ТЕХНОПОЛИС

Advanced Java. Работа с файлами. Блокирующий IO

Галкин Александр Сергеевич

Оглавление

Файловая система

Работа с диском может заменить вам БД.

Папки и файлы

File или Path

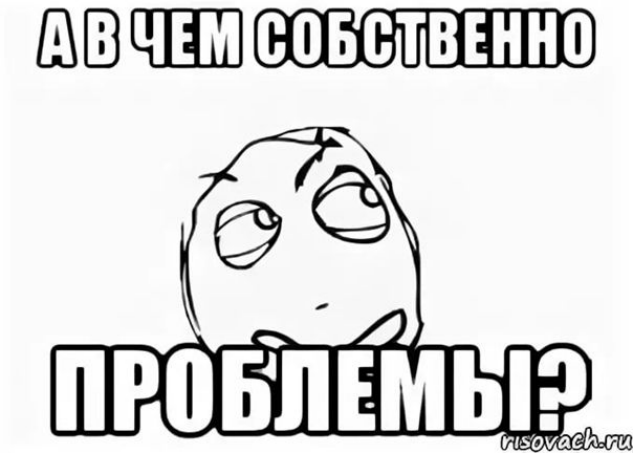
Не обязательно HDD

- Класс файл представляет файл или директорию в файловой системе
- Задается с помощью относительного или абсолютного пути
- `static final String separator`
- Работа с путями: `String getName(), String getParent(), File getCanonicalFile(), String getAbsolutePath()`

- Объекты класс File никак не привязаны к реальным директориям и файлам на диске
- Методы `boolean exists()`, `boolean isDirectory()`, `boolean isFile()` проверяют существование и тип объекта
- Файлы: `long length()`, `long lastModified()`
- Директории: `String[] list()`, `File[] listFiles(FilenameFilter filter)`
- Создание файла: `boolean createNewFile()` - кидает исключение
- Создание директории: `boolean mkdir()`, `boolean mkdirs()` - не кидают исключения

- Переименование (и перенос в другую директорию): `boolean renameTo(File dest)`
- Метод копирования отсутствует
- Удаление файла или пустой директории: `boolean delete()`
- Удаление не пустой папки надо писать самому с рекурсивным обходом

java.io.File. Недостатки



- Почти все методы возвращают true или false — не можем узнать причину ошибки
- Много методов не реализовано
- Обратная совместимость - нельзя просто взять и поменять контракты

Что с этим делать?

- Написать новый API

- ***Path*** это интерфейс, а не класс
- Не связан с файлам на диске
- По сути это просто строка с набором методов
- Создание: `Paths.get("polis/homework2");`
- Методы конвертации: `path.toFile(); file.toPath();`
- ***Path*** имеет аналоги всех синтаксических операций из класса ***File***
- Также многие методы доработаны, например

```
Path getName(int index)
```

- ***Path*** не имеет методов доступа к файловой системе в отличие от класса ***File***

- Почти любой доступ к файловой системе получается через статические методы класса ***Files***
- В ***Files*** есть аналоги всех операций из класса ***File*** и даже больше, например есть копирование:

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

- Работа с существующими директориями осуществляется с помощью метода в классе ***Files*** и нового объекта:

```
DirectoryStream<Path> newDirectoryStream(Path dir)
```

- ***DirectoryStream*** - это интерфейс, представляющий директорию открытую на чтение, поэтому его нужно закрывать и освобождать ресурсы
- ***DirectoryStream*** должен использоваться в блоке try с ресурсами
- Имеет всего два метода: `void close()` и `Iterator<T> iterator();`
- Загружает директории последовательно, что удобно, т. к. директории могут быть очень большими

- API рекурсивного удаления все также нет
- Добавлен новый механизм рекурсивного обхода:

```
Path walkFileTree(Path start,FileVisitor<? super Path> visitor)
```

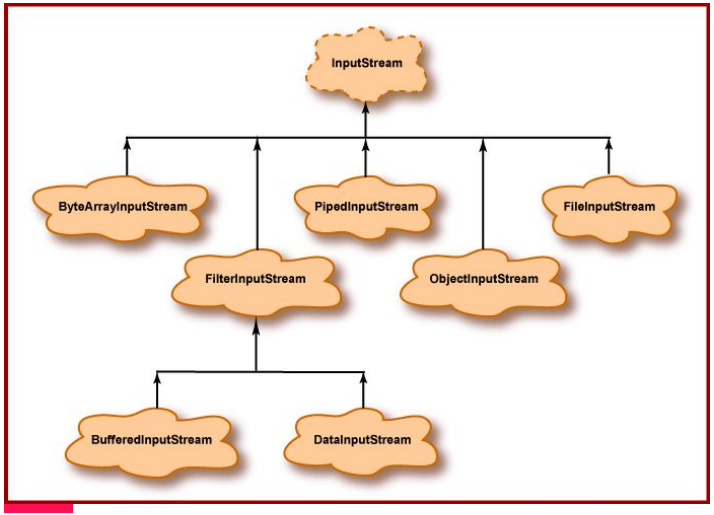
- Интерфейс ***FileVisitor*** содержит методы, в которых описывается что делать перед входом в dir, после выхода из dir, после нахождения файла и после того, как файл нашли, но не смогли прочитать атрибуты
- `SimpleFileVisitor<T>` — простейшая реализация из заглушек

Input/Output stream

Только байты, только хардкор

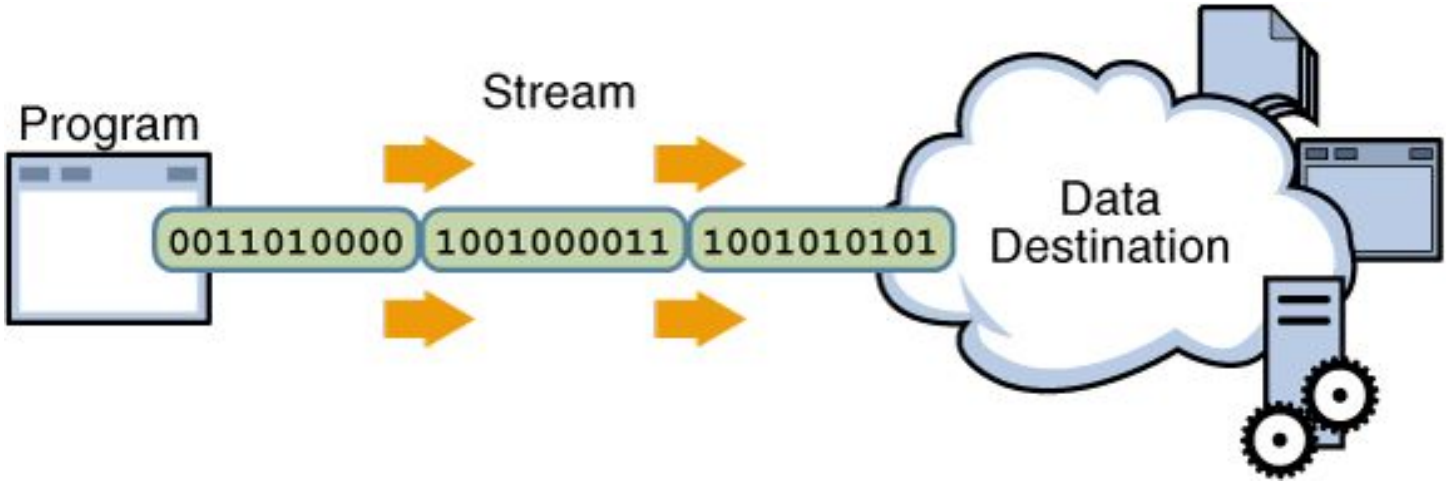
Низкоуровневые потоки

Высокоуровневые потоки



И это только верхушка айсберга

- Данные читаются/записываются из потока/в поток без какого либо кэширования
- Надо сделать буффер, из/в которого/который мы будем читать/писать



- ***java.io.InputStream*** - поток байтов из которого можно читать
- ***java.io.OutputStream*** - поток байтов в который можно писать
- Это абстрактные классы реализующие интерфейс ***Closeable***, т.е. экземпляры наследников данных классов нужно закрывать
- Неизвестно откуда мы читаем и куда мы пишем, эта информация зависит от наследников

- `int read()` — читает 1 байт из потока и сдвигается на 1 байт
- int read(byte b[]) — читает ***b.length*** байт и записывает их в массив, возвращает количество прочитанных байт
- `int read(byte b[], int off, int len)` — читает ***len*** байт и пишет в массив начиная с индекса ***off***
- `long skip(long n)` — пропускает ***n*** байт
- `void mark(int readlimit)`, `void reset()` — метод ***mark*** помечает байт с которого начнется чтение, если вызвать метод ***reset***
- Все методы могут бросить ***IOException***

- `void write(int b)` — пишем 1 байт в поток
- `write(byte b[])` — пишем массив байт в поток
- `void write(byte b[], int off, int len)` — пишем ***len*** байт из массива в поток, с позиции ***off***
- `void flush()` — сбрасывает промежуточные буфера, где хранятся данные перед передачей их операционной системе
- Просто вызов метода write не гарантирует, что операционная система получит данные, но метод ***close*** внутри себя вызывает метод ***flush***
- Все методы могут бросить ***IOException***

- ***FileInputStream***, ***FileOutputStream*** — для работы со старым API (конструктор принимает или строку - путь до файла, или экземпляр класса ***File***)
- `InputStream newInputStream(Path path, OpenOption... options)`, `OutputStream newOutputStream(Path path, OpenOption... options)` — методы для получения стримов с помощью нового API

Дополнительные варианты стримов

- `OutputStream outputStream =socket.getOutputStream();`
  `InputStream inputStream = socket.getInputStream();`
  — стримы сетевого соединения
- ***ByteArrayInputStream***, ***ByteArrayOutputStream*** — стрим который можно создать с помощью существующего массива байт и стрим из которого этот массив байт можно получить

- ***FilterInputStream***, ***FilterOutputStream***
- Стримы можно заворачивать друг в друга (внутренний — низкоуровневый, внешний реализует высокоуровневые методы)
- `public DataOutputStream(OutputStream out)` — оборачивает некоторый стрим и добавляет методы для записи примитивов и строк
- Аналогично
  `public DataInputStream(InputStream in)`

- ***BufferedInputStream***, ***BufferedOutputStream*** — содержат буфер для ввода/выводы сразу нескольких байтов
- ***PushbackInputStream*** — умеет записывать данные обратно в поток
- ***SequenceInputStream*** — читает из нескольких стримов по очереди
- ***DataInputStream***, ***DataOutputStream*** — добавляет методы для чтения/записи примитивов и строк

- Добавляет методы для записи примитивов, строк и объектов
- Делает внутри себя два преобразования:
- Объект в строку
- Строку в последовательность байт
- Не бросает исключений, выставляет флаг

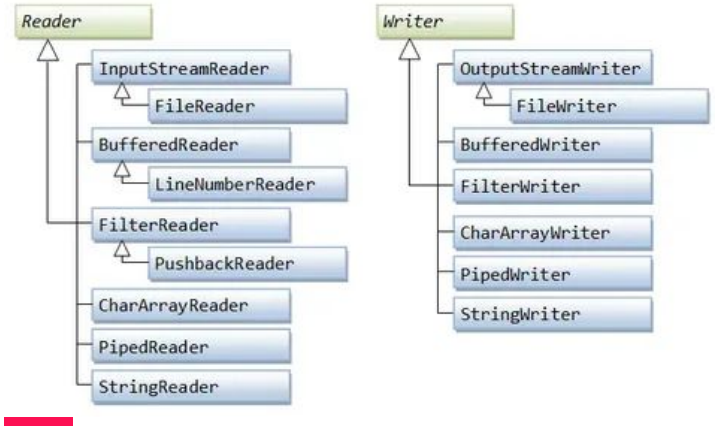
- ***DataInput***, ***DataOutput*** — реализуемые интерфейсы, НЕ стримы
- Позволяет позиционироваться внутри файла

Reader/Writer

Работа с символами

Низкоуровневые потоки

Высокоуровневые потоки



Scanner прячется где-то рядом

- ***java.io.Reader*** — поток символов из которого можно читать
- ***java.io.Writer*** — поток символов в который можно писать
- Это абстрактные классы реализующие интерфейс ***Closeable***, т.е. экземпляры наследников данных классов нужно закрывать
- Неизвестно откуда мы читаем и куда мы пишем, эта информация зависит от наследников

- ***BufferedReader***, ***BufferedWriter*** — содержат буфер для ввода/выводы сразу нескольких символов. Может записывать строки
- ***PushbackReader*** — умеет записывать данные обратно в поток
- ***CharArrayReader***, ***CharArrayWriter*** — в качестве данных выступает массив символов
- ***StringReader***, ***StringWriter*** — в качестве данных выступает строка

- `PrintWriter (Writer out)` — создаем удобный ***Writer*** на основе заданного символьного стрима
- `void print(int i)` — пишем целое число в поток
- `void print(String s)` — пишем строку в поток
- `void print(Object obj)` — пишем объект как строку (используется метод ***toString()***)
- `PrintWriter format(String format, Object ... args)` — аналог записи строки с параметрами из C++
- Методы НЕ кидают исключения, они просто устанавливают флаг ошибки, который можно прочитать с помощью метода `boolean checkError()`

Конвертация потока байт в поток символов

- `InputStreamReader(InputStream in, String charsetName)` — ***charsetName*** это название кодировки, например UTF-8
- `OutputStreamWriter(OutputStream out, Charset cs)` — ***cs*** и есть объект кодировки
- Основные кодировки лежат в классе ***StandardCharsets***
- Если не указывать ничего кроме стрима, будет использовать кодировка по умолчанию

- `FileReader(String fileName)`, `FileReader(String fileName)` — классы для строкового чтения и записи
- `Reader reader = new InputStreamReader( new FileInputStream(fileName), StandardCharsets.UTF_8)`
- `Writer writer = new OutputStreamWriter( new FileOutputStream(fileName), StandardCharsets.UTF_8);`
- Второй способ предпочтительнее, так как можно указать кодировку

Высокоуровневое чтение файлов

- Метод для чтения с помощью ***BufferdReader*** в классе ***Files***

```
BufferedReader newBufferedReader(Path path,Charset cs)
```

- Метод для чтения небольших файлов

```
List<String> readAllLines(Path path, Charset cs)
```

- Метод для записи с помощью ***BufferdWriter*** в классе Files

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

- Метод для записи нескольких строк в файл

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

Не является ***Reader***-ом и ***InputStream***-ом

- Аналог ***PrintWriter*** только для чтения
- Не является ***Reader***-ом и ***InputStream***-ом
- `public Scanner(InputStream source)` — можно создать на основе стрима и ридера
- `String next()` — возвращает следующую строку
- `double nextDouble()` — возвращает следующий дабл
- `boolean hasNextLine()` — возвращает истину, если дальше есть еще строка
- `Scanner useDelimiter(Pattern pattern)` — устанавливает разделить между элементами, по умолчанию это пробелы

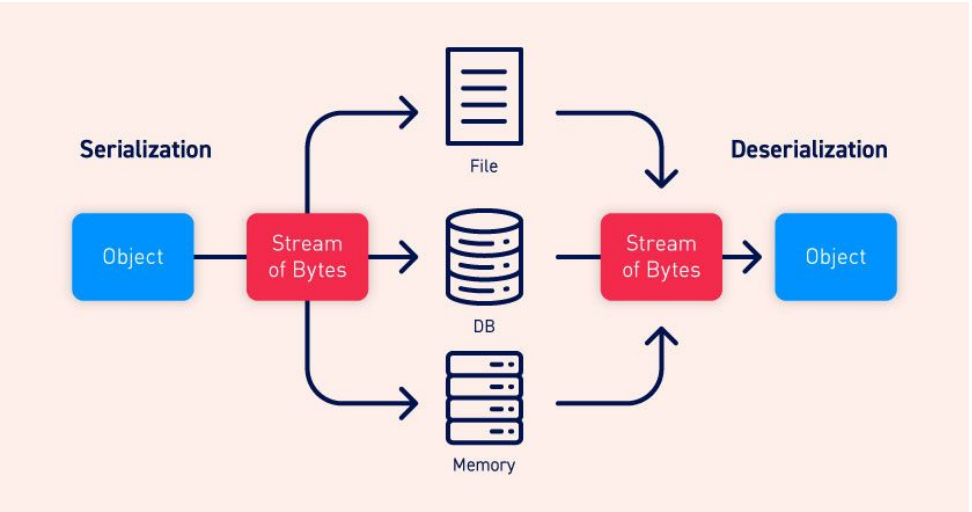
- ***System.in*** — ***InputStream***, для работы с текстом - удобно обернуть в ***Scanner***
- ***System.out***, ***System.err*** — ***PrintStream***, можно выводить двоичные данные (метод ***write***) и текстовые (метод ***print***)

Сериализация объектов

Так ли все просто?

Автоматическая

Ручная



А выглядит просто!

- `interface Serializable` — говорит *jvm*, что этот объект можно сериализовать в поток байт. *Jvm* это делает сама
- Если какое-то поле объекта не нужно сериализовать, его надо пометить идентификатором ***transient***
- Все поля объекта ***Serializable***, должны быть или ***transient***, или Serializable или примитивами
- `interface Externalizable extends java.io.Serializable` — ручная сериализация с помощью методов

```
void writeExternal(ObjectOutput out)
void readExternal(ObjectInput in)
```

Для сериализации и десериализации объектов, помеченных как ***Serializable***, используются следующие стримы

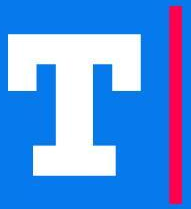
- `class ObjectOutputStream` и метод `void writeObject(Object obj)`
- `class ObjectInputStream` и метод `Object readObject()`

Сериализация. Порядок записи

- Основной класс и его поля с типами
- Суперклассы и их поля с типами
- Сами данные в обратном порядке (от супер класса к исходному)
- Если в данных содержится объект, то работаем с ним с первого шага


ОТЛИЧНАЯ ФУНКЦИЯ СЕРИАЛИЗАЦИИ
ИСПОЛЬЗОВАТЬ ЕЁ Я КОНЕЧНО НЕ БУДУ
risovach.ru

- `interface Serializable`
  - `void writeObject(ObjectOutputStream obj)`
  - `void readObject(ObjectInputStream obj)`
- `interface Externalizable`
  - `void writeExternal(ObjectOutput out)`
  - `void readExternal(ObjectInput in)`
- Руками в отдельном методе записываем поля в заданном порядке



Спасибо за внимание!