



HIGHER SCHOOL OF ECONOMICS
NATIONAL RESEARCH UNIVERSITY

Лекция 16

Работа с файловой системой

Программирование на языке Java

Роман Гуров

ВШЭ БИ 2021





Файловая система

Фа́йловая систе́ма (англ. *file system*) — порядок, определяющий способ организации, хранения и именования данных на носителях информации в виде файлов

Любые носители информации, типа жёстких дисков или USB-флешек, по сути представляют собой набор пронумерованных ячеек памяти (то есть, *одномерный* массив байтов)

Файловая система – это способ организовать такой кусок памяти в приятный и понятный для нас формат – в виде иерархии файлов и директорий, которые можно создавать, удалять и перемещать

Директории также называют папками или каталогами

Часто директории считают особым типом файла, для однородности



Абсолютные пути

Основание файловой системы (когда не вошли ни в одну директорию) называется её корнем или корневым каталогом

У каждого файла в файловой системе есть некоторый путь, по которому он находится

При этом на разных ОС пути выглядят по-разному:

- Windows: `C:\Users\romangurov\Videos\lecture-16.mkv`
- Linux: `/home/romangurov/Videos/lecture-16.mkv`

Заметим, что разделитель отличается

Такие полные пути от самого корня называются абсолютными путями

Абсолютный путь самого корня выглядит так:

- Windows: `C:\`
- Linux: `/`



Относительные пути

Писать каждый раз полный путь до файла от корня неудобно и не практично

Например, у нас есть папка с игрой. В папке с игрой, помимо самого исполняемого файла игры, хранятся ресурсы: звуки, картинки со спрайтами 2D-персонажей

Если бы игра искала эти ресурсы по абсолютному пути, то при переносе директории с игрой в другое место, ресурсы бы просто больше не нашлись

Путь, который начинается не с корня, считается относительным:

- Windows: `resources\sprites\char01.png`
- Linux: `resources/sprites/char01.png`



Относительные пути

Путь, который начинается не с корня, считается относительным:

- Windows: resources\sprites\char01.png
- Linux: resources/sprites/char01.png

А относительно чего?

У каждой программы есть *текущая рабочая директория*

Относительно неё и считается относительный путь

Программа способна в процессе работы изменять свою рабочую директорию,
но изначальное значение передаётся ей при запуске

Обычно это директория, в которой лежал сам исполняемый файл, но подсунуть можно что угодно

Относительные пути

В путях можно использовать особые элементы: `.` и `..`

Можно представлять, что в любой директории есть две фиктивные директории с такими именами

- `"."` – при переходе в неё попадаем в ту же самую директорию, где были
- `".."` – при переходе в неё попадаем в директорию выше в иерархии (выходим из директории наружу)

Пусть есть такая иерархия: ->

И текущая рабочая директория:
`C:\dir1\`



Зная, что по соседству с нами должна лежать `dir2`, к ней можно обратиться относительным путём:

`..\dir2\a.txt`

Так же можно и выразить явно файл из текущей директории:

`.\a.txt`

А можно и такое:

`../dir2/./../dir1/./b.txt`

`C:\.\.\dir2\..\a.txt`

Это полезно при склейке нескольких путей



Класс java.io.File

Класс File задаёт путь к некоторому файлу (или директории):

```
package java.io;

public class File implements Serializable, Comparable<File> {
    public File(String pathname)
        // ...
}
```

Путь задаётся в виде, зависящем от ОС, на которой выполняется программа:

```
// Windows
File javaExecutable = new File("C:\\jdk1.8.0_60\\bin\\java.exe");

// Linux
File bashExecutable = new File("/bin/bash");
```

Опять видим, что разделитель отличается

Обратный слэш приходится удваивать, так как он уже занят для спецсимволов типа "\n"

Хоть класс и называется File, он задаёт лишь путь; указанному файлу не обязательно даже существовать

Можно передавать как абсолютные, так и относительные пути

Разделители

Чтобы самостоятельно склеить путь из двух частей, нужно знать, какой разделитель используется на текущей платформе

Для этого у `File` есть статические константы `separator` и `separatorChar`:

```
public static final String separator;  
public static final char pathSeparatorChar;
```

Отличаются только типом, внутри одно и то же

```
String sourceDirName = "src";  
String mainFileName = "Main.java";  
  
String mainFilePath = sourceDirName + File.separator + mainFileName;  
// Результат: src\Main.java
```

Но лучше в таком случае позволить классу `File` склеить пути самостоятельно, при помощи его двухаргументных конструкторов:

```
public File(String parent, String child)  
  
public File(File parent, String child)
```

```
File mainFile = new File(sourceDirName, mainFileName);  
File weirdFile = new File(mainFile, "some_child");
```

Ничто не мешает `Main.java` оказаться директорией ☺



Разделители

Также, существуют похожие константы для разделения путей

```
public static final char pathSeparatorChar;  
public static final String pathSeparator;
```

Много занятий назад, мы пробовали собирать проект вместе с некоторой библиотекой, которую скачали в виде .jar файла

Путь к библиотеке при компиляции и запуске мы передавали в параметр -classpath

Собственно, для перечисления нескольких библиотек в одном classpath, их пути нужно перечислять через разделитель, который тоже зависит от платформы:

- Windows: ; (точка с запятой)
 - Linux: : (двоеточие)

Данные константы как раз вернут подходящий разделитель для путей

Получение абсолютных путей

Есть несколько методов, связанных с абсолютными путями:

```
public boolean isAbsolute()  
public String getAbsolutePath()  
public File getAbsoluteFile()
```

- `isAbsolute()`
 - Проверяет, хранит ли этот `File` абсолютный путь
- `getAbsolutePath()`
 - Возвращает строку с абсолютным путём (то есть, позволяет превратить относительный в абсолютный)
- `getAbsoluteFile()`
 - То же самое, но возвращает сразу `File`

```
File absoluteFile = new File("/usr/bin/java");  
absoluteFile.isAbsolute(); // true  
absoluteFile.getAbsolutePath(); // /usr/bin/java  
  
File relativeFile = new File("readme.txt");  
relativeFile.isAbsolute(); // false  
relativeFile.getAbsolutePath(); // /home/romangurov/lectures/lecture-15/readme.txt
```

Относительный путь преобразуется относительно текущей рабочей директории



Получение частей пути

При помощи File можно получить из пути его подчасти

```
public String getPath()  
public String getName()  
public String getParent()  
public File getParentFile()
```

Достаточно увидеть пример, тут всё просто:

```
File file = new File("/usr/bin/java");  
String path = file.getPath();      // /usr/bin/java  
String name = file.getName();      // java  
String parent = file.getParent();  // /usr/bin
```

getParentFile(), вновь, отличается только тем, что возвращает сразу File



Символические ссылки

Символическая ссылка – особый тип файла (наряду с директориями)

В Windows символическая ссылка часто называется известным всем словом “ярлык”

Символическая ссылка не хранит данных, а лишь указывает некоторый путь к файлу, в который она ведёт

Она может указывать и на директорию, в таком случае она похожа на портал, ведущий из одного места файловой системы в другое



Канонический путь

Допустим, даны два пути и нужно узнать, ведут ли они в один и тот же файл

Обычный вызов `getAbsolutePath()` и сравнение путей не поможет, поскольку абсолютный путь может иметь в себе “.”, “..” или даже символическую ссылку

Для таких целей есть канонические пути:

```
public String getCanonicalPath() throws IOException  
public File getCanonicalFile() throws IOException
```

Канонический путь гарантирует быть абсолютным и уникальным – все лишние переходы будут схлопнуты

```
File file = new File("./prj/../symlinkdir/readme.txt");  
String canonicalPath = file.getCanonicalPath();  
// "/home/romangurov/readme.txt"
```

При этом для разрешения символических ссылок, методу приходится явно образаться к файловой системе и узнавать, на что каждая ссылка указывает

В случае ошибки такого обращения, может броситься `java.io.IOException`, оно проверяемое, его придётся обрабатывать

Проверка путей

Есть методы для проверки существования пути, и определения того, что по нему лежит:

```
public boolean exists()
public boolean isDirectory()
public boolean isFile()
```

```
public long lastModified()
public long length()
```

```
File java = new File("/usr/bin/java");
java.exists();           // true
java.isFile();           // true
java.isDirectory();      // false

java.length();           // 1337
java.lastModified();     // 1231914805000
```

- length()
 - Размер файла в байтах
- lastModified()
 - Дата последней модификации

```
public String[] list()
public String[] list(FilenameFilter filter)

public File[] listFiles()
public File[] listFiles(FilenameFilter filter)
public File[] listFiles(FileFilter filter)
```

```
File usrbin = new File("/usr/bin");
java.exists();           // true
java.isFile();           // false
java.isDirectory();      // true

java.list();             // String[]
java.listFiles();        // File[]
```

- list()
 - Массив путей всех файлов в директории
- listFiles()
 - То же самое, но сразу массив File

Все эти методы возвращают null, false или 0 в случае ошибки (то есть, не бросают исключений)

Фильтры для метода list()

У list() и listFiles() есть перегрузки, принимающие фильтр:

```
public String[] list()
public String[] list(FileNameFilter filter)

public File[] listFiles()
public File[] listFiles(FileNameFilter filter)
public File[] listFiles(FileFilter filter)
```

Фильтры – функциональные интерфейсы, принимающие путь и решающие, пропустить ли файл с таким путём
По сути, частный случай Predicate

```
package java.io;

@FunctionalInterface
public interface FileFilter {
    boolean accept(File pathname);
}
```

```
package java.io;

@FunctionalInterface
public interface FileNameFilter {
    boolean accept(File dir, String filename);
}
```

Например, можно отфильтровать содержимое директории, получив только файлы с расширением .java:

```
File dir = new File("C:\\Projects\\MyPreciousJavaFiles\\");
File[] javaSourceFiles = dir.listFiles(f -> f.getName().endsWith(".java"));
```



Операции с файловой системой

Теперь, рассмотрим операции, позволяющие вносить изменения в файловую систему

Создание нового (пустого) файла:

```
public boolean createNewFile() throws IOException
```

Возвращает false, если файл с таким именем уже существует
В случае ошибок, бросает IOException

```
try {  
    boolean success = file.createNewFile();  
} catch (IOException ex) {  
    // ...  
}
```

Для записи данных в файл, используются другие специальные классы, которые сами сумеют создать файл, если надо.

Поэтому этот метод используется редко



Создание директории

Для создания директорий есть два метода:

```
public boolean mkdir()  
public boolean mkdirs()
```

Если операция успешна, возвращают true, в любом другом случае (и при ошибках) – false

```
File dir = new File("a\\b\\c\\d");  
boolean success = dir.mkdir();  
boolean success2 = dir.mkdirs();
```

mkdir() может создать не больше одной директории, то есть только d

Если надо создать целиком всю цепочку директорий до указанной, то нужно использовать mkdirs()



Удаление

Метод `delete()` позволяет удалить файл или директорию:

```
public boolean delete()
```

У него так же все беды спрятаны в один `false`

```
boolean success = file.delete();
```

Важно, что для удаления директории, она должна быть пустой

То есть, потребуется сначала рекурсивно удалить из неё все вложенные файлы и поддиректории, а только потом её саму



Переименование

Есть метод для смены имени файла

```
public boolean renameTo(File dest)
```

По сути, переименование файла эквивалентно его перемещению на новый путь

Поэтому метод принимает File, что позволяет задать любой путь, по которому переедет файл

```
File file = new File("ab/o/ba.txt");  
boolean success = file.renameTo(new File("bo/o/ba.txt"));
```

Но, смена директории может провалиться по разным причинам, зависящим от платформы

Например, если пункт назначения окажется в другой файловой системе

И при этом ещё и все ошибки приводят просто к возврату false, понять причину проблемы будет сложно

Более-менее безопасным остается переименование в пределах той же директории

Метода для копирования файла вообще нет



java.nio.file.Path

Класс `File` устроен странно: один метод бросает исключение, другой нет

При этом, возвращение `boolean` очевидно хуже исключения (на лекции про исключения уже обсуждали) – банально невозможно понять почему возникла ошибка

Так как в мире уже было много кода, использующего `java.io.File`, исправить поведение было невозможно

Поэтому, было решено сделать новый, более продуманный набор классов для работы с ФС

Роль класса `File` – хранения путей – тут исполняет интерфейс `Path`:

```
public interface Path extends Comparable<Path>, Iterable<Path>, Watchable {  
    // ...  
}
```



java.nio.file.Path

Роль класса File – хранения путей – тут исполняет интерфейс Path:

```
public interface Path extends Comparable<Path>, Iterable<Path>, Watchable {  
    // ...  
}
```

Чтобы получить объект для интерфейса Path, используется статический метод соседнего класса Paths:

```
public static Path get(String first, String... more)
```

Можно склеить сразу несколько путей в один

```
Path path = Paths.get("Projects", "lectures", "lecture_test");  
// "Projects\lectures\lecture_test"
```

Для совместимости со старыми классами, есть методы для конвертации:

```
File fromPath = path.toFile();  
  
Path fromFile = fromPath.toPath();
```

Продвинутый разбор пути

Есть методы разбора пути как у File:

```
Path java = Paths.get("/usr/bin/java");
java.isAbsolute();    // true
java.getFileName();  // java
java.getParent();     // /usr/bin
```

Но есть и новые:

```
int getNameCount();
Path getName(int var1);
boolean startsWith(String other);
Path resolveSibling(String other);
Path relativize(Path var1);
```

- Количество частей в пути (по разделителям)
- Получить i-ую часть пути начиная слева
- Проверка, что один путь является префиксом другого
- Получение пути файла-соседа
- Получить относительный путь к текущему относительно любого другого

```
java.getNameCount();           // 3
java.getName(0);               // usr
java.resolveSibling("javap");   // /usr/bin/javap
java.startsWith("/usr");       // true
Paths.get("/usr").relativize(java); // bin/java
```

И это ещё далеко не всё, хорошо, что есть [документация](#)



Доступ к ФС

Интерфейс Path устроен таким образом, что никакие его методы не требуют доступа к файловой системе, а значит не бросают ошибок этого характера

По сути, Path – это просто умно завёрнутая строка, все операции проводятся чисто со строками, без какой-либо связи с реальностью

Чтобы сделать что-то с файловой системой, придётся вызывать статические методы класса Files:

```
Path java = Paths.get("/usr/bin/java");
Files.exists(java);
Files.isRegularFile(java);
Files.size(java);
Files.getLastModifiedTime(java).toString();
```

Умеет всё то же самое, и даже больше,
и всё с нормальными исключениями



Доступ к ФС

Исправлены даже копирование и перемещение:

```
public static Path copy(Path source, Path target, CopyOption... options) throws IOException
public static Path move(Path source, Path target, CopyOption... options) throws IOException
```

Копирование в принципе появилось, а перемещение теперь не ограничено невозможностью переноса на другую файловую систему

```
Path java = Paths.get("/usr/bin/java");
Files.copy(java, Paths.get("/usr/bin/java_copy"), StandardCopyOption.REPLACE_EXISTING);
```

Ну и создание директорий работает похожим принципом, но с нормальными исключениями:

```
public static Path createDirectory(Path dir, FileAttribute<?>... attrs) throws IOException
public static Path createDirectories(Path dir, FileAttribute<?>... attrs) throws IOException
```

Все подробности методов легко подсмотреть в документации класса [Files](#)



Итерирование по файлам в директории

Вместо методов `list()` и `listFiles()` используется отдельный объект интерфейса `DirectoryStream`

Получить его можно методом `Files.newDirectoryStream()`:

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

`DirectoryStream` является `Iterable`, поэтому позволяет просто проитерироваться по всем файлам:

```
try (DirectoryStream<Path> dirStream = Files.newDirectoryStream(usrbn)) {
    for (Path child : dirStream) {
        System.out.println(child);
    }
} catch (IOException ex) {
    // ...
}
```

`DirectoryStream` требуется закрывать, поэтому используем блок `try-with-resources`

Такой подход позволяет обработать папку с очень большим количеством файлов потоково — то есть, без хранения всех элементов

Рекурсивный обход

Все предыдущие методы для получения содержимого директории не посещали вложенные директории

Для упрощения произведения рекурсивного обхода дерева директорий, существует метод `Files.walkFileTree()`:

```
public static Path walkFileTree(Path start, FileVisitor<? super Path> visitor) throws IOException
```

```
public interface FileVisitor<T> {  
    FileVisitResult preVisitDirectory(T var1, BasicFileAttributes var2) throws IOException;  
    FileVisitResult visitFile(T var1, BasicFileAttributes var2) throws IOException;  
    FileVisitResult visitFileFailed(T var1, IOException var2) throws IOException;  
    FileVisitResult postVisitDirectory(T var1, IOException var2) throws IOException;  
}
```

`walkFileTree()` производит обход дерева и вызывает методы [визитора](#) для каждого файла в определённые моменты

Есть реализация интерфейса – `SimpleFileVisitor`, каждый метод которого не делает ничего, от него удобно наследовать свои визиторы, чтобы не определять каждый раз все четыре метода

Рекурсивный обход

Рассмотрим визитор для рекурсивного удаления всей директории:

```
Path directory = Paths.get("C:\\Projects");

Files.walkFileTree(directory, new SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException {
        Files.delete(file);
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult postVisitDirectory(Path dir, IOException ex) throws IOException {
        if (ex == null) {
            Files.delete(dir);
            return FileVisitResult.CONTINUE;
        } else {
            throw ex;
        }
    }
});
```



Виртуальные файловые системы

java.nio позволяет очень гибко работать с разными файловыми системами

Можно даже открыть zip-архив как самостоятельную файловую систему и работать с ним в привычном стиле:

```
Path zipPath = Paths.get("jdk/src.zip");

try (FileSystem zipfs = FileSystems.newFileSystem(zipPath, null)) {
    for (Path path : zipfs.getRootDirectories()) {
        Files.walkFileTree(path, new SimpleFileVisitor<Path>() {
            @Override
            public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException {
                System.out.println(file);
                return FileVisitResult.CONTINUE;
            }
        });
    }
}
```