IO в Java

Юрий Литвинов
yurii.litvinov@gmail.com

23.01.2019

1. Введение

В Java ввод-вывод довольно развит, есть даже несколько пакетов стандартной библиотеки, реализующих функциональность файлового ввода вывода и делающих в принципе одно и то же несколько по-разному. Сегодня речь пойдёт про исторически первый и самый простой способ читать/писать данные, пакет java.io.

Основа системы ввода вывода — понятие “Поток”, точнее, отдельно “Поток ввода” и “Поток вывода”. Для того, чтобы говорить потокам, откуда читать и куда писать, применяются либо просто строковые пути до файлов, либо файловые дескрипторы (как в C++) — класс File. File же представляет кроссплатформенное API для работы с файловой системой — именно там есть методы, проверяющие наличие файла, его атрибуты, методы создания, удаления, перемещения и т.д. File же, контринтуитивно, отвечает за работу с папками — листинг, создание и удаление.

Потоки в стандартной библиотеке представляются несколькими интерфейсами (на самом деле, абстрактными классами). InputStream, OutputStream представляют байтовые потоки — они позволяют осуществлять ввод-вывод на уровне байтов или массивов байт. Reader, Writer представляют символьные потоки, они читают/пишут char-ы в определённой кодировке.

Конкретные реализации этих абстракций можно разбить на несколько категорий:

- File* (FileInputStream, FileOutputStream, FileReader, FileWriter) — работа с файлами
- Buffered* — оптимизация с помощью буферизации
- Data* —работа с данными
- ByteArray* — потоки в оперативной памяти

Интересно, что раз это Java, тут всё архитектурно красиво, поэтому потоки могут вкладываться друг в друга, предоставляя “слои” функциональности (паттерн “Декоратор”, если по-научному). Например, файловые потоки умеют читать только посимвольно, но если мы “наденем” на FileReader буферизующий поток, то, во-первых, получим прирост производительности раз в 10 за счёт буферизации чтения, во-вторых, получим возможность читать построчно:

```
new BufferedReader(new FileReader("path/to/file"));
```

Причём, поскольку BufferedReader-у всё равно, что за поток в него вложен, лишь бы это был Reader, то его можно "надеть" на любую реализацию Reader-а — из файла, из памяти, по сети и т.д.

Как обычно, при начале работы потоки открывают тот файл, с которым работают, и ожидают, что в конце работы файл будет закрыт. Но тут проблема — в Java нет деструкторов, и то, что в C++ делал деструктор сам, в Java требуется делать явно (ну, почти), вызывая у потока метод close(). Либо попросив компилятор вызвать close() автоматически, с помощью синтаксиса try-with-resources, о котором чуть позже. try-with-resources — самое близкое к деструкторам C++, и единственный идеологически правильный способ закрывать потоки, потому что с close() надо думать, не пропустим ли мы вызов close() из-за случайного исключения. Как и обычно, потоки надо закрывать обязательно, и чем раньше, тем лучше.

2. Потоки, основные операции

На самом деле, сейчас будет краткий пересказ JavaDoc-ов стандартной библиотеки, и при любом вопросе желательно обратиться к первоисточнику (https://docs.oracle.com/en/java/javase/11/docs/api/index.html). Тем не менее, наиболее важные штуки приведены тут. Что можно делать с потоками ввода:

- `int read()` — чтение элемента. Возвращает *двухбайтовый* беззнаковый char (число от 0 до 65535), либо -1, если достигнут конец потока.
- `read(T[] v)`, `read(T[] v, off, len)` — чтение элементов в массив. Возвращает число прочитанных элементов, либо -1, если ничего не прочиталось. Обратите внимание, в Java массивы знают свой размер, поэтому переполнения буфера опасаться не стоит.
- `skip(n)` — пропуск n элементов. Возвращает, сколько символов реально пропущено.
- `available()` — сколько элементов доступно.
- `mark(limit)` — пометка текущей позиции, чтобы потом можно было на неё вернуться reset-ом. limit — сколько символов мы можем считать, сохраняя при этом возможность вернуться. Если из потока к моменту вызова reset считано больше символов, reset имеет право не отработать (это чтобы не было так, что из-за забытого mark-а мы буферизовали 20 гигабайт сетевого трафика). не все реализации стримов поддерживают mark.
- `reset()` — возврат к помеченной позиции.
- `close()` — закрытие потока. Закрывает также вложенные потоки (хотя это надо уточнять в документации про каждый конкретный класс). Этот метод реализует метод из интерфейса Closeable, который, в свою очередь, реализует интерфейс AutoCloseable. Про AutoCloseable знает компилятор, так что для любого класса, реализующего AutoCloseable, работает try-with-resources — генерируется вызов close(), который отработает и при корректном завершении работы, и при исключении.

Вот что можно делать с потоками вывода:

- `write(int v)` — запись элемента (двухбайтового символа).
- `write(T[] v)` — запись массива элементов.
- `write(T[] v, off, len)` — запись части массива (начиная с индекса off, len ячеек).
- `flush()` — запись буфера. Если поток работает поверх ещё какой-нибудь реализации потока вывода, делает flush() и ей. Метод объявлен в интерфейсе Flushable.
- `close()` — закрытие потока. Тут действуют те же соображения про AutoCloseable, что и с потоками чтения.

Например, как скопировать данные из одного потока в другой:

```
void copy(InputStream is, OutputStream os) throws IOException
{
    var b = new byte[1024];
    int c = 0;
    while ((c = is.read(b)) > 0) {
        os.write(b, 0, c);
    }
}
```

Или как считать байты из файла и побайтово сделать что-нибудь:

```
FileInputStream in = null;

try {
   in = new FileInputStream("test.txt");
   int c;

   while ((c = in.read()) != -1) {
       doSomething(c);
   }
} finally {
   if (in != null) {
       in.close();
   }
}
```

3. try-with-resources

Обратите внимание на страшную конструкцию с finally, чтобы если исключение было брошено, файл всё равно закрылся. Как раз для этого и придуман try-with-resources, и начиная с Java 7 правильнее (и короче) писать так:

```
try (var in = new FileInputStream("test.txt")) {
   int c;

   while ((c = in.read()) != -1) {
       doSomething(c);
   }
}
```

Если ресурсов, с которыми надо работать, много, они указываются через точку с запятой:

```
void copy(String src, String dst) throws IOException
{
    try (var in = new FileInputStream(src);
            var out = new FileOutputStream(dst)) {
        var b = new byte[1024];
        int c = 0;
        while ((c = in.read(b)) > 0) {
            out.write(b, 0, c);
        }
    }
}
```

4. Исключения

Основные исключения, бросаемые операциями ввода вывода, таковы. IOException — это корень иерархии исключений подсистемы ввода-вывода, если вы хотите ловить ситуации, когда просто с вводом-выводом что-то не так, имеет смысл ловить именно его. EOFException — когда достигнут конец потока, но вы пытаетесь сделать что-то, что ожидает данные в потоке. Обратите внимание, что read() это исключение не бросает, а просто возвращает -1. FileNotFoundException кидается, когда запрошенный файл не найден, оказался папкой, у вас нет на него прав, он вообще только для чтения и т.д. UnsupportedEncodingException бросается методами, которые принимают кодировку, если передать им кодировку, которая неизвестна Java-машине, на которой они работают.

5. Преобразование потоков

Как уже упоминалось ранее, потоки можно комбинировать, “вкладывая” один в другой. Вот самые распространённые случаи такого комбинирования:

- При чтении возможно преобразование байтового потока в символьный, с указанием кодировки. Это делает класс InputStreamReader, с конструктором `InputStreamReader(InputStream, encoding?)`, принимающим опциональный параметр с кодировкой. Если кодировка не указана, используется кодировка из текущей локали.

- Наоборот, из символьного в байтовый поток преобразует OutputStreamWriter, с конструктором `OutputStreamWriter(OutputStream, encoding?)`. Тут с кодировками дела обстоят так же, как и у InputStreamReader-а

Небольшой пример, перекодирование файла из кодировки 1251 в кодировку 866:

```
Reader reader = new InputStreamReader(
    new FileInputStream("input.txt"), "Cp1251");
Writer writer = new OutputStreamWriter(
    new FileOutputStream("output.txt"), "Cp866");
int c = 0;
while ((c = reader.read()) >= 0) {
    writer.write(c);
}

reader.close();
writer.close();
```

Ну или более правильно с точки зрения исключений:

```
try (var reader = new InputStreamReader(
            new FileInputStream("input.txt"), "Cp1251");
        var writer = new OutputStreamWriter(
            new FileOutputStream("output.txt"), "Cp866")) {
    int c = 0;
    while ((c = reader.read()) >= 0) {
        writer.write(c);
    }
}
```

Ещё один полезный случай комбинирования потоков — это буферизация. BufferedReader читает данные сразу пачкой из “вложенного” в него потока и складывает в буфер. Настоятельно рекомендуется к использованию при чтении из файлов. Кроме того, BufferedReader позволяет читать по строкам. Пример:

```
static String readFirstLineFromFile(String path)
        throws IOException {
   try (var br = new BufferedReader(new FileReader(path))) {
        return br.readLine();
   }
}
```

Ну или вот пример чтения пользовательского ввода из консоли:

```
try (BufferedReader in = new BufferedReader(
    new InputStreamReader(System.in)))
{
   String input = in.readLine();
```

```
    ...
} catch (Exception e) {
    ...
}
```

6. Потоки для работы с данными в памяти

Механизм работы с потоками ввода вывода бывает полезен не только при собственно вводе-выводе, но и как средство удобно преобразовывать данные прямо в памяти. Также классы, симулирующие ввод-вывод на данных в памяти, бывают очень полезны, если у вас есть API, принимающее потоки, а вы не хотите что-то писать в файл только для того, чтобы тут же считать это что-то из файла. Вот основные такие классы.

- ByteArrayInputStream — чтение из массива байт. Это самый настоящий InputStream, использующий массив байт в памяти как источник данных.
- CharArrayReader — чтение из массива символов. Это самый настоящий Reader, делающий то же самое.
- StringReader — чтение из строки, если массив символов — это не удобно.
- ByteArrayOutputStream — запись в массив байт. Используем его как обычный OutputStream, а потом, как закончили, вызываем метод toByteArray() и получаем массив, который записался.
- CharArrayWriter — запись в массив символов. В общем-то, то же самое, но Writer. Для получения данных используются методы toString() или toCharArray().
- StringWriter – запись в StringBuffer. Для получения результатов используется либо toString(), либо toStringBuffer(). StringBuffer — это по сути потокобезопасный мутабельный String.

7. Высокоуровневая работа с вводом-выводом

Работать с байтами и символами такое себе удовольствие, если вам нужно сохранять/загружать сложные структуры данных, читать пользовательский ввод и т.д. Поэтому придуманы более высокоуровневые вещи, которые знают про элементарные типы и умеют их сохранять/загружать. На самом деле, в Java ещё есть встроенный механизм сериализации, через интерфейс Serializable и рефлексию, но про него предлагается почитать самостоятельно.

Первая высокоуровневая штука — это класс Scanner, который можно нацепить на поток ввода и заставить его возвращать значения элементарных типов, которые он сам будет парсить:

```
try (var s = new Scanner(
        new BufferedReader(new FileReader("test.txt")))) {
    while (s.hasNextDouble()) {
```

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

Ещё есть классы DataOutputStream и DataInputStream (реализации интерфейсов DataOutput и DataInput соответственно). DataOutputStream также реализует OutputStream, DataInputStream — InputStream, как не трудно догадаться. Они обеспечивают кроссплатформенную запись/чтение элементарных типов.

Вот основные методы, DataOutputStream:

- `writeBoolean()`, `writeByte()`, `writeChar()` и т.д. — запись примитивных типов
- `writeUTF()` — запись строки в кодировке UTF-8

И DataInputStream:

- `T readT()` — чтение примитивных типов
- `readUTF()` — чтение строки в кодировке UTF-8