Notatki - Streamy - BufferedInputStream i BufferedOutputStream

Spis treści

BufferedInputStream i BufferedOutputStream

Czyli jak poprawić wydajność poprzedniego rozwiązania. Wystarczy powyższe `FileInputStream` oraz `FileOutputStream` opakować w `BufferedStream`. W takim podejściu zamiast operować na pojedynczych bajtach, zaczniemy operować na tablicy bajtów.

Klasa StreamsExamples

```
public class StreamsExamples {

    public static void main(String[] args) throws IOException {
        File inputFile = new File("myInputFile.txt");
        File outputFile = new File("myOutputFile.txt");
        justCopyWithBuffer(inputFile, outputFile);
    }

    private static void justCopyWithBuffer(File source, File destination) throws IOException {
        try (
            InputStream input = new BufferedInputStream(new FileInputStream(source));
            OutputStream output = new BufferedOutputStream(new FileOutputStream(destination))
        ) {
            byte[] buffer = new byte[1024];
            int lengthRead = input.read(buffer);
            System.out.printf("Starting buffered reading file: [%s]%n", source);
            System.out.printf(
                "Read value: [%s], chars: [%s], length: [%s]%n",
                PrintingUtils.byteArrToStr(buffer), PrintingUtils.toCharString(buffer), lengthRead);

            while (lengthRead > 0) {
                System.out.printf(
                    "Writing buffered file value: [%s], char: [%s]%n",
                    PrintingUtils.byteArrToStr(buffer), PrintingUtils.toCharString(buffer));

                output.write(buffer, 0, lengthRead);
                output.flush();
                lengthRead = input.read(buffer);

                System.out.printf(
                    "Read buffered file value: [%s], char: [%s], length: [%s]%n",
                    PrintingUtils.byteArrToStr(buffer), PrintingUtils.toCharString(buffer), lengthRead);
            }
        }
    }
}
```

Klasa PrintingUtils

```
public class PrintingUtils {

    static String toCharString(byte[] input) {
        char[] charArray = new char[input.length];
        for (int i = 0; i < input.length; i++) {
            charArray[i] = (char) input[i];
        }
        return replaceNewLines(Arrays.toString(charArray));
    }

    static String byteArrToStr(byte[] buffer) {
        return replaceNewLines(Arrays.toString(buffer));
    }

    static String replaceNewLines(String input) {
        return input.replace("\n", "\\n").replace("\r", "\\r");
    }
}
```

Zamiast odczytywać bajt bo bajcie, używamy metody `read(byte[])`, która jako wynik zwraca nam ilość odczytanych bajtów z pliku i zapisanych w tablicy `byte[]`. Ta zwracana wartość jest o tyle istotna, że jeżeli metoda `read(byte[])` zwróci wartość `<= 0`, oznacza to koniec pliku. Pamiętać również należy o tym, że gdy dojdziemy do końca odczytywanego pliku, tablica `byte[]` tylko częściowo wypełni się danymi, które nas interesują. Czyli jeżeli tablica `byte[]` ma rozmiar `1024`, a w pliku zostało tylko `24` bajty do odczytu, to kolejne `1000` bajtów zostanie wypełnione wartościami z poprzednich odczytów. Po to właśnie mamy wartość `length` zwracaną przy odczycie, żebyśmy wiedzieli ile faktycznie odczytaliśmy podczas tego przebiegu przydatnych bajtów. Resztę możemy wtedy spokojnie pominąć.

Metoda `write()` jest o tyle ciekawa, że przyjmuje 3 argumenty:

- **faktyczną tablicę z bajtami do zapisu**,
- **offset** - ilość bajtów, którą możemy pominąć przy zapisie, najczęściej jednak wstawia się tu 0,
- **length** - ilość bajtów z tablicy, którą chcemy zapisać (znowu przydaje się tutaj w naszym przykładzie długość tablicy odczytanej, bo przy ostatnim zapisie `output.write(buffer, 0, lengthRead);` możemy określić ile bajtów z tej tablicy faktycznie zapiszemy, odrzucając te, które jak wspomniałem wcześniej, są uzupełnione wartościami z poprzednich odczytów)

I jeszcze raz, używanie klas z `Buffered` w nazwie, poprawia wydajność aplikacji ☺. W tym przykładzie dodano też metodę `flush()`, która wymusza faktyczny zapis do pliku wypełnimy w kodzie naszą tablicę z bajtami ponownie. O metodzie `flush()` wspominaliśmy na etapie omawiania teorii Streamów.

Jeszcze jedna dygresja. Mówiliśmy o poprawie wydajności przy przejściu z `FileInputStream` na `BufferedInputStream`. Można też próbować poprawić wydajność określając rozmiar tablicy `byte[]`. Najczęściej stosuje się rozmiar `1024`. Jednakże największą różnicę zobaczymy przesiadając się z `FileInputStream` na `BufferedInputStream`. Chociaż tutaj można też bawić się różnymi rozmiarami tablicy żeby zobaczyć jak wpłynie to na czas zapisu i odczytu.

W Javie 9 została wprowadzona metoda `readAllBytes()`, która pozwala na odczyt wszystkich pozostałych w Streamie bajtów. Należy jednak pamiętać, że dokumentacja tej metody wspomina, że jest ona dedykowana do małych plików, gdzie można wczytać całą zawartość pliku do tablicy z bajtami "na raz".

Nie jest ona przeznaczona do odczytu dużych plików. Do prawidłowego stosowania tej metody polecam zapoznać się z dokumentacją.

Po co w sumie są BufferedStreamy?

Systemy plików są tworzone w taki sposób, że są wydajne, gdy staramy się dostać do plików na dysku w sposób sekwencyjny. Im więcej bajtów staramy się odczytać w sekwencji jednocześnie, tym mniej operacji odczytu musimy wykonać. W ten sposób optymalizujemy naszą aplikację, bo mniej musi ona wykonać faktycznych odwołań do dysku w celu wykonania jakiejś operacji na pliku (np. odczyt). Przykładowo odczyt 8 bajtów w sekwencji będzie szybszy niż odczyt 8 bajtów rozmieszczonych losowo po dysku komputera.