NIO2 (Java, обучающая статья)

Вопросы рассматриваемые в статье:

- Использование интерфейса Path

- Использование класса Files

- Копирование файлов

- Перемещение файла

- Удаление файла

- Обход дерева файлов

- Доработка копирования файлов

- Поиск файлов

- Отслеживание изменений в каталоге

- Files.walk – примеры:

1. List all files

2. List all folders or directories

3. Find files by file extension

4. Find files by filename

5. Find files by file size

6. Update all files last modified date

Предисловие

За основу данной статьи была взята информация из 9-ой главы книги «Oracle Certified Professional Java SE 7 Programmers Exams 1Z0-804 and 1Z0-805». Она была немного изменена (кое-где обрезана, а кое-где дополнена с помощью Google и Википедии). Здесь показаны далеко не все возможности NIO — для более подробной информации следует обратиться к официальной документации. Приятного прочтения.

Немного терминологии

**Интерфейс программирования приложений** (иногда интерфейс прикладного программирования) (англ. application programming interface, API) — набор готовых классов, процедур, функций, структур и констант, предоставляемых приложением (библиотекой, сервисом) для использования во внешних программных продуктах.

**I/O** (input/output, Ввод-вывод ) — взаимодействие между обработчиком информации и её поставщиком и/или получателем. Ввод — сигнал или данные, полученные обработчиком, а вывод — сигнал или данные, посланные им (или из него).

**NIO** (/\*в контексте Java\*/ Non-blocking I/O, New I/O) — коллекция прикладных программных интерфейсов для языка Java, предназначенных для реализации высокопроизводительных операций ввода-вывода. Также встречается аббревиатура **NIO.2** – она относится к нововведениям относительно этого направления в Java 7.

**Символьная ссылка**(Symbolic link, симлинк) — специальный файл в файловой системе, содержащий только текстовую строку с указателем. Эта строка трактуется как путь к файлу, который должен быть открыт при попытке обратиться к данному файлу.

**Абсолютный путь** — это путь, который указывает на одно и то же место в файловой системе, вне зависимости от текущей директории. Полный путь всегда начинается с корневого каталога.

**Относительный путь** — это путь по отношению к текущему рабочему каталогу.

Немного истории

Изначально Java предоставляла класс File (в пакете java.io) для доступа к файловым системам. Этот класс представляет файл/каталог в файловой системе и позволяет выполнять такие операции, как проверка на существование файла/каталога, получении свойств, и удаление файла/каталога. Тем не менее, первый вариант API не был достаточен для удовлетворения потребностей разработчиков. Ощущалась явная необходимость доработки I/O API.

Краткий список недостатков первой I/O API:

* Классу File не хватало функциональности. Например не было метода copy для копирования файла/каталога.
* В классе File определено много методов, которые возвращают Boolean-значение. В случае ошибки, возвращалось false, а не бросалось исключение, что затрудняло обнаружение и исправление ошибок.
* Класс File не предоставляет хорошей обработки символьных ссылок.
* Класс File обрабатывает файлы/каталоги неэффективно (проблемы с масштабированием);
* Класс File предоставляет доступ к ограниченному набору атрибутов файлов, который зачастую недостаточен.

Для преодоления этих проблем, в Java 4 введен NIO (New IO). **Ключевые особенности NIO:**

* **Каналы и селекторы**: NIO поддерживает различные типы каналов. Канал является абстракцией объектов более низкого уровня файловой системы (например, отображенные в памяти файлы и блокировки файлов), что позволяет передавать данные с более высокой скоростью. Каналы не блокируются и поэтому Java предоставляет еще такие инструменты, как селектор, который позволяет выбрать готовый канал для передачи данных, и сокет, который является инструментом для блокировки.
* **Буферы:** в Java 4 была введена буферизация для всех классов-обёрток примитивов (кроме Boolean). Появился абстрактный класс Buffer, который предоставляет такие операции, как clear, flip, mark и т.д. Его подклассы предоставляют методы для получения и установки данных.
* **Кодировки**: в Java 4 появились кодировки (java.nio.charset), кодеры и декодеры для отображения байт и символов Unicode.

В Java 7 был введён пакет **NIO2** - **java.nio.file** для лучшей поддержки и обработки символьных ссылок, полного доступа к атрибутам и работы с файловой системой через интерфейсы или классы, такие как **Path**, **Paths**, and **Files**.

Главное будет не путать **File** с **Files**, **Path** с **Paths**, и **FileSystem** c **FileSystems**:

- **File** старый класс (Java 4), а **Files** был введен в Java 7.

- интерфейс **Path** представляет собой путь к файлу/каталогу и определяет полезный список методов, а класс **Paths** является служебным классом, который предоставляет два метода для получения объекта типа Path.

- класс **FileSystems** предоставляет список фабричных методов для получения класса FileSystem, в то время как **FileSystem** предоставляет набор методов, для получения информации о файловой системе.

Использование интерфейса Path

В NIO2 (Java 7) представляет новую абстракцию для пути, а именно интерфейс Path. Он используется в новых функциях и API, по всему NIO.2. Объект пути содержит имена каталогов и файлов, которые составляют полный путь до файла/каталога, представленного объектом Path; Path содержит методы для извлечения элементов пути, манипуляций с ними и их добавления.

Существование соответствующих файлов и катологов (test и testfile.txt) в файловой системе не обязательно. В этом примере создаётся объект Path и извлекается основная информация, связанная с ним:

**package** test;   
   
**import** java.nio.file.\*;   
   
**public** **class** Test1 {   
    **public** **static** **void** main(String[] args) {   
           
        //Cоздание объекта Path через вызов статического метода get() класса Paths

        //Пример строки создания объекта Path пути для запуска в Windows   
        Path testFilePath = Paths.get("D:\\test\\testfile.txt");   
   
        //Вывод инормации о файле   
        System.out.println("Printing file information: ");   
        System.out.println("\t file name: " + testFilePath.getFileName());   
        System.out.println("\t root of the path: " + testFilePath.getRoot());   
        System.out.println("\t parent of the target: "   
                + testFilePath.getParent());   
   
        //Вывод элементов пути   
        System.out.println("Printing elements of the path: ");   
        **for** (Path element : testFilePath) {   
            System.out.println("\t path element: " + element);   
        }   
    }   
}

Вывод получается такой:

Printing file information:    
     file name: testfile.txt   
     root of the path: /   
     parent of the target: /home/heorhi   
Printing elements of the path:    
     path element: home   
     path element: heorhi   
     path element: testfile.txt

*Пояснения к коду:*

* Сначала создаётся объект класса Path, с использованием метода get () класса. Данный метод принимает строку, содержащую путь.
* Обратите внимание на использование управляющего символа '\' в Paths.get("D:\\test\\testfile.txt") для Windows-систем. Без него '\t' будет интерпретироваться как символ табуляции, что приведёт к java.nio.file.InvalidPathException при запуске программы, т.к. символы табуляции не могут содержаться в путях. (На мой взгляд пути Linux-систем более удобны для программистов, т.к. используют символ слэша, не требующий экранирования.)
* Затем происходит извлечение имени файла с использованием метода getFilename() объекта Path
* Далле используется метод getRoot() для получения корневого элемента объекта Path и метод getParent() для получения родительской директории целевого файла.
* В конце данного примера происходит обход елементов пути с помощью цикла foreach. Как альтернативу можно использовать обычный цикл и методы getNameCount() (для получения числа элементов в пути) и getName(index) (для получения элемента по индексу).

Перейдём к другому*примеру* который включает получение абсолютного пути от относительного пути и нормализацию пути:

**package** test;   
   
**import** java.io.IOException;   
**import** java.nio.file.\*;   
   
**class** Test2 {   
    **public** **static** **void** main(String[] args) **throws** IOException {   
                   
        //Пример строки пути для запуска в Windows   
        Path testFilePath = Paths.get(".\\Test");   
           
        System.out.println("The file name is: " + testFilePath.getFileName());   
        System.out.println("It's URI is: " + testFilePath.toUri());   
        System.out.println("It's absolute path is: "   
                + testFilePath.toAbsolutePath());   
        System.out.println("It's normalized path is: "   
                + testFilePath.normalize());   
   
        //Получение другого объекта строки по нормализованному относительному пути   
        Path testPathNormalized = Paths   
                .get(testFilePath.normalize().toString());   
        System.out.println("It's normalized absolute path is: "   
                + testPathNormalized.toAbsolutePath());   
        System.out.println("It's normalized real path is: "   
                + testFilePath.toRealPath(LinkOption.NOFOLLOW\_LINKS));   
    }   
}

*Пояснения к коду:*

* Метод toUri() возвращает URI (путь который может быть открыт из браузера).
* Метод toAbsolutePath() возвращает абсолютный путь от данного относительного пути. В случае, если был введён абсолютный путь, метод вернёт его же.
* Метод normalize() выполняет нормализацию пути, другими словами удаляет ненужные символы (такие как “ . ” и “ .. ”) из объекта Path.
* Метод toRealPath () возвращает абсолютный путь от полученного пути (как toAbsolutePath ()) и нормализует его (как normalize()). Кроме того, если все параметры выбраны правильно, то он может даже работать с символьными ссылками. Однако, для этого метода необходимо, чтобы конечный файл/каталог существовал в файловой системе (это не является обязательным условием для других методов Path).

Пример вывода при выполнении данного кода (файл Test **не должен** существовать в файловой системе для аналогичного вывода):

The file name is: Test   
It's URI is: file:///home/heorhi/workspace/OCPJP/./Test   
It's absolute path is: /home/heorhi/workspace/OCPJP/./Test   
It's normalized path is: Test   
It's normalized absolute path is: /home/heorhi/workspace/OCPJP/Test   
Exception in thread "main" java.nio.file.NoSuchFileException: /home/heorhi/workspace/OCPJP/Test   
    at sun.nio.fs.UnixException.translateToIOException(UnixException.java:86)   
    at sun.nio.fs.UnixException.rethrowAsIOException(UnixException.java:102)   
    at sun.nio.fs.UnixException.rethrowAsIOException(UnixException.java:107)   
    at sun.nio.fs.UnixPath.toRealPath(UnixPath.java:876)   
    at test.Test2.main(Test2.java:23)

(Пользователи Windows-систем получат sun.nio.fs.WindowsException.translateToIOException вместо sun.nio.fs.UnixException.translateToIOException и т. д. и т.п.)

Пример вывода при выполнении данного кода (файл Test **должен** существовать в файловой системе для аналогичного вывода):

The file name is: Test   
It's URI is: file:///home/heorhi/workspace/OCPJP/./Test/   
It's absolute path is: /home/heorhi/workspace/OCPJP/./Test   
It's normalized path is: Test   
It's normalized absolute path is: /home/heorhi/workspace/OCPJP/Test   
It's normalized real path is: /home/heorhi/workspace/OCPJP/Test

Интерфейс Path содержит два метода для сравнения объектов Path: equals() and compareTo(). Метод equals() сравнивает пути и возвращает Boolean. Метод compareTo() сравнивает пути посимвольно и возвращает: 0, если пути равны; отрицательное целое значение, если путь в объекте вызывающем метод лексикографически меньше пути в объекте, переданном в качестве параметра; положительное целое значение в противоположном случае.

*Пример кода:*

**package** test;   
   
**import** java.nio.file.\*;   
   
**class** Test3 {   
    **public** **static** **void** main(String[] args) {   
        Path path1 = Paths.get("Test");   
        Path path2 = Paths.get("/home/heorhi/workspace/OCPJP/Test");   
        System.out.println("(path1.compareTo(path2) == 0) is: "   
                + (path1.compareTo(path2) == 0));   
        System.out.println("path1.equals(path2) is: " + path1.equals(path2));   
        System.out.println("path2.equals(path1.toAbsolutePath()) is "   
                + path2.equals(path1.toAbsolutePath()));   
           
        System.out.println(path1.toAbsolutePath());   
    }   
}

Последний System.out.println является подсказкой — он выводит путь, который должен быть передан в path2 для аналогичного вывода. Вывод должен получится такой:

(path1.compareTo(path2) == 0) is: **false**   
path1.equals(path2) is: **false**   
path2.equals(path1.toAbsolutePath()) is **true**   
/home/heorhi/workspace/OCPJP/Test

Использование класса Files

Рассмотрим класс Files (введён в Java 7, находится в пакете java.nio.file), который можно использовать для выполнения различных операций с файлами и каталогами. Files является служебным классом, это означает, что это final-класс с private-конструктором и содержит только статические методы. В этом классе находится множество методов для выполнения различных действий. Рассмотрим некоторые из них.

Выше был показан пример кода, в котором выяснялось, указывают ли два пути на один файл. Сущесвует способ проверить это с помощью метода isSameFile () из класса Files:

**package** test;   
   
**import** java.io.IOException;   
**import** java.nio.file.\*;   
   
**class** Test4 {   
    **public** **static** **void** main(String[] args) **throws** IOException {   
        Path path1 = Paths.get("Test");   
        Path path2 = Paths.get("/home/heorhi/workspace/OCPJP/Test");   
   
        System.out.println("Files.isSameFile(path1, path2) is: "   
                + Files.isSameFile(path1, path2));   
    }   
}

Т.к. тут сравниваются файлы, а не пути, то существование соответствующего файла обязательно (см. подсказку для пути в предыдущем примере) иначе будет получена ошибка java.nio.file.NoSuchFileException.

В случае, если файл по указанному адресу существует, получим такой вывод:

Files.isSameFile(path1, path2) is: **true**

Можно определить, имеем мы дело с файлом или директорией (папкой) с помощью метода isDirectory() класса Files и проверить их существование с помощью метода exists():

**package** test;   
   
**import** java.nio.file.\*;   
   
**class** Test5 {   
    **public** **static** **void** main(String[] args) {   
        //Проверка для файла   
        Path path = Paths.get("/home/heorhi/workspace/OCPJP/src/test/Test5.java");   
           
        //Проверка для дериктории   
        //Path path = Paths.get("/home/heorhi/workspace/OCPJP/src/test");   
           
        **if** (Files.exists(path, LinkOption.NOFOLLOW\_LINKS)) {   
            System.out.println("The file/directory " + path.getFileName()   
                    + " exists");   
            **if** (Files.isDirectory(path, LinkOption.NOFOLLOW\_LINKS)) {   
                System.out.println(path.getFileName() + " is a directory");   
            } **else** {   
                System.out.println(path.getFileName() + " is a file");   
            }   
        } **else** {   
            System.out.println("The file/directory " + path.getFileName() + " does not exist");   
        }   
    }   
}

Вывод должен получится примерно такой (выбирайте пути в соответствии с вашей ОС и расположением файлов):

The file/directory Test5.java exists   
Test5.java is a file

Интересный вывод можно получить, если написать Path path = Paths.get("/"); для Unix-систем или Path path = Paths.get("С:\\"); для Windows-систем, т.е. если передать в качестве параметра имя корневого котолога:

The file/directory **null** exists   
**null** is a directory

Для корневого каталога path.getFileName() возвращает null.

Класс Files содержит методы **isReadable(**),**isWriteable()** и **isExecutable()** для проверки возможности чтения, записи и выполнения файлов:

**package** test;   
   
**import** java.nio.file.\*;   
   
**class** Test6 {   
    **public** **static** **void** main(String[] args) {   
        Path path = Paths.get("Вставьте сюда путь к какому-либо файлу");   
        System.out.printf("Readable: %b, Writable: %b, Executable: %b ",   
                Files.isReadable(path), Files.isWritable(path),   
                Files.isExecutable(path));   
    }   
}

Метод getAttribute() позволяет получить свойства (атрибуты) файла. Метод принимает переменное число параметров: первый — объект Path; второй — имя атрибута; далее от нуля до нескольких значений LinkOption (это enum):

**package** test;   
   
**import** java.io.IOException;   
**import** java.nio.file.\*;   
   
**class** Test7 {   
    **public** **static** **void** main(String[] args) {   
        Path path = Paths.get("Вставьте сюда путь к какому-либо файлу");   
        **try** {   
            Object object = Files.getAttribute(path, "creationTime");   
            System.out.println("Creation time: " + object);   
               
            //Здесь указан третий параметр   
            object = Files.getAttribute(path, "lastModifiedTime",   
                    LinkOption.NOFOLLOW\_LINKS);   
            System.out.println("Last modified time: " + object);   
   
            object = Files.getAttribute(path, "size");   
            System.out.println("Size: " + object);   
   
            object = Files.getAttribute(path, "isDirectory");   
            System.out.println("isDirectory: " + object);   
        } **catch** (IOException e) {   
            e.printStackTrace();   
        }   
    }   
}

Копирование файлов

Теперь рассмотрим коппирование файла/диретории. Для этого используем метод Files.copy(). Сигнатура данного метода:

Path copy(Path source, Path target, CopyOption. . . options)

, где первый параметр — путь к исходному файлу, второй — путь к тому файлу, что будет создан в результате копирования (включая имя нового файла), далее можно задать параметры копирования, а можно и не задать,

как в *примере*ниже:

**package** test;   
   
**import** java.io.IOException;   
**import** java.nio.file.\*;   
   
**public** **class** Test8 {   
    **public** **static** **void** main(String[] args) {   
        Path pathSource = Paths.get("Вставьте сюда путь к файлу/директории для копирования");   
        Path pathDestination = Paths.get("Вставьте сюда путь для нового файла/директории ");   
        **try** {   
            Files.copy(pathSource, pathDestination);   
            System.out.println("Source file copied successfully");   
        } **catch** (IOException e) {   
            e.printStackTrace();   
        }   
    }   
}

После первого запуска всё должно отработать корректно для корректных путей. Но если данный код без изменений скомпилировать и выполнить повторно, то будет получено исключение java.nio.file.FileAlreadyExistsException. Оно связано с тем, что целевой файл уже существует. Для избежания таких проблем можно указать, чтобы в случае его существования он перезаписывался. Для этого надо немного изменить одну строку кода:

Files.copy(pathSource, pathDestination, StandardCopyOption.REPLACE\_EXISTING);

**Следует отметить, что при копировании директории не будут копироваться содержащиеся в ней файлы и директории. Это выглядит глупо — ниже будет показано, как это исправить.**

Перемещение файла

Метод для перемещения файла очень похож на метод для копирования:

Path move(Path source, Path target, CopyOption. . . options)

Значения передаваемых параметров совпадают по смыслу. Пример кода отличается от предыдущего минимально:

**package** test;   
   
**import** java.io.IOException;   
**import** java.nio.file.Files;   
**import** java.nio.file.Path;   
**import** java.nio.file.Paths;   
**import** java.nio.file.StandardCopyOption;   
   
**public** **class** Test9 {   
    **public** **static** **void** main(String[] args) {   
        Path pathSource = Paths.get("Вставьте сюда путь к файлу/директории, который надо переместить");   
        Path pathDestination = Paths.get("Вставьте сюда путь для нового местоположения файла/директории");   
        **try** {   
            Files.move(pathSource, pathDestination, StandardCopyOption.REPLACE\_EXISTING);   
            System.out.println("Source file copied successfully");   
        } **catch** (IOException e) {   
            e.printStackTrace();   
        }   
    }   
}

Если при копировании директории содержащиеся в ней файлы и директории не копировались, то при её перемещении, в случае отсутствия ошибок, перемещается и всё содержимое.

Удаление файла

Посмотрим на пример кода:

**package** test;   
   
**import** java.io.IOException;   
**import** java.nio.file.\*;   
   
**public** **class** Test10 {   
    **public** **static** **void** main(String[] args) {   
        Path pathSource = Paths.get("Вставьте сюда путь к файлу/директории для удаления");   
        **try** {   
            Files.delete(pathSource);   
            System.out.println("File deleted successfully");   
        } **catch** (IOException e) {   
            e.printStackTrace();   
        }   
    }   
}

Несколько моментов, которые необходимо помнить относительно метода Files.delete():

* В случае удаления каталога, необходимо, чтобы он был пустым, иначе будет получено исключение (java.nio.file.DirectoryNotEmptyException)
* Если передать в данный метод символьную ссылку, то будет удалена ссылка, а не целевой файл.
* Для данного метода необходимо, чтобы файл существовал, иначе будет получено исключение (java.nio.file.NoSuchFileException). Если необходимо игнорировать данные случаи, то лучше подойдёт метод deleteIfExists(), который удаляет файл в случае его существования и не бросает подобное исключение при его отсутствии.

Обход дерева файлов

При работе с файловой системой может возникнуть необходимость обхода дерева файлов, например при поиске файла или копировании каталога со всем его содержимым. Класс Files содержит два метода, позволяющих обходить дерево файлов. Их сигнатуры приведены ниже:

Path walkFileTree(Path start, FileVisitor<!--? **super** Path--> visitor)   
Path walkFileTree(Path start, Set<filevisitoption> options, **int** maxDepth, FileVisitor<!--? **super** Path--> visitor)

Оба метода принимают путь, с которого начнётся обход дерева и экземпляр типа FileVisitor, который будет определять поведение при обходе дерева. Второй метод имеет два дополнительных параметра: Set, содержащий опции обхода, и максимальную глубину. Максимальная глубина определяет, насколько уровней каталогов будет происходить обход. Если в её качестве указать 0, то будет рассматриваться только указанный файл, а если указать MAX\_VALUE, то будут пройдены все подкаталоги.

**FileVisitor** — это интерфейс, содержащий следующие методы:

* FileVisitResult preVisitDirectory(T dir, BasicFileAttributesattrs) — выполняется перед доступом к элементам каталога.
* FileVisitResult visitFile(T file, BasicFileAttributes attrs) — выполняется при доступе к файлу.
* FileVisitResult postVisitDirectory(T dir, IOException exc) — выполняется, когда все элементы директории пройдены .
* FileVisitResult visitFileFailed(T file, IOException exc) — выполняется, если к файлу нет доступа.

Вам необходимо реализовать интерфейс FileVisitor, чтобы передать соответствующий объект в метод walkFileTree(). Но если необходимости реализовывать все четыре метода этого интерфейса нет, то можно просто расширить реализацию класса SimpleFileVisitor, переопределив лишь необходимые методы.

*Пример:*

**package** test;   
   
**import** java.io.IOException;   
**import** java.nio.file.\*;   
**import** java.nio.file.attribute.BasicFileAttributes;   
   
**class** MyFileVisitor **extends** SimpleFileVisitor {

**public** FileVisitResult **preVisitDirectory**(Path path, BasicFileAttributes fileAttributes) {  
        System.out.println("Directory name:" + path);  
        **return** FileVisitResult.CONTINUE;  
    }

**public** FileVisitResult **visitFile**(Path path, BasicFileAttributes fileAttributes) {  
        System.out.println("file name:" + path.getFileName());  
        **return** FileVisitResult.CONTINUE;  
    }  
}  
   
**public** **class** Test11 {   
    **public** **static** **void** main(String[] args) {   
   
        Path pathSource = Paths.get("Введите сюда путь к какому-либо каталогу, содержащему другие каталоги и файлы");   
        **try** {   
            Files.walkFileTree(pathSource, **new** MyFileVisitor());   
        } **catch** (IOException e) {   
            e.printStackTrace();   
        }   
    }   
}

При выполнении данного кода будут выведены все вложенные каталоги и файлы по указанному пути. Вот что следует понять:

* Объявляется класс MyFileVisitor, унаследованный от SimpleFileVisitor, в котором переопределены два метода: visitFile() (для вывода имени файла) и preVisitDirectory() (для вывода имени директории).
* Вызывается walkFileTree() в который передаётся объект MyFileVisitor.
* Метод walkFileTree() начинает выполнение с переданного в него каталога. При этом вызывается метод visitFile() при каждом проходе файла, preVisitDirectory() — перед просмотром элементов директории, postVisitDirectory() — после просмотра элементов директории, visitFileFailed() — в случае отсутствия доступа к файлу/дириктории.
* Из этих четырёх методов были переопределены только два для вывода имён каталогов и файлов.
* Можно контролировать поток обхода с помощью возвращаемых этими методами значений (enum FileVisitResult). Их четыре:
  1. **CONTINUE:** указывает на то, что обход дерева следует продолжить.
  2. **TERMINATE:** указывает, что обход нужно немедленно прекратить.
  3. **SKIP\_SUBTREE:** указывает, что подкаталоги должны быть пропущены для обхода.
  4. **SKIP\_SIBLINGS:** указывает на то, что обход должен быть остановлен в текущем каталоге и каталогах одного уровня с ним. Если это значение возвращается из preVisitDirectory(), то вложенные файлы/каталоги не обходятся и postVisitDirectory() не срабатывает. Если это значение возвращается из visitFile (), то остальные файлы каталога не обходятся. Если он возвращается из postVisitDirectory (), то остальные каталоги того же уровня не будут обходиться.

Доработка копирования папок с файлами

Возвращаясь к «глупому» копированию каталога в котором что-то есть — используя полученные знания можно реализовать его более логично, относительно результата, ожидаемого пользователем:

**package** test;   
   
**import** java.io.IOException;   
**import** java.nio.file.\*;   
**import** java.nio.file.attribute.\*;   
   
**class** MyFileCopyVisitor **extends** SimpleFileVisitor {   
    **private** Path source, destination;   
   
    **public** MyFileCopyVisitor(Path s, Path d) {   
        source = s;   
        destination = d;   
    }

**public** FileVisitResult **preVisitDirectory**(Path path, BasicFileAttributes fileAttributes) {

        Path newd = destination.resolve(source.relativize(path));

**try** {  
            Files.copy(path, newd, StandardCopyOption.REPLACE\_EXISTING);  
        } **catch** (IOException e) {  
            e.printStackTrace();  
        }

**return** FileVisitResult.CONTINUE;  
    }   
  
     **public** FileVisitResult **visitFile**(Path path, BasicFileAttributes fileAttributes) {

        Path newd = destination.resolve(source.relativize(path));

**try** {  
            Files.copy(path, newd, StandardCopyOption.REPLACE\_EXISTING);  
        } **catch** (IOException e) {  
            e.printStackTrace();  
        }

**return** FileVisitResult.CONTINUE;  
    }   
   
}   
   
**public** **class** Test12 {   
    **public** **static** **void** main(String[] args) {   
        Path pathSource = Paths.get("Вставьте сюда путь к желательно непустой директории для копирования");   
        Path pathDestination = Paths.get("Вставьте сюда путь к новому положению директории");   
        **try** {   
            Files.walkFileTree(pathSource, **new** MyFileCopyVisitor(pathSource,    pathDestination));   
            System.out.println("Files copied successfully!");   
        } **catch** (IOException e) {   
            e.printStackTrace();   
        }   
    }   
}

В методе preVisitDirectory() происходит копирование посещаемого каталога и аналогично копируется файл в методе visitFile(). Чтобы получить новый путь назначения, используется метод relativize() из класса Path.

Поиск файлов

Поняв принципы обхода дерева файлов, можно легко организовать поиск нужного файла. При поиске конкретного файла/каталога можно проверять соответствие имени файла/каталога с искомым с помощью метода visitFile () или preVisitDirectory (). Однако, если необходимо найти все файлы, соответствующие некоторому шаблону (например, все исходные файлы Java или XML-файлы ), то лучше использовать универсальный символ (glob) или регулярное выражение (regex). Тут пригодится интерфейс PathMatcher. Данный интерфейс реализован для каждой файловой системы и вы можете получить экземпляр этого типа из класса FileSystem используя метод getPathMatcher().

Перед тем, как перейти к примеру стоит пояснить шаблоны Glob (похожи на regex, но немного проще. Если понятие regex для Вас ново, то ближе с ним можно ознакомится здесь — [Регулярные выражения в Java](http://www.quizful.net/post/Java-RegExp)). В таблице ниже приведены шаблоны, поддерживаемые glob-синтаксисом:

| **Шаблон** | **Описание** |
| --- | --- |
| \* | Соответствует любой строке любой длины, даже пустой. |
| \*\* | Как и \*, но выходит за границы каталогов. |
| ? | Любой одиночный символ. |
| [XYZ] | Либо X, либо Y, либо Z. |
| [0-5] | Соответствует любому символу от 0 до 5. |
| [a–z] | Любой строчный символ латинского алфавита. |
| {XYZ, ABC} | Либо XYZ или ABC. |

Ниже приведён пример кода, который находит все java-файлы в указанном каталоге. Для поиска используется glob-шаблон, но в коментариях приведён regex-шаблон, который можно использовать для этой же цели. Обратите внимание, что в строке с шаблоном сначала указывается его тип (glob или regex), потом ставится доеточие, а потом пишется сам шаблон. Ради интереса можете запустить этот же код убрав первую часть с двоеточием, но сначала просто попробуйте скомпилировать и выполнить:

**package** test;   
**import** java.io.IOException;   
**import** java.nio.file.\*;   
**import** java.nio.file.attribute.\*;   
   
**class** MyFileFindVisitor **extends** SimpleFileVisitor<Path> {

**private** PathMatcher matcher;   
   
    **public** MyFileFindVisitor(String pattern) {

**try** {   
            matcher = FileSystems.getDefault().getPathMatcher(pattern);   
        } **catch** (IllegalArgumentException iae) {   
            System.err.println("Invalid pattern; did you forget to prefix \"glob:\" or \"regex:\"?");   
            System.exit(1);   
        }  
     }

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

        Path name = path.getFileName();

**if** (matcher.matches(name)) System.out.println("Matching file:" + path.getFileName());

    }

    **public** FileVisitResult **visitFile**(Path path, BasicFileAttributes fileAttributes) {

        find(path);

**return** FileVisitResult.CONTINUE;   
    }

**public** FileVisitResult **preVisitDirectory**(Path path, BasicFileAttributes fileAttributes) {

        find(path);

**return** FileVisitResult.CONTINUE;   
    }  
}

**public** **class** Test13 {

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

        Path startPath = Paths.get("Введите сюда путь к каталогу для поиска (желательно, чтобы он содержал java-файлы)");   
           
        //Строка с glob-шаблоном    
        String pattern = "glob:\*.java";   
           
        //Строка с regex-шаблоном    
        //String pattern = "regex:\\S+\\.java";   
           
        **try** {   
            Files.walkFileTree(startPath, **new** MyFileFindVisitor(pattern));   
            System.out.println("File search completed!");   
        } **catch** (IOException e) {   
            e.printStackTrace();   
        }   
    }   
}

Отслеживание изменений в каталоге

Предположим, что необходимо написать некое приложение, работающее с файлами (IDE, файловый менеджер или какой-либо редактор). Допустим какой-либо файл, важный для приложения был либо создан, либо изменён, либо удалён из вне и надо сообщить об этом пользователю.

Java 7 предоставляет сервис для слежения за каталогами. Вы можете зарегистрировать в нём каталог, чтобы получать уведомления о любом изменении в каталоге (создание, изменение и удаление файла).

Рассмотрим API этого сервиса на примере. Для этого примера используйте каталог в котором при выполнении программы в бесконечном цикле Вы будете создавать, изменять и удалять файлы — действия должны выводится на консоли. Код ниже:

**package** test;   
   
**import** java.io.IOException;   
**import** java.nio.file.\*;   
   
**public** **class** Test14 {   
    **public** **static** **void** main(String[] args) {   
        Path path = Paths.get("/Введите сюда путь к каталогу, изменения в котором будут отслеживаться");   
        WatchService watchService = **null**;   
        **try** {   
            watchService = path.getFileSystem().newWatchService();   
            path.register(watchService,   
                    StandardWatchEventKinds.ENTRY\_CREATE,   
                    StandardWatchEventKinds.ENTRY\_DELETE,   
                    StandardWatchEventKinds.ENTRY\_MODIFY);   
        } **catch** (IOException e1) {   
            e1.printStackTrace();   
        }   
        // Бесконечный цикл   
        **for** (;;) {   
            WatchKey key = **null**;   
            **try** {   
                key = watchService.take();   
            } **catch** (InterruptedException e) {   
                e.printStackTrace();   
            }   
            // Итерации для каждого события   
            **for** (WatchEvent event : key.pollEvents()) {   
                **switch** (event.kind().name()) {   
                **case** "OVERFLOW":   
                    System.out.println("We lost some events");   
                    **break**;   
                **case** "ENTRY\_CREATE":   
                    System.out.println("File " + event.context()   
                            + " is created!");   
                    **break**;   
                **case** "ENTRY\_MODIFY":   
                    System.out.println("File " + event.context()   
                            + " is modified!");   
                    **break**;   
                **case** "ENTRY\_DELETE":   
                    System.out.println("File " + event.context()   
                            + " is deleted!");   
                    **break**;   
                }   
            }   
            // Сброс ключа важен для получения последующих уведомлений   
            key.reset();   
} } }

*Пояснения к коду:*

* Получение экземпляра WatchService осуществляется через вызов метода newWatchService() из класса FileSystem, который в свою очередь был получен из объекта класса Path через вызов метода getFileSystem(). Но его (экземпляр FileSystem) можно получить и через вызов метода getDefault() из класса FileSystems, т.е. можно заменить watchService = path.getFileSystem().newWatchService();   
  на   
  watchService = FileSystems.getDefault().newWatchService();
* Далее происходит регистрация каталога на данном сервисе. Из объекта Path можно вызвать два метода register(), которые различаются принимаемыми параметрами. В нашем случае — это метод принимающий объект сервиса и переменное число параметров, определяющих отслеживаемые события.

path.register(watchService,   
StandardWatchEventKinds.ENTRY\_CREATE,   
StandardWatchEventKinds.ENTRY\_DELETE,   
StandardWatchEventKinds.ENTRY\_MODIFY);

* OVERFLOW указывает, что несколько уведомлений о событиях были отброшены или пропущены. ENTRY\_CREATE, ENTRY\_DELETE, ENTRY\_MODIFY соответствуют созданию, редактированию и удалению.
* В цикле происходит ожидание события. Здесь можно использовать три метода для уведомления:
* Метод poll() возвращает ключи очереди, если они есть, в противном случае сразу завершается.
* Метод poll(long, TimeUnit) возвращает ключи очереди, если они есть, в противном случае ждёт указанное число единиц времени.
* Метод take() возвращает ключ очереди, если он доступен, иначе ждёт, пока он не будет доступен.
* Получить вид события можно с помощью метода kind(), а имя файла, для которого произошло событие - с помощью метода context().

Основное различие между poll() и take() в том, что poll() это неблокирующий вызов, а take() — блокирующий.

Когда ключ возвращается, то одно или более событий могут быть помещены в очередь, именно поэтому используется ещё один цикл (для перебора всех событий).

Files.walk()

method signature

Review the Files.walk) method signature.

Files.java

public static Stream<Path> walk(Path start, FileVisitOption... options) throws IOException {

return walk(start, Integer.MAX\_VALUE, options);

}

public static Stream<Path> walk(Path start, int maxDepth, FileVisitOption... options) throws IOException {}

The path, starting file or folder.

The maxDepth, maximum number of directory levels to search, if omitted, default is Integer.MAX\_VALUE, which means 2,147,483,647 directory levels to search. In short, by default, it will search all levels of the directory. If we want to search from the top-level or root directory only, puts 1.

The FileVisitOption tells if we want to follow symbolic links, default is no. We can put FileVisitOption.FOLLOW\_LINKS to follow symbolic links.

1. List all files

This example uses Files.walk to list all files from a directory, including all levels of sub-directories (default).

FilesWalkExample1.java

package com.mkyong.io.api;

import java.io.IOException;

import java.nio.file.Files;

import java.nio.file.Path;

import java.nio.file.Paths;

import java.util.List;

import java.util.stream.Collectors;

import java.util.stream.Stream;

public class FilesWalkExample1 {

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

Path path = Paths.get("C:\\test\\");

List<Path> paths = listFiles(path);

paths.forEach(x -> System.out.println(x));

}

// list all files from this path

public static List<Path> listFiles(Path path) throws IOException {

List<Path> result;

try (Stream<Path> walk = Files.walk(path)) {

result = walk.filter(Files::isRegularFile)

.collect(Collectors.toList());

}

return result;

} }

Output

Terminal

C:\test\bk\logo-new.png

C:\test\bk\New Text Document.txt

C:\test\bk\readme.encrypted.txt

C:\test\bk\readme.txt

C:\test\data\ftp\google.png

C:\test\example.json

C:\test\google-decode.png

C:\test\google.png

If we want to list files from the root directory only, put maxDepth = 1.

*FilesWalkExample1.java*

try (Stream<Path> walk = Files.walk(path, 1)) {

result = walk.filter(Files::isRegularFile)

.collect(Collectors.toList());

}

Output

Terminal

C:\test\example.json

C:\test\google-decode.png

C:\test\google.png

2. List all folders or directories

// list all directories from this path

public static List<Path> listDirectories(Path path) throws IOException {

List<Path> result;

try (Stream<Path> walk = Files.walk(path)) {

result = walk.filter(Files::isDirectory)

.collect(Collectors.toList());

}

return result;

}

Output

Terminal

C:\test

C:\test\bk

C:\test\data

C:\test\data\ftp

C:\test\test1

C:\test\test1\test2

3. Find files by file extension

This example uses Files.walk to find all text files .txt from a path.

FilesWalkExample3.java

package com.mkyong.io.api;

import java.io.IOException;

import java.nio.file.Files;

import java.nio.file.Path;

import java.nio.file.Paths;

import java.util.List;

import java.util.stream.Collectors;

import java.util.stream.Stream;

public class FilesWalkExample3 {

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

Path path = Paths.get("C:\\test\\");

List<Path> paths = findByFileExtension(path, ".txt");

paths.forEach(x -> System.out.println(x));

}

public static List<Path> findByFileExtension(Path path, String fileExtension) throws IOException {

if (!Files.isDirectory(path)) { throw new IllegalArgumentException("Path must be a directory!"); }

List<Path> result;

try (Stream<Path> walk = Files.walk(path)) {

result = walk

.filter(Files::isRegularFile) // is a file

.filter(p -> p.getFileName().toString().endsWith(fileExtension)) // **endsWith()** это уже метод класса String

.collect(Collectors.toList());

}

return result;

} }

Output

Terminal

C:\test\bk\New Text Document.txt

C:\test\bk\readme.encrypted.txt

C:\test\bk\readme.txt

4. Find files by filename

public static List<Path> findByFileName(Path path, String fileName) throws IOException {

if (!Files.isDirectory(path)) {

throw new IllegalArgumentException("Path must be a directory!");

}

List<Path> result;

// walk file tree, no more recursive loop

try (Stream<Path> walk = Files.walk(path)) {

result = walk

.filter(Files::isReadable) // read permission

.filter(Files::isRegularFile) // is a file

.filter(p -> p.getFileName().toString().equalsIgnoreCase(fileName)) // **equalsIgnoreCase ()** это уже метод класса String

.collect(Collectors.toList());

}

return result;

}

5. Find files by file size

Find all files containing file size greater or equal to 10MB.

FilesWalkExample5.java

package com.mkyong.io.api;

import java.io.IOException;

import java.nio.file.Files;

import java.nio.file.Path;

import java.nio.file.Paths;

import java.util.List;

import java.util.stream.Collectors;

import java.util.stream.Stream;

// Files.walk example

public class FilesWalkExample5 {

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

Path path = Paths.get("C:\\test\\");

long fileSizeInBytes = 1024 \* 1024 \* 10; // 10MB

List<Path> paths = findByFileSize(path, fileSizeInBytes);

paths.forEach(x -> System.out.println(x));

}

// fileSize in bytes

public static List<Path> findByFileSize(Path path, long fileSize)

throws IOException {

if (!Files.isDirectory(path)) {

throw new IllegalArgumentException("Path must be a directory!");

}

List<Path> result;

// walk file tree, no more recursive loop

try (Stream<Path> walk = Files.walk(path)) {

result = walk

.filter(Files::isReadable) // read permission

.filter(p -> !Files.isDirectory(p)) // is a file

.filter(p -> checkFileSize(p, fileSize))

.collect(Collectors.toList());

}

return result;

}

private static boolean checkFileSize(Path path, long fileSize) {

boolean result = false;

try {

if (Files.size(path) >= fileSize) {

result = true;

}

} catch (IOException e) { System.err.println("Unable to get the file size of this file: " + path); }

return result;

} }

6. Update all files last modified date

This example update all files’ last modified date from a path to yesterday.

FilesWalkExample6.java

package com.mkyong.io.api;

import java.io.IOException;

import java.nio.file.Files;

import java.nio.file.Path;

import java.nio.file.Paths;

import java.nio.file.attribute.FileTime;

import java.time.Instant;

import java.time.temporal.ChronoUnit;

import java.util.stream.Stream;

public class FilesWalkExample6 {

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

Path path = Paths.get("C:\\test\\");

Instant yesterday = Instant.now().minus(1, ChronoUnit.DAYS);

setAllFilesModifiedDate(path, yesterday);

}

// set all files' last modified time to this instant

public static void setAllFilesModifiedDate(Path path, Instant instant) throws IOException {

try (Stream<Path> walk = Files.walk(path)) {

walk

.filter(Files::isReadable) // read permission

.filter(Files::isRegularFile) // file only

.forEach(p -> {

try {

// set last modified time

Files.setLastModifiedTime(p, FileTime.from(instant));

} catch (IOException e) { e.printStackTrace(); }

});

} } }

**P.S.** Не путайте File с Files, Path с Paths, и FileSystem c FileSystems. File старый класс (Java 4), а Files был введен в Java 7. Интерфейс Path представляет собой путь к файлу/каталогу и определяет полезный список методов, а класс Paths является служебным классом, который предоставляет два метода для получения объекта типа Path. Класс FileSystems предоставляет список фабричных методов для получения класса FileSystem, в то время как FileSystem предоставляет набор методов, для получения информации о файловой системе.