\system32"))     // true
Files.exists(Paths.get("C:\\file\\not\\found"))      // наверное, false
Files.isReadable(Paths.get("/usr/bin/firefox"))      // true
Files.isWritable(Paths.get("/usr/bin/firefox"))      // false, если вы не root
Files.isExecutable(Paths.get("/usr/bin/firefox"))    // true
Files.size(Paths.get("/home/user/eclipse/eclipse"))  // 79058
```

Важно! Будьте осторожны с этими методами в приложениях, чувствительных к безопасности. Результаты их выполнения **сразу же устаревают**. Дело в том, что файловая система является *разделяемым ресурсом*, совместно используемым всеми процессами и потоками в ОС. Состояние файла может измениться немедленно после получения информации о нём; файл может быть удалён, перемещён, его права доступа и размер могут измениться и т.д. Например, если звёзды сойдутся не так, следующий код упадёт по `NoSuchFileException`:

```
Path file = Paths.get("myfile.txt");
byte[] buffer = null;

// Поток 1                                  // Поток 2
if (Files.isReadable(file)) {
    // тут управление получает поток 2
                                            Files.delete(file);
                                            // управление в поток 1
    buffer = Files.readAllBytes(file);
}
```

Такая разновидность *состояния гонки* (race condition) называется ошибкой *TOCTTOU* (time of check to time of use — от времени проверки до времени использования). При операциях с файловой системой принято *просить прощения, а не разрешения* (ask for forgiveness, not permission), то есть не проверять, допустима ли операция, а пытаться выполнить операцию безусловно и обрабатывать исключение при провале:

```
try {
    buffer = Files.readAllBytes(file);
} catch (IOException e) {
    // ...
}
```

Получение содержимого каталога

Семейство методов `newDirectoryStream` открывает каталог и возвращает все найденные в нём элементы, включая файлы, подкаталоги и псевдофайлы.

```
static DirectoryStream<Path> newDirectoryStream(Path dir) throws IOException
static DirectoryStream<Path> newDirectoryStream(Path dir, String glob)
    throws IOException
static DirectoryStream<Path> newDirectoryStream(Path dir,
    DirectoryStream.Filter<? super Path> filter) throws IOException
```

Естественно, объект `dir` должен указывать на каталог, иначе нам прилетит `NotDirectoryException`.

Разберёмся с этими методами по порядку.

Все они возвращают объект типа `DirectoryStream<Path>`. Это несколько странный

`Iterable<Path>`, по которому можно пройтись циклом for-each (или вручную итератором) только один раз, после чего его нужно закрыть методом `close`. Проще всего обеспечить его закрытие с помощью блока `try` с ресурсами:

```
Path dir = Paths.get("C:\\docs");

try (DirectoryStream<Path> files = Files.newDirectoryStream(dir)) {
    for (Path file : files) {
        System.out.println(file);
    }
}
```

Почему у `DirectoryStream` такой неудобный синтаксис? Дело в том, что возвращаемый им итератор ленивый: он не получает список всех файлов в каталоге сразу, а запрашивает имя каждого следующего файла у ФС при вызове `next`. Это сделано для оптимизации; если же мы действительно хотим запросить у ФС список именно *всех* файлов в каталоге, можно преобразовать `DirectoryStream` в `List`, с которым гораздо удобнее работать:

```
void List<Path> listDirectory(Path dir) throws IOException {
    List<Path> result = new ArrayList<>();

    try (DirectoryStream<Path> files = Files.newDirectoryStream(dir)) {
        files.forEach(result::add);
    }

    return result;
}
```

Полученный список можно, например, отсортировать — `DirectoryStream` возвращает элементы каталога без сортировки, в том же порядке, в котором их возвращают низкоуровневые системные вызовы ФС.

Итак, мы разобрались с методом `newDirectoryStream` с одним параметром. Он возвращает *все* элементы каталога. Два оставшихся варианта метода фильтруют возвращаемую последовательность файлов. В частности, параметр `String glob` задаёт хорошо известные нам по файловым менеджерам файловые маски, а точнее, чуть более мощный их вариант, документацию по которым можно посмотреть в описании метода `FileSystem.getPathMatcher`:

```
Files.newDirectoryStream(dir, "*.txt")         // все файлы .txt
Files.newDirectoryStream(dir, "*.{html|pdf}")  // все файлы .html и .pdf
```

Важно! В Windows по историческим причинам файловая маска `*.*` означает "все файлы". В Java она имеет то же значение, что и в Unix: все файлы, содержащие точку в имени. Например, выражение

```
Files.newDirectoryStream(Paths.get("C:"), "*.*")
```

не найдёт файл `bootmgr` и каталоги `Windows`, `Program Files` и т.д., но найдёт файл `pagefile.sys`. Чтобы найти действительно *все* файлы, можно использовать файловую маску `*`, как и в Unix.

Наконец, третий и последний вариант метода `newDirectoryStream` выглядит страшно, но на самом деле в нём нет ничего сложного. Вместо файловой маски третьим параметром передаётся

объект интерфейса `DirectoryStream.Filter<Path>`, состоящего из единственного метода:

`boolean accept(Path entry)`

`DirectoryStream` вернёт только те элементы каталога, для которых переданный фильтр вернёт `true`. Поскольку у этого интерфейса только один абстрактный метод, он является функциональным и может быть реализован лямбда-выражением:

```
// Вернуть все файлы в каталоге, но не подкаталоги
Files.newDirectoryStream(dir, Files::isRegularFile)

// Вернуть все подкаталоги в каталоге, но не файлы
Files.newDirectoryStream(dir, Files::isDirectory)

// Вернуть все файлы, доступные для чтения и записи
Files.newDirectoryStream(dir,
    file -> Files.isReadable(file) && Files.isWritable(file))

// Вернуть все файлы не менее 4 МБ размером.
// К сожалению, много писанины из-за IOException
Files.newDirectoryStream(dir, file -> {
    try {
        return Files.size(file) >= 4 * 1024 * 1024;
    } catch (IOException e) {
        throw new UncheckedIOException(e);
    }
})
```

Класс `UncheckedIOException` специально предназначен для того, чтобы заворачивать проверяемое исключение `IOException` в непроверяемое при работе с интерфейсами, которые не могут бросать проверяемые исключения. Естественно, его не стоит забывать ловить во внешнем коде и обрабатывать (или разворачивать завёрнутый `IOException` через `getCause` и бросать во внешний код уже его.

Важно! К сожалению, термин "stream" (поток) в Java 7 был перегружен, а в Java 8 — ещё более. Тип `DirectoryStream` не имеет отношения ни к потокам ввода-вывода `InputStream` и `OutputStream`, ни к потокам вычислений `Stream`. Кстати, в Java 8 можно пользоваться `Stream` вместо `DirectoryStream` с помощью метода `Files.list` и использовать при получении элементов каталога всю мощь нового потокового API:

`Stream<Path> list(Path dir) throws IOException`

Полученный поток `Stream` нужно закрыть, поскольку он оборачивает `DirectoryStream`. Единственная тонкость состоит в том, что его методы выбрасывают `UncheckedIOException`, который нужно разворачивать в обычный `IOException`.

```
List<Path> getSubdirectoriesSorted(Path dir) throws IOException {
   try (Stream<Path> stream = Files.list(dir)) {
       return stream.filter(Files::isDirectory).sorted()
               .collect(Collectors.toList());
   } catch (UncheckedIOException e) {
       throw e.getCause();
   }
}
```

Файловые операции

Класс `Files` поддерживает все обычные операции над файлами и каталогами:

`static void delete(Path path) throws IOException`
: Удаляет файл или **пустой** каталог. Ругается (`NoSuchFileException`), если такого файла или каталога нет.

`static boolean deleteIfExists(Path path) throws IOException`
: Удаляет файл или каталог, если они существуют. Возвращает `true`, если файл был удалён, и `false`, если он и не существовал.

`static void move(Path source, Path target, CopyOption... options) throws IOException`
: Переименовывает файл или каталог и/или перемещает его в другой каталог.

`static Path createDirectory(Path dir, FileAttribute<?>... attrs) throws IOException`
: Создаёт каталог, ругается (`FileAlreadyExistsException`), если он уже существует.

`static Path createDirectories(Path dir, FileAttribute<?>... attrs) throws IOException`
: Создаёт каталог и все его родительские каталоги, по необходимости. Ничего не делает для уже существующих каталогов. (Аналог команды `mkdir -p` в Unix.)

`static Path createFile(Path path, FileAttribute<?>... attrs) throws IOException`
: Создаёт пустой файл, ругается (`FileAlreadyExistsException`), если файл с таким именем уже существует.

`static Path createTempDirectory(Path dir, String prefix, FileAttribute<?>... attrs) throws IOException`
`static Path createTempDirectory(String prefix, FileAttribute<?>... attrs) throws IOException`
`static Path createTempFile(Path dir, String prefix, String suffix, FileAttribute<?>... attrs) throws IOException`
`static Path createTempFile(String prefix, String suffix, FileAttribute<?>... attrs) throws IOException`
: Создание временных файлов и каталогов. Удалять их придётся вручную. Необязательный первый параметр типа `Path` задаёт каталог, в котором будет создан временный файл или каталог. Если он не задан, используется стандартный системный каталог для временных файлов (`/tmp` в Unix, `%TEMP%` в Windows).

`static void copy(Path source, Path target, CopyOption... options) throws IOException`
`static void copy(InputStream in, Path target, CopyOption... options) throws IOException`
`static void copy(Path source, OutputStream out, CopyOption... options) throws IOException`
: Копирует файл, либо перезаписывает файл содержимым потока ввода, либо записывает содержимое файла в поток вывода.

В методах `move` и `copy` можно использовать следующие необязательные константы после имени файла:

`StandardCopyOption.ATOMIC_MOVE`
: По возможности перемещает файл как атомарное действие. Это возможно только в случае, если файл перемещается на тот же раздел диска.

`StandardCopyOption.COPY_ATTRIBUTES`
: Копирует не только содержимое, но и атрибуты старого файла в новый.

`StandardCopyOption.REPLACE_EXISTING`
: Перезаписывать файл, если он уже существует на новом месте. По умолчанию, если файл существует, методы `move` и `copy` выбрасывают `FileAlreadyExistsException`.

"Однострочный" файловый ввод-вывод

В классе `Files` имеются удобные методы, позволяющие прочитать или записать всё содержимое файла за одну операцию. Эти методы удобны тем, что при их использовании не

приходится явно работать с потоками ввода-вывода и закрывать их методом `close`. Но при этом, конечно, эти методы менее эффективны по потреблению памяти, чем более низкоуровневые операции ввода-вывода, потому что читают всё содержимое файла в память.

`static byte[] readAllBytes(Path file) throws IOException`

Читает всё содержимое файла как массив байт.

`static List<String> readAllLines(Path file) throws IOException`
`static List<String> readAllLines(Path file, Charset cs) throws IOException`

Читает всё содержимое файла как список строк в указанной кодировке (по умолчанию — UTF-8).

Вот так можно вывести всё содержимое текстового файла в консоль:

```
Path file = Paths.get("C:\\file.txt");
Files.readAllLines(file).forEach(System.out::println);
```

А что делать, если нужно прочитать всё содержимое файла в виде одной строки? Здесь есть два способа:

- Создать строку из массива байт с помощью конструктора класса `String`:

```
new String(Files.readAllBytes(file), StandardCharsets.UTF_8)
```

- Склеить прочитанные строки статическим методом `String.join`:

```
String.join("\n", Files.readAllLines(file))
```

Разница в этих методах состоит в том, что в первом случае строка будет раскодирована в точности из той последовательности байт, которая содержалась в исходном файле, и переводы строки будут зависеть от ОС, в которой этот файл был создан (`"\n"` для Unix, `"\r\n"` для Windows), а во втором случае строки будут склеены в точности переданной строкой. Кстати, для получения системного разделителя строк можно использовать статический метод `System.lineSeparator()`.

Аналогично чтению выполняется и запись массива байт или списка строк:

`static Path write(Path path, byte[] bytes, OpenOption... options) throws IOException`

`static Path write(Path path, Iterable<? extends CharSequence>, OpenOption... options) throws IOException`

`static Path write(Path path, Iterable<? extends CharSequence>, Charset cs, OpenOption... options) throws IOException`

(Не обращайте пока внимания на страшное объявление `Iterable<? extends CharSequence>`. Старый добрый `List<String>` вполне подойдёт.)

Последний список параметров `OpenOption` можно оставить пустым.

Вот так мы можем программно сгенерировать, откомпилировать и запустить Java-файл:

```
List<String> javaSource = Arrays.asList(
    "public class Hello {",
    "    public static void main(String[] args) {",
    "        System.out.println(\"Hello World!\");",
    "    }",
    "}"
);

Path file = Paths.get("Hello.java");
Files.write(file, javaSource);

JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
compiler.run(null, null, null, file.toString());

new ProcessBuilder("java", "Hello")
    .directory(file.getParent().toFile())
    .inheritIO()
    .start();
```

Текстовый ввод-вывод

Для построчного текстового ввода-вывода в Java служат интерфейсы `Reader` и `Writer`. В классе `Files` есть статические методы, возвращающие буферизованные реализации этих интерфейсов: `BufferedReader` и `BufferedWriter`.

`static BufferedReader newBufferedReader(Path path) throws IOException`

`static BufferedReader newBufferedReader(Path path, Charset cs,`
`    OpenOption... options) throws IOException`

Если кодировка не задана, используется UTF-8. После использования полученный объект нужно закрыть. Проще всего это сделать через блок `try` с ресурсами:

```
try (BufferedReader br = Files.newBufferedReader(file)) {
    String line;

    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }
}
```

Для удобства можно обернуть возвращённый `BufferedReader` в более мощный класс `Scanner` и закрывать уже его:

```
try (Scanner sc = new Scanner(Files.newBufferedReader(file))) {
    while (sc.hasNextLine()) {
        System.out.println(sc.nextLine());
    }
}
```

Аналогично `BufferedReader` можно пользоваться классом `BufferedWriter` для записи строк в файл.

`static BufferedWriter newBufferedWriter(Path path,`
`    OpenOption... options) throws IOException`

`static BufferedWriter newBufferedWriter(Path path, Charset cs,`
`    OpenOption... options) throws IOException`

Как и в случае `newBufferedReader`, по умолчанию используется кодировка UTF-8. По умолчанию, если параметры `OpenOption` не заданы, файл создаётся, если он не существует, и очищается, если существует.

```
try (BufferedWriter bw = Files.newBufferedWriter(file)) {
    bw.write("Строка 1\n");
    bw.write("Строка 2\n");
    bw.write("Строка 3\n");
}
```

Класс `BufferedWriter` не очень удобен для использования: он позволяет записывать только целые строки и заставляет явно добавлять перевод строки к каждой из них. Для удобства можно обернуть его в класс `PrintWriter`, имеющий привычные методы `print`, `println` и `printf`:

```
try (PrintWriter pw = new PrintWriter(Files.newBufferedWriter(file))) {
    pw.print("Строка 1\n");
    pw.println("Строка 2");
    pw.printf("Строка %d\n", 3);
}
```