Java IO

Croc Java School

Потоки ввода-вывода

Поток ввода-вывода

Последовательность данных одного типа, которые можно либо писать, либо читать.

С потоком, как правило, ассоциирован источник данных:

- файл
- сетевое соединение
- буфер в памяти
- системный поток процесса

Поток vs массив

В отличие от массивов в потоках нет понятия индексов.

Нет возможности адресовать элемент с произвольным положением внутри потока.

Поток можно обработать только последовательно.

Поток vs связный список

В отличие от связанных списков в потоках нет понятия длины.

Узнать, закончился ли поток, можно только при чтении очередного элемента.

Потоки в Java

Байтовые

- `InputStream`
- `OutputStream`

Символьные

- `Reader`
- `Writer`

Байтовые потоки

java.io.InputStream

Базовый класс для всех потоков ввода данных (на чтение).

```
int read() throws IOException;

int read(byte[] b) throws IOException;

int read(byte[] b, int off, int len) throws IOException;
```

```
result == -1 => end of stream
```

Чтение файла

```
InputStream in = new FileInputStream("input.txt");
int b;
while ((b = in.read()) != -1)
    System.out.println("Next byte: " + (byte)b);
```

Потоки и ресурсы

Потоки часто связаны с ресурсами, поэтому их необходимо освобождать (закрывать) после использования.

```
InputStream in = null;
try {
    in = new FileInputStream("input.txt");
    int b;
    while ((b = in.read()) != -1)
        System.out.println("Next byte: " + b);
} finally {
    if (in != null)
        in.close();
}
```

Потоки и ресурсы (try-with-resources)

Конструкция try-with-resources применима к потокам, так как они реализуют интерфейс `Closeable`.

```
try (InputStream in = new FileInputStream("input.txt")) {
    int b;
    while ((b = in.read()) != -1)
        System.out.println("Next byte: " + b);
}
```

Закрытый поток

После закрытия поток, как правило, больше нельзя использовать.

available()

Не путать с длиной.

Метод возвращает доступное *на текущий момент* для чтения количество байтов.

“На текущий момент” означает, что данных в потоке может быть больше, но для их получения может потребоваться дополнительное ожидание.

java.io.OutputStream

Базовый класс для всех потоков вывода данных (на запись).

```
void write(int b) throws IOException;
write(byte b[]) throws IOException;
void write(byte b[], int off, int len) throws IOException;
void flush() throws IOException;
```

flush()

Принудительная запись накопленных в буфере данных.

Данные из временной памяти (поля класса) переносятся в постоянную (физический источника данных: файл, сеть, проч.)

Запись файла

```
try (OutputStream out = new FileOutputStream("output.txt")) {
    String str = "TODO";
    out.write(str.getBytes(Charset.forName("UTF-8")));
}
```

Нет необходимости использовать flush, если поток корректно закрывается.

Запись в конец файла

Параметр конструктора append (по умолчанию false).

```
String message = ...
try (OutputStream out = new FileOutputStream(“log.txt”, true))
    out.write(message.getBytes());
)
```

Фильтрующие потоки

Не связаны с ресурсами.

Трансформируют логику чтения или записи.

Chaining.

Базовые классы:

- `FilterInputStream`
- `FilterOutputStream`

Буферизация потоков

```
InputStream in = new BufferedInputStream(
    new FileInputStream(“input.txt”));
```

Чтение файла по одному байту медленное.

Буферизированный фильтр читает порцию данных во временную память (буфер) и работает с ней даже при чтении по одному байту.

Существенно ускоряет работу с файлами и другими медленными источниками данных.

Конкатенация потоков

```
InputStream in = new SequenceInputStream(
    new FileInputStream(“input.txt.0”),
    new FileInputStream(“input.txt.1”));
```

Chaining

```
InputStream in = new FileInputStream(“source.txt”);
in = new SequenceInputStream(
    in,
    new FileInputStream(“source-ext.txt”)
);
in = new BufferedInputStream(in);
```

Chaining

То же самое, но запись лаконичнее.

```
InputStream in = new BufferedInputStream(
    new SequenceInputStream(
        new FileInputStream(“source.txt”),
        new FileInputStream(“source-ext.txt”)
    )
);
```

Стандартные потоки ввода-вывода

```
System.in

System.out

System.err
```

Символьные потоки

java.io.Reader

Трансформирует байты в символы в зависимости от указанной кодировки.

Кодировка — способ представления символов в виде последовательности байтов (не всегда фиксированной длины).

```
try (Reader r = new InputStreamReader(
        new FileInputStream("input.txt"), "Windows-1251")) {
    int c;
    while ((c = r.read()) != -1)
        System.out.print((char)c);
}
```

Реализации

`InputStreamReader`

`FileReader`

`CharArrayReader`

`BufferedReader`

Построчное чтение файла

```
try (BufferedReader r = new BufferedReader(new FileReader("log.txt"))) {
     String line;
     while ((line = r.readLine()) != null)
          System.out.println("Log line: " + line);
}
```

java.io.Writer

По аналогии с `Reader`.

Форматирование

```
java.util.Scanner

java.io.DataInputStream

java.io.DataOutputStream
```

Построчное чтение файла (Scanner)

```
try (Scanner s = new Scanner(new FileReader("log.txt"))) {
    while (s.hasNextLine())
        System.out.println("Log line: " + s.nextLine());
}
```

Работа с файлами

java.io.RandomAccessFile

В отличие от потокового представления файла обеспечивает доступ по произвольному смещению.

```
try (RandomAccessFile f = new RandomAccessFile("data.txt", "rw")) {
    f.seek(100);
    f.write("100 bytes before".getBytes());
}
```

java.io.File

Доступ к функциям файловой системы, таким как:

- Проверка существования файла.
- Смена названия и удаление файлов.
- Создание директорий.
- Создание временных файлов.

java.nio.file

Содержит набор классов и функций для работы с файловой системой.

Основные интерфейсы/классы:

- `Path`
- `Paths`
- `Files`

Удаление файла (old way)

```
File file = new File(“secret.txt”);
if (file.exists())
    file.delete();
```

Удаление файла (new way)

```
Path path = Paths.get("secret.txt");

Files.deleteIfExists(path);
```

Сканирование иерархии файлов

```
Path root = Paths.get("root");


Stream<Path> topLevel = Files.list(root);
Stream<Path> top3Levels = Files.walk(root, 3);
```

FileVisitor УДАЛЯЕТ ФАЙЛЫ РЕКУРСИВНО И НАВСЕГДА!!!

```
public class DeleteFileVisitor implements FileVisitor<Path> {

  public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
      throws IOException {
    Files.delete(file);
    return FileVisitResult.CONTINUE;
  }
  // ...
}

Path toDelete = Paths.get("trash");
Files.walkFileTree(toDelete, new DeleteFileVisitor());
```

Немного (еще) непривычного синтаксиса

Выводим все строки из файла, длина которых больше трех символов

```
Path path = Paths.get("resources/words.txt");
Files.lines(path, StandardCharsets.UTF_8)
    .filter(word -> word.length() > 3)
    .forEach(System.out::println);
```

Читаем все строки из файла в список

```
Path path = Paths.get("resources/words.txt");
List<String> words = Files.readAllLines(path, StandardCharsets.UTF_8);
```

Нетворкинг (но не тот, что с Наташей)

Сокеты

Приложения взаимодействую по сети через сокеты. Сокет — программная абстракция, которую мы используем для:

- установки соединения
- отправки данных
- получения данных

Создание сокета

Сервер

```
ServerSocket serverSocket = new ServerSocket(2021); // 0 auto port
Socket socket = serverSocket.accept(); // ожидание соединения
```

Клиент

```
Socket socket = new Socket("127.0.0.1", 2021);
```

Получение и отправка данных

Поток для чтения данных (получаем)

```
InputStream in = socket.getInputStream();
```

Поток для записи данных (отправляем)

```
OutputStream out = socket.getOutputStream();
```

В действии

```
try (Socket socket = new Socket("cat-fact.herokuapp.com", 80)) {
   Writer w = new OutputStreamWriter(socket.getOutputStream());
   w.write("GET /facts/random HTTP/1.1\r\n");
   w.write("Host: cat-fact.herokuapp.com\r\n");
   w.write("\r\n");
   w.flush();

   InputStream in = socket.getInputStream();
   int b;
   while ((b = in.read()) != -1)
       System.out.write(b);
}
```

Шаги

1. Устанавливаем соединение с сервером cat-fact.herokuapp.com по удаленному порту 80
2. Отправляем HTTP запрос (как набрать http://cat-fact.herokuapp.com/facts/random в браузере)
3. Убеждаемся, что данные отправлены на сервер (flush)
4. Читаем ответ от сервера и выводим его на экран

Неблокирующие сокеты

java.nio.channels

Методы сокетов (`accept`, `read`, `write`) блокируют текущий поток исполнениия, что может быть проблемой для приложений с большим количеством подключений. (На каждое подключение требуется отдельный поток.)

Java предоставляет API для работы с сокетами в неблокирующем режиме.

`SocketChannel`

`ServerSocketChannel`

`Selector`

Каналы и селекторы

Каналы ассоциируются с конкретными соединениями и позволяют работать с буферами соединений в асинхронном режиме.

Селекторы позволяют проверять “наличие” событий в разных каналах без необходимости заблокировать поток до наступления события. (Пример события: “поступил запрос на входящее соединение”.)

API достаточно сложный для реализация и отладки, но есть готовые фреймворки.

https://netty.io/

java.nio.ByteBuffer

Быстрый низкоуровневый ввод-вывод

mark <= position <= limit <= capacity