Professor Hans Noodles

41 уровень

10.05.2019 99263 97 + 32

Files, Path

Статья из группы Java Developer

42324 участника

Вы в группе

Привет!

Сегодня мы поговорим о работе с файлами и каталогами. Ты уже знаешь, как управлять содержимым файлов: у нас было немало занятий, посвященных этому :) Думаю, ты легко сможешь вспомнить несколько классов, которые нужны для этих целей.

На сегодняшней же лекции мы поговорим именно об управлении файлами — о создании, переименовании и т.д.

До появления Java 7 все подобные операции проводились с помощью класса `File`. О его работе ты можешь прочитать здесь.

Но в Java 7 создатели языка решили изменить работу с файлами и каталогами.

Это произошло из-за того, что у класса `File` был ряд недостатков. Например, в нем не было метода `copy()`, который позволил бы скопировать файл из одного места в другое (казалось бы, явно необходимая функция).

Кроме того, в классе `File` было достаточно много методов, которые возвращали `boolean`-значения. При ошибке такой метод возвращает *false*, а не выбрасывает исключение, что делает диагностику ошибок и установление их причин очень непростым делом.

НАЧАТЬ ОБУЧЕНИЕ

интерфейс, а не класс.

Давай разберемся, чем они друг от друга отличаются и зачем нужен каждый из них.

Начнем с самого легкого — `Paths`.

Paths

`Paths` — это совсем простой класс с единственным статическим методом `get()`. Его создали исключительно для того, чтобы из переданной строки или URI получить объект типа `Path`.

Другой функциональности у него нет.

Вот пример его работы:

```
1  import java.nio.file.Path;
2  import java.nio.file.Paths;
3
4  public class Main {
5
6     public static void main(String[] args) {
7
8         Path testFilePath = Paths.get("C:\\Users\\Username\\Desktop\\testFile.txt");
9     }
10 }
```

Не самый сложный класс, да? :)

Ну, раз уж мы получили объект типа `Path`, давай разбираться, что это за `Path` такой и зачем он нужен :)

Path

`Path`, по большому счету, — это переработанный аналог класса `File`. Работать с ним значительно проще, чем с `File`.

Во-первых, из него убрали многие утилитные (статические) методы, и перенесли их в класс `Files`.

Во-вторых, в `Path` были упорядочены возвращаемые значения методов. В классе `File` методы возвращали то `String`, то `boolean`, то `File` — разобраться было непросто.

Например, был метод `getParent()`, который возвращал родительский путь для текущего файла в виде строки. Но при этом был метод `getParentFile()`, который возвращал то же самое, но в виде объекта `File`!

Это явно избыточно. Поэтому в интерфейсе `Path` метод `getParent()` и другие методы работы с файлами возвращают просто объект `Path`. Никакой кучи вариантов — все легко и просто.

Какие же полезные методы есть у `Path`?

Вот некоторые из них и примеры их работы:

- `getFileName()` — возвращает имя файла из пути;
- `getParent()` — возвращает «родительскую» директорию по отношению к текущему пути (то есть ту директорию, которая находится выше по дереву каталогов);
- `getRoot()` — возвращает «корневую» директорию; то есть ту, которая находится на вершине дерева каталогов;

```java
import java.nio.file.Path;
import java.nio.file.Paths;

public class Main {

   public static void main(String[] args) {

       Path testFilePath = Paths.get("C:\\Users\\Username\\Desktop\\testFile.txt");

       Path fileName = testFilePath.getFileName();
       System.out.println(fileName);

       Path parent = testFilePath.getParent();
       System.out.println(parent);

       Path root = testFilePath.getRoot();
       System.out.println(root);

       boolean endWithTxt = testFilePath.endsWith("Desktop\\testFile.txt");
       System.out.println(endWithTxt);

       boolean startsWithLalala = testFilePath.startsWith("lalalala");
       System.out.println(startsWithLalala);
   }
}
```

Вывод в консоль:

testFile.txt
C:\Users\Username\Desktop
C:
true
false

Обрати внимание на то, как работает метод `endsWith()`. Он проверяет, заканчивается ли текущий путь на переданный **путь**. Именно на **путь**, а **не на набор символов**.

Сравни результаты этих двух вызовов:

```java
import java.nio.file.Path;
import java.nio.file.Paths;

public class Main {

   public static void main(String[] args) {

       Path testFilePath = Paths.get("C:\\Users\\Username\\Desktop\\testFile.txt");

       System.out.println(testFilePath.endsWith("estFile.txt"));
       System.out.println(testFilePath.endsWith("Desktop\\testFile.txt"));
   }
}
```

Вывод в консоль:

false
true

В метод `endsWith()` нужно передавать именно полноценный путь, а не просто набор символов: в противном случае результатом всегда будет *false*, даже если текущий путь действительно заканчивается такой последовательностью символов (как в случае с “estFile.txt” в примере выше).

Кроме того, **в `Path` есть группа методов, которая упрощает работу с абсолютными (полными) и относительными путями**.

Давай рассмотрим эти методы:

- `boolean isAbsolute()` — возвращает *true*, если текущий путь является абсолютным:

```
import java.nio.file.Path;
import java.nio.file.Paths;

public class Main {

   public static void main(String[] args) {

       Path testFilePath = Paths.get("C:\\Users\\Username\\Desktop\\testFile.txt");

       System.out.println(testFilePath.isAbsolute());
   }
}
```

Вывод в консоль:

true

- `Path normalize()` — «нормализует» текущий путь, удаляя из него ненужные элементы. Ты, возможно, знаешь, что в популярных операционных системах при обозначении путей часто используются символы “.” (“текущая директория”) и “..” (родительская директория). Например: “***./Pictures/dog.jpg***” обозначает, что в той директории, в которой мы сейчас находимся, есть папка Pictures, а в ней — файл “dog.jpg”

  Так вот. Если в твоей программе появился путь, использующий “.” или “..”, метод `normalize()` позволит удалить их и получить путь, в котором они не будут содержаться:

```
import java.nio.file.Path;
import java.nio.file.Paths;

public class Main {

   public static void main(String[] args) {


       Path path5 = Paths.get("C:\\Users\\Java\\.\\examples");

       System.out.println(path5.normalize());

       Path path6 = Paths.get("C:\\Users\\Java\\..\\examples");
       System.out.println(path6.normalize());
   }
}
```

Вывод в консоль:

C:\Users\Java\examples
C:\Users\examples

- `Path relativize()` — вычисляет относительный путь между текущим и переданным путем.

```java
import java.nio.file.Path;
import java.nio.file.Paths;

public class Main {

   public static void main(String[] args) {

       Path testFilePath1 = Paths.get("C:\\Users\\Users\\Users\\Users");
       Path testFilePath2 = Paths.get("C:\\Users\\Users\\Users\\Users\\Username\\Desktop\\testFil

       System.out.println(testFilePath1.relativize(testFilePath2));
   }
}
```

Вывод в консоль:

Username\Desktop\testFile.txt

Полный список методов `Path` довольно велик. Найти их все ты сможешь в документации Oracle.

Мы же перейдем к рассмотрению **`Files`**.

Files

`Files` — это утилитный класс, куда были вынесены статические методы из класса `File`. `Files` — это примерно то же, что и `Arrays` или `Collections`, только работает он с файлами, а не с массивами и коллекциями :)

Он сосредоточен на управлении файлами и директориями. Используя статические методы `Files`, мы можем создавать, удалять и перемещать файлы и директории.

Для этих операций используются методы **`createFile()`** (для директорий — **`createDirectory()`**), **`move()`** и **`delete()`**.

Вот как ими пользоваться:

```java
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

import static java.nio.file.StandardCopyOption.REPLACE_EXISTING;

public class Main {

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

       //создание файла
       Path testFile1 = Files.createFile(Paths.get("C:\\Users\\Username\\Desktop\\testFile111.txt"));
       System.out.println("Был ли файл успешно создан?");
       System.out.println(Files.exists(Paths.get("C:\\Users\\Username\\Desktop\\testFile111.txt")));

       //создание директории
       Path testDirectory = Files.createDirectory(Paths.get("C:\\Users\\Username\\Desktop\\testDirect
       System.out.println("Была ли директория успешно создана?");
       System.out.println(Files.exists(Paths.get("C:\\Users\\Username\\Desktop\\testDirectory")));
```

```
        testFile1 = Files.move(testFile1, Paths.get("C:\\Users\\Username\\Desktop\\testDirectory\\test
        System.out.println("Остался ли наш файл на рабочем столе?");
        System.out.println(Files.exists(Paths.get("C:\\Users\\Username\\Desktop\\testFile111.txt")));
        System.out.println("Был ли наш файл перемещен в testDirectory?");
        System.out.println(Files.exists(Paths.get("C:\\Users\\Username\\Desktop\\testDirectory\\testFi
        //удаление файла
        Files.delete(testFile1);
        System.out.println("Файл все еще существует?");
        System.out.println(Files.exists(Paths.get("C:\\Users\\Username\\Desktop\\testDirectory\\testFi
    }
}
```

Здесь мы сначала создаем файл (метод `Files.createFile()`) на рабочем столе, далее создаем там же папку (метод `Files.createDirectory()`). После этого мы перемещаем файл (метод `Files.move()`) с рабочего стола в эту новую папку, а в конце — удаляем файл (метод `Files.delete()`).

Вывод в консоль:

Был ли файл успешно создан?
true
Была ли директория успешно создана?
true
Остался ли наш файл на рабочем столе?
false
Был ли наш файл перемещен в testDirectory?
true
Файл все еще существует?
false

`Обрати внимание:` так же, как и методы интерфейса `Path`, **многие методы `Files` возвращают объект `Path`**.

Большинство методов класса `Files` принимают на вход также объекты `Path`. Тут твоим верным помощником станет метод `Paths.get()` — активно им пользуйся.

Что еще интересного есть в `Files`? То, чего очень не хватало старому классу `File` — метод `copy()`! Мы говорили о нем в начале лекции, самое время с ним познакомиться!

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

import static java.nio.file.StandardCopyOption.REPLACE_EXISTING;

public class Main {

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

        //создание файла
        Path testFile1 = Files.createFile(Paths.get("C:\\Users\\Username\\Desktop\\testFile111.txt"));
```

```
16
17          //создание директории
18          Path testDirectory2 = Files.createDirectory(Paths.get("C:\\Users\\Username\\Desktop\\testDirec
19          System.out.println("Была ли директория успешно создана?");
20          System.out.println(Files.exists(Paths.get("C:\\Users\\Username\\Desktop\\testDirectory2")));
21
22          //копируем файл с рабочего стола в директорию testDirectory2.
23          testFile1 = Files.copy(testFile1, Paths.get("C:\\Users\\Username\\Desktop\\testDirectory2\\tes
24
25          System.out.println("Остался ли наш файл на рабочем столе?");
26          System.out.println(Files.exists(Paths.get("C:\\Users\\Username\\Desktop\\testFile111.txt")));
27
28          System.out.println("Был ли наш файл скопирован в testDirectory?");
29          System.out.println(Files.exists(Paths.get("C:\\Users\\Username\\Desktop\\testDirectory2\\testF
30      }
31  }
```

Вывод в консоль:

Был ли файл успешно создан?

true

Была ли директория успешно создана?

true

Остался ли наш файл на рабочем столе?

true

Был ли наш файл скопирован в testDirectory?

true

Теперь ты умеешь копировать файлы программно! :)

Но класс `Files` позволяет не только управлять самими файлами, но и работать с его содержимым.

Для записи данных в файл у него есть метод **`write()`**, а для чтения — целых 3: **`read()`**, **`readAllBytes()`** и **`readAllLines()`**

Мы подробно остановимся на последнем. Почему именно на нем?

Потому что у него есть очень интересный тип возвращаемого значения — **`List<String>`**! То есть он возвращает нам список строк файла. Конечно, это делает работу с содержимым очень удобной, ведь весь файл, строку за строкой, можно, например, вывести в консоль в обычном цикле `for`:

```
1   import java.io.IOException;
2   import java.nio.file.Files;
3   import java.nio.file.Paths;
4   import java.util.List;
5
6   import static java.nio.charset.StandardCharsets.UTF_8;
7
8   public class Main {
9
10     public static void main(String[] args) throws IOException {
11
12         List<String> lines = Files.readAllLines(Paths.get("C:\\Users\\Username\\Desktop\\pushkin.txt")
```

НАЧАТЬ ОБУЧЕНИЕ

```
            System.out.println(s);
        }
    }
}
```

Вывод в консоль:

Я помню чудное мгновенье:
Передо мной явилась ты,
Как мимолетное виденье,
Как гений чистой красоты.

Очень удобно! :)

Такая возможность появилась еще в Java 7.

В версии Java 8 появился **Stream API**, который добавил в Java некоторые элементы функционального программирования. В том числе более богатые возможности работы с файлами.

Представь, что у нас есть задача: найти в файле все строки, которые начинаются со слова «Как», привести их к UPPER CASE и вывести в консоль.

Как выглядело бы решение с использованием класса `Files` в Java 7?

Примерно вот так:

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.List;

import static java.nio.charset.StandardCharsets.UTF_8;

public class Main {

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

        List<String> lines = Files.readAllLines(Paths.get("C:\\Users\\Username\\Desktop\\pushkin.txt")

        List<String> result = new ArrayList<>();

        for (String s: lines) {
            if (s.startsWith("Как")) {
                String upper = s.toUpperCase();
                result.add(upper);
            }
        }

        for (String s: result) {
            System.out.println(s);
        }
    }
```

Вывод в консоль:

КАК МИМОЛЕТНОЕ ВИДЕНЬЕ,
КАК ГЕНИЙ ЧИСТОЙ КРАСОТЫ.

Мы вроде справились, но не кажется ли тебе, что для такой простой задачи наш код получился немного...многословным?

С использованием Java 8 Stream API решение выглядит намного более элегантным:

```
1  import java.io.IOException;
2  import java.nio.file.Files;
3  import java.nio.file.Paths;
4  import java.util.List;
5  import java.util.stream.Collectors;
6  import java.util.stream.Stream;
7
8  public class Main {
9
10     public static void main(String[] args) throws IOException {
11
12         Stream<String> stream = Files.lines(Paths.get("C:\\Users\\Username\\Desktop\\pushkin.txt"));
13
14         List<String> result  = stream
15                 .filter(line -> line.startsWith("Как"))
16                 .map(String::toUpperCase)
17                 .collect(Collectors.toList());
18         result.forEach(System.out::println);
19     }
20 }
```

Мы добились того же результата, но с гораздо меньшим объемом кода! Причем нельзя сказать, что мы потеряли в «читабельности». Думаю, ты легко сможешь прокомментировать что делает этот код, даже не будучи знакомым со Stream API.

Но если вкратце, **Stream** — это последовательность элементов, над которыми можно выполнять разные функции. Мы получаем объект Stream из метода `Files.lines()`, после чего применяем к нему 3 функции:

1. С помощью метода `filter()` отбираем только те строки из файла, которые начинаются с «Как».
2. Проходимся по всем отобранным строкам с помощью метода `map()` и приводим каждую из них к UPPER CASE.
3. Объединяем все получившиеся строки в `List` с помощью метода `collect()`.

На выходе мы получаем тот же результат:

КАК МИМОЛЕТНОЕ ВИДЕНЬЕ,
КАК ГЕНИЙ ЧИСТОЙ КРАСОТЫ.

Если тебе будет интересно узнать больше о возможностях этой библиотеки, рекомендуем прочесть вот эту статью.

Мы же вернемся к нашим баранам, то есть файлам :)

Последняя возможность, которую мы сегодня рассмотрим — это **проход по дереву файлов**.

Файловая структура в современных операционных системах чаще всего имеет вид дерева: у него есть корень и есть ветки, от

Роль корня и веток выполняют директории.

Например, в роли корня может выступать директория “***C://***”.

От него отходят две ветки: “***C://Downloads***” и “***C://Users***”.

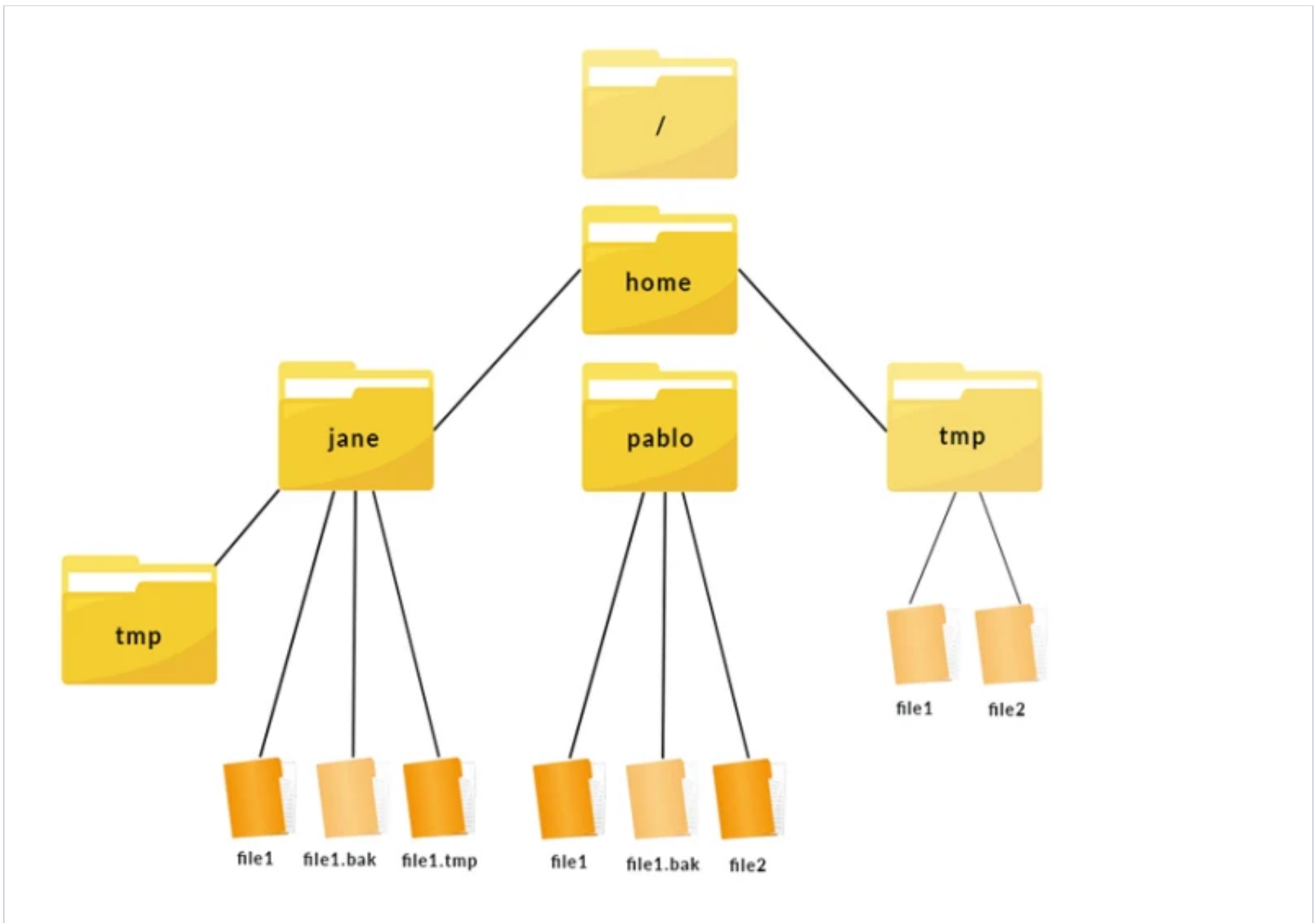
От каждой из этих веток отходят еще по 2 ветки:

“***C://Downloads/Pictures***”, “***C://Downloads/Video***”,

“***C://Users/JohnSmith***”, “***C://Users/Pudge2005***”.

От этих веток отходят другие ветки и т.д. — так и получается дерево.

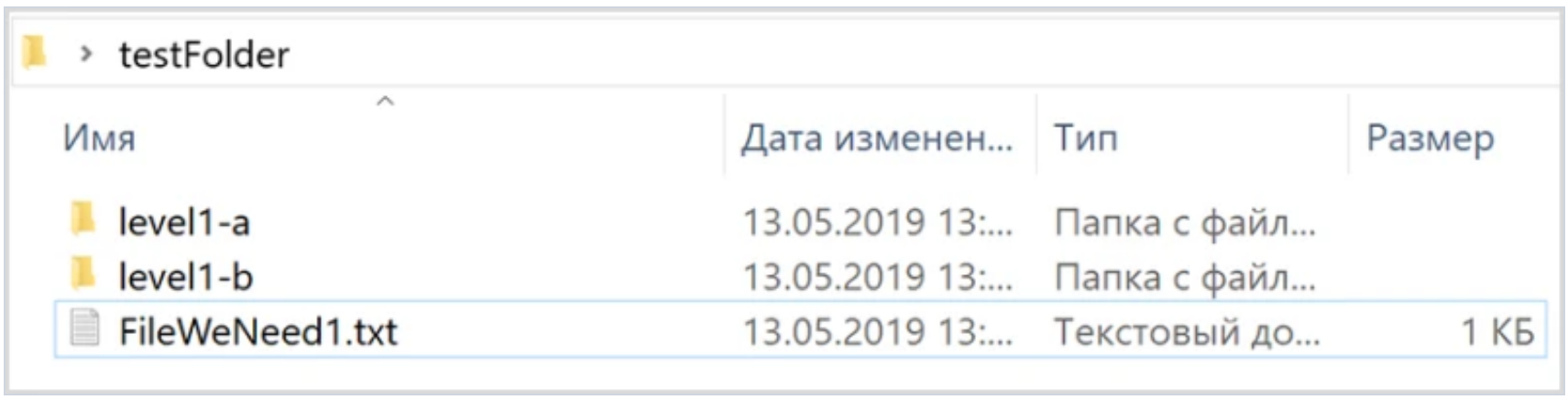
В Linux это выглядит примерно так же, только там в роли корня выступает директория */*



Теперь представь, что у нас есть задача: зная корневой каталог, мы должны пройтись по нему, заглянуть в папки всех уровней и найти в них файлы с нужным нам содержимым. Мы будем искать файлы, содержащие строку «This is the file we need!»

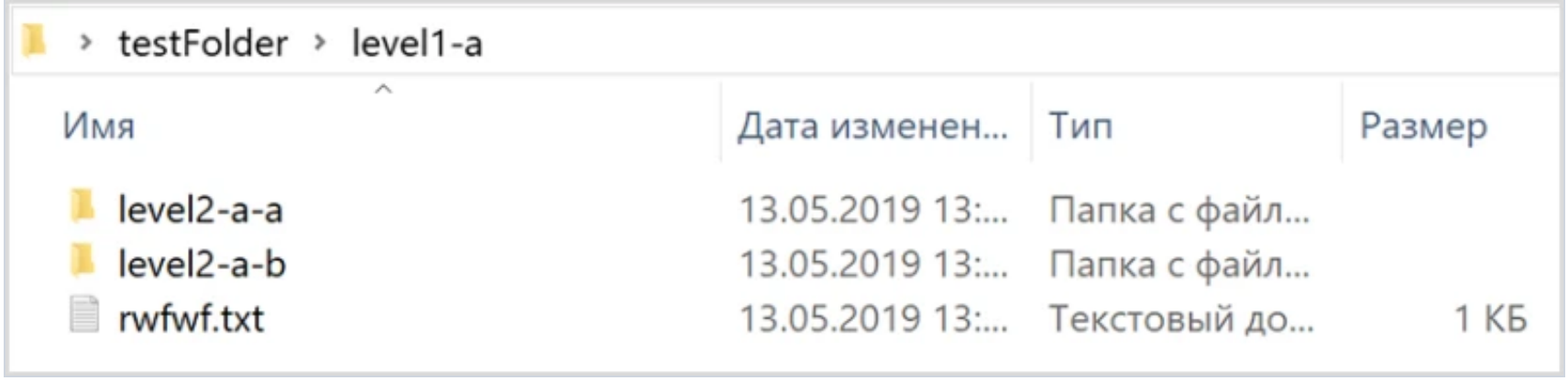
Нашим корневым каталогом будет папка «testFolder», которая лежит на рабочем столе.

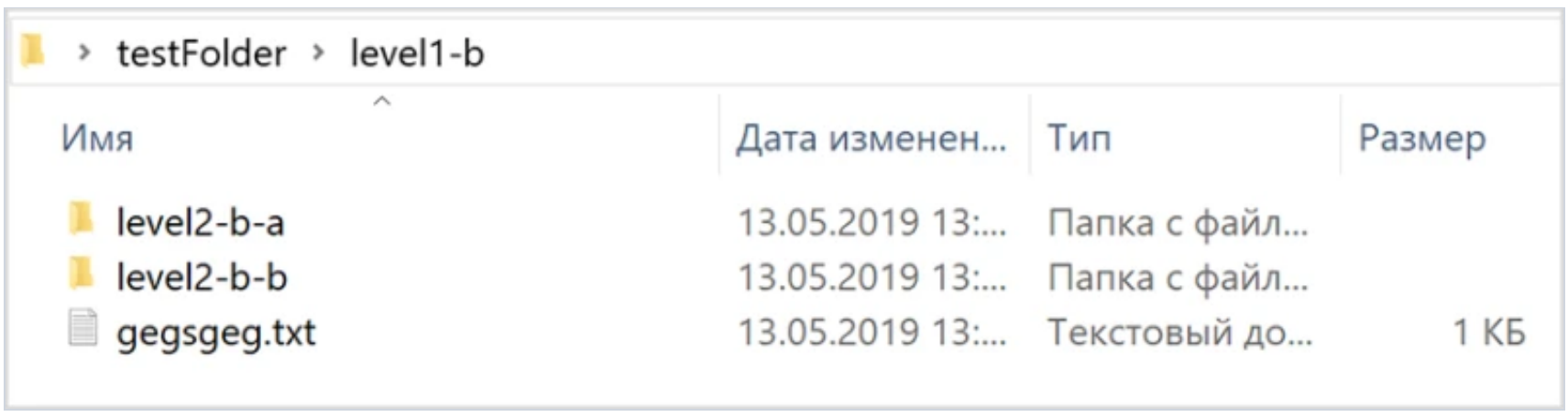
Внутри у нее вот такое содержимое:



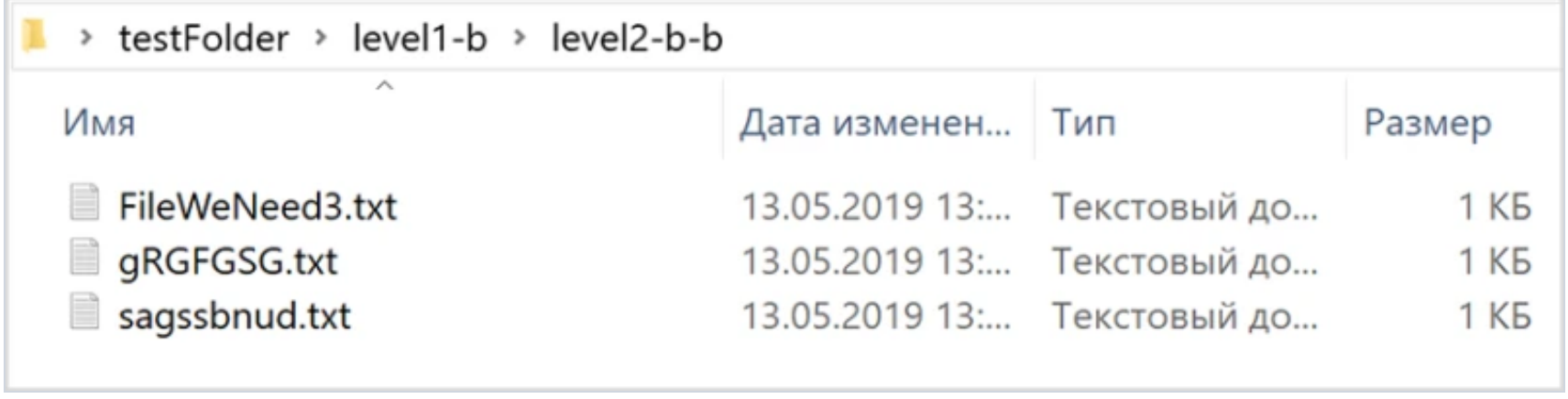
Внутри папок level1-a и level1-b тоже есть папки:

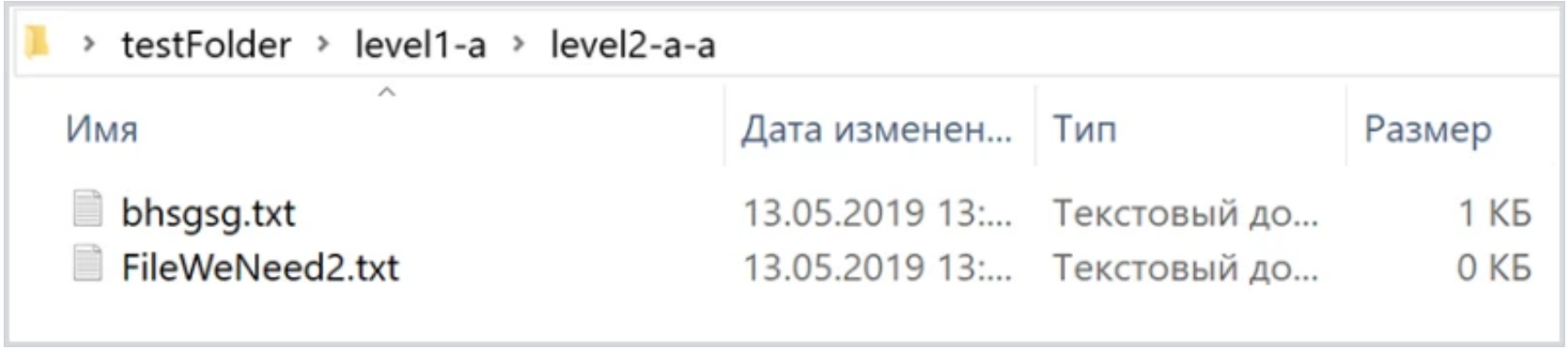
НАЧАТЬ ОБУЧЕНИЕ





Внутри этих «папок второго уровня» папок уже нет, только отдельные файлы:





3 файла с нужным нам содержимым мы специально обозначим понятными названиями — FileWeNeed1.txt, FileWeNeed2.txt, FileWeNeed3.txt

Именно их нам и нужно найти по содержимому с помощью Java.
Как же нам это сделать?

На помощь приходит очень мощный метод для обхода дерева файлов — **`Files.walkFileTree()`**.

Вот что нам нужно сделать.

Во-первых, нам понадобится **`FileVisitor`**. `FileVisitor` — это специальный интерфейс, в котором описаны все методы для обхода дерева файлов.

В частности, мы поместим туда логику считывания содержимого файла и проверки, содержит ли он нужный нам текст.

Вот как будет выглядеть наш `FileVisitor`:

```
1  import java.io.IOException;
2  import java.nio.file.FileVisitResult;
```

```
import java.nio.file.SimpleFileVisitor;
import java.nio.file.attribute.BasicFileAttributes;
import java.util.List;

public class MyFileVisitor extends SimpleFileVisitor<Path> {

    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException {

        List<String> lines = Files.readAllLines(file);
        for (String s: lines) {
            if (s.contains("This is the file we need")) {
                System.out.println("Нужный файл обнаружен!");
                System.out.println(file.toAbsolutePath());
                break;
            }
        }

        return FileVisitResult.CONTINUE;
    }
}
```

В данном случае наш класс наследуется от **`SimpleFileVisitor`**. Это класс, реализующий `FileVisitor`, в котором нужно переопределить всего один метод: **`visitFile()`**. Здесь мы и описываем что нужно делать с каждым файлом в каждой директории.

Если тебе нужна более сложная логика обхода, стоит написать свою реализацию `FileVisitor`. Там понадобится реализовать еще 3 метода:

- **`preVisitDirectory()`** — логика, которую надо выполнять перед входом в папку;
- **`visitFileFailed()`** — что делать, если вход в файл невозможен (нет доступа, или другие причины);
- **`postVisitDirectory()`** — логика, которую надо выполнять после захода в папку.

У нас такой логики нет, поэтому нам достаточно `SimpleFileVisitor`.

Логика внутри метода `visitFile()` довольно проста: прочитать все строки из файла, проверить, есть ли в них нужное нам содержимое, и если есть — вывести абсолютный путь в консоль.

Единственная строка, которая может вызвать у тебя затруднение — вот эта:

```
return FileVisitResult.CONTINUE;
```

На деле все просто. Здесь мы просто описываем что должна делать программа после того, как выполнен вход в файл, и все необходимые операции совершены. В нашем случае необходимо продолжать обход дерева, поэтому мы выбираем вариант **`CONTINUE`**.

Но у нас, например, могла быть и другая задача: найти не все файлы, которые содержат «This is the file we need», а **только один такой файл**. После этого работу программы нужно завершить. В этом случае наш код выглядел бы точно так же, но вместо break; было бы:

```
return FileVisitResult.TERMINATE;
```

```java
import java.io.IOException;
import java.nio.file.*;

public class Main {

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

       Files.walkFileTree(Paths.get("C:\\Users\\Username\\Desktop\\testFolder"), new MyFileVisitor());
   }
}
```

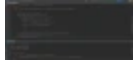
Вывод в консоль:

Нужный файл обнаружен!
C:\Users\Username\Desktop\testFolder\FileWeNeed1.txt
Нужный файл обнаружен!
C:\Users\Username\Desktop\testFolder\level1-a\level2-a-a\FileWeNeed2.txt
Нужный файл обнаружен!
C:\Users\Username\Desktop\testFolder\level1-b\level2-b-b\FileWeNeed3.txt

Отлично, у нас все получилось! :)

Если тебе хочется узнать больше о `walkFileTree()`, рекомендую тебе вот эту статью. Также ты можешь выполнить небольшое задание — заменить `SimpleFileVisitor` на обычный `FileVisitor`, реализовать все 4 метода и придумать предназначение для этой программы. Например, можно написать программу, которая будет логировать все свои действия: выводить в консоль название файла или папки до/после входа в них.

На этом все — до встречи! :)



Ответить

0

Anonymous #3084075 Уровень 7, Azerbaijan

8 июня, 20:49

Топовая лекция

Ответить

+1

Kotamadeo Уровень 32

4 июня, 19:11

Лучшая лекция по полезности материала!

Ответить

+1

Andrey Уровень 41, Санкт-Петербург, Россия

11 мая, 11:27

Одна из полезнейших лекций на JR

Ответить

+4

Ольга Уровень 23, Н.Новгород, Russian Federation

3 мая, 20:25

статья не открывается (ссылка), где про Java 8 Stream API

Ответить

+1

brd Уровень 29, Москва , Россия

1 июня, 14:30

открывает

Ответить

0

Q1R27 Уровень 17, Ukraine

29 апреля, 19:58

Чтение текстового файла в строковый поток, строка за строкой
В Java 8 появились потоки. Соответственно, в той же версии Java класс Files был расширен методом lines(), который возвращает строки текстового файла не в виде списка String, а в виде потока Strings:

```
String fileName = ...;
Stream<String> lines = Files.lines(Path.of(fileName));
```

Code language: GLSL (glsl)
Например, с помощью только одной инструкции кода можно вывести все строки текстового файла, содержащие строку "foo":

```
Files.lines(Path.of(fileName))
     .filter(line -> line.contains("foo"))
     .forEach(System.out::println);
```

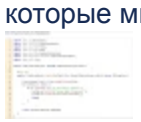
Ответить

+2

John F Уровень 25

19 апреля, 16:51

Здравствуйте, пожалуйста помогите понять. В 18 строке мы выводим в консоль абсолютный путь изначальной директории, которая нам дана. Но в итоге в консоль выводятся 3 абсолютные пути файлов, которые мы ищем. Как так получается?



Ответить

0

Ada Уровень 35

7 мая, 11:56

Само условие задачи такое: зная корневой каталог, мы должны пройтись по нему, заглянуть в папки всех уровней и найти в них файлы с нужным нам содержимым.

В этом коде мы обходим дерево файлов: заходим в файл, проверяем строки, которые в нем записаны, если находим нужную — печатаем её абсолютный путь, а затем, с помощью строчки return FileVisitResult.CONTINUE; мы говорим нашей программе, что как только она закончит в се свои манипуляции с файлом, ей нужно продолжать обход дерева

Ответить

+1

John F Уровень 25

8 мая, 13:32

Я понял, спасибо.

Ответить

0

Алексей Уровень 24, Витебск, Belarus

13 апреля, 15:02

Подскажите чем отличаются Path.of() и Paths.get() ?

НАЧАТЬ ОБУЧЕНИЕ

Я конечно глуповат в этом еще, но как я понял, чтобы создать объект класса path, нам нужно прописать что-то вроде этого: Path filePath = Path.of(scanner.nextLine());, и только потом уже использовать где-то, а Paths.get простой класс с одним методом, чтобы сразу создавать объект класса Path и можно сразу пользоваться им, как например здесь: Stream<String> stream = Files.lines(Paths.get("C:\\Users\\Username\\Desktop\\pushkin.txt")); кароч тупо код сокращаем на 1 строчку, где это можно сделать

Ответить 0

Q1R27 Уровень 17, Ukraine 29 апреля, 20:05

вообще Path это интерфейс,а объекты интерфейса создавать нельзя,видимо поэтому и был создан класс Paths с единственным методом Paths.get() который создаёт объект типа Path
но в оракле написано лучше используйте Path.of() вместо Paths.get() т.к. в дальнейшем возможно его уберут (устареет)

Ответить 0

Pavel Titov Уровень 1, Ukraine 11 апреля, 02:17

Подскажите пожалуйста, почему Path normalize() во втором примере с двумя точками в консоли удалил папку Java?

Ответить +1

milyasow Уровень 30, Москва, Russian Federation 13 апреля, 00:05

Две точки обозначают переход на уровень выше по файловой системе, в "C:\Users\". А дальше - вход в каталог examples. В итоге получается полный путь:
C:\Users\examples

Ответить 0

Серега Батенин Уровень 16, Москва, Россия 14 мая, 12:24

честно говоря из представленных методов тоже не совсем понял как работает normalize. Просто как я понял путь до какого то файла может быть очень длинным и к примеру его центральную часть заменяют на ".."
То есть нам показывают его начало C:\Users\User\ и так далее ".." и доходим там через возможно еще десятки разных папок до какой то конечной \project\example\test.txt
то есть эти точки просто "скрывают" ту простыню переходов по директориям. И если я вызываю метод normalize то мой путь каким то образом вообще срезается в разы и говорит что якобы у меня в папке User лежит папка project которой там по факту может не существовать. Или я чтото совсем не так понял((

Ответить 0

milyasow Уровень 30, Москва, Russian Federation 14 мая, 23:07

Читайте внимательнее.
Две точки обозначают переход на уровень выше по файловой системе.

Ответить 0

Серега Батенин Уровень 16, Москва, Россия 15 мая, 16:54

я все равно хоть убей не могу понять нафиг такое нужно? По сути у нас на выходе получается невалидный путь.. Если взять код из примера то папка "Example" лежит внутри папки "Java", а в итоге мы получаем путь где у нас эта папка с Примерами лежит в "User" и если вдруг кто то попробует в коде обратиться к этому пути то у нас вылетит исключение

Ответить 0

milyasow Уровень 30, Москва, Russian Federation 15 мая, 17:16

Очень нужно, например, в командной строке Linux, где одной командой можно выйти из одного каталога корневой папки и войти в другой.
В коде с примером идет изменение пути к каталогу, а не обращение к нему. Понятно, что обращаться к несуществующему каталогу не имеет смысла, но если просто предположить, что в каталоге "User" все же существует папка "Example", то все встанет на свои места.

Ответить 0

Показать еще комментарии

RUSH

JavaRush — это интерактивный онлайн-курс по изучению Java-программирования с нуля. Он содержит 1200 практических задач с проверкой решения в один клик, необходимый минимум теории по основам Java и мотивирующие фишки, которые помогут пройти курс до конца: игры, опросы, интересные проекты и статьи об эффективном обучении и карьере Java-девелопера.

ПОДПИСЫВАЙТЕСЬ

ЯЗЫК ИНТЕРФЕЙСА

Русский