9.6 フィルタの活用

9.6.1 フィルタとは

ストリームの概念を導入するもう１つのメリットは「フィルタ」が使えるようになること。これはとても強力な機能だよ。

フィルタ？…うちの台所についてる浄水器のフィルタも、すごく強力ですよ！

実は湊くんが挙げた例は当たらずとも遠からず、です。浄水器のフィルタは「上流から流れてきた水道水を、よりきれいな水に変換して下流に流す装置」です。Javaの世界でもストリームの途中にさまざまな変換処理をする部品（のインスタンス）を挟み込むことが可能であり、それらは**フィルタ**（filter）と呼ばれています。

浄水器のフィルタは「水道水をきれいな水にする」だけの役割ですが、Javaのフィルタは流れるデータに対して多種多様な変換をすることができます。たとえば、本来はとても複雑な「データを暗号化してファイルに書き込む処理」も、CipherOutputStreamフィルタを１つ挟むだけで実現可能です（図9-13）。

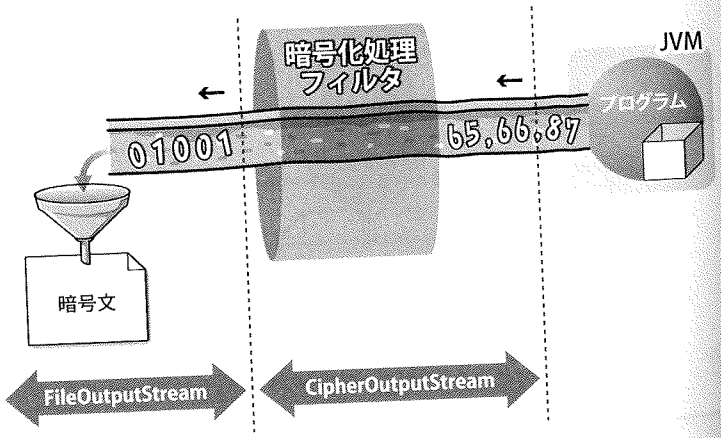

図9-13　通るデータを暗号文に変換するフィルタを用いた例

9.6.2 フィルタの特徴

JavaAPIではたくさんのフィルタが標準で提供されています（オリジナルのフィルタも開発可能です）。それらのフィルタは、必ず次の特徴を備えています。

特徴①　Filter～クラスを継承している

フィルタは必ず、FilterReader、FilterWriter、FilterInputStream、FilterOutputStreamのいずれかのクラスを継承しています。たとえば、図9-13の暗号化フィルタjavax.crypto.CipherOutputStreamは「JVMからデータを出力するバイトストリームに挟むフィルタ」なので、FilterOutputStreamを継承しています。

特徴②　単独で存在できず、他に接続する形で生成する

フィルタはあくまでストリームに付加的に接続するための道具ですので、通常はフィルタ単独で生成（new）することはできません。次のリストのように**すでに存在するストリームを接続先としてコンストラクタで指定して生成**します。

なお、コード内に登場する変数algoには、暗号化方式等の指定が格納されているものとします。

```
// STEP1: まず通常のファイル出力ストリームfosを生成
FileOutputStream fos = new FileOutputStream("data.txt");
// STEP2: このストリームを下流に持つ暗号化ストリームcosを生成し接続
CipherOutputStream cos = new CipherOutputStream(fos, algo);
// STEP3: cosに書き込めば、暗号化された上でファイルに流れていく
cos.write(65);
```

特徴③　フィルタを複数連結することもできる

次のリストのようにストリームにすでにフィルタが接続されている状態で、さらに別のフィルタを接続して利用することも可能です（次ページの図9-14）。

```
// STEP1: まず通常のファイル出力ストリームfosを生成
FileOutputStream fos = new FileOutputStream("data.txt");
// STEP2: このストリームを下流に持つ暗号化ストリームcosを接続
CipherOutputStream cos = new CipherOutputStream(fos, algo);
// STEP3: さらに文字バイト変換をするストリームoswを接続
OutputStreamWriter osw = new OutputStreamWriter(cos);
// STEP4: oswに文字を書き込めば、バイト変換&暗号化されファイルに流れていく
osw.write("AB");
```

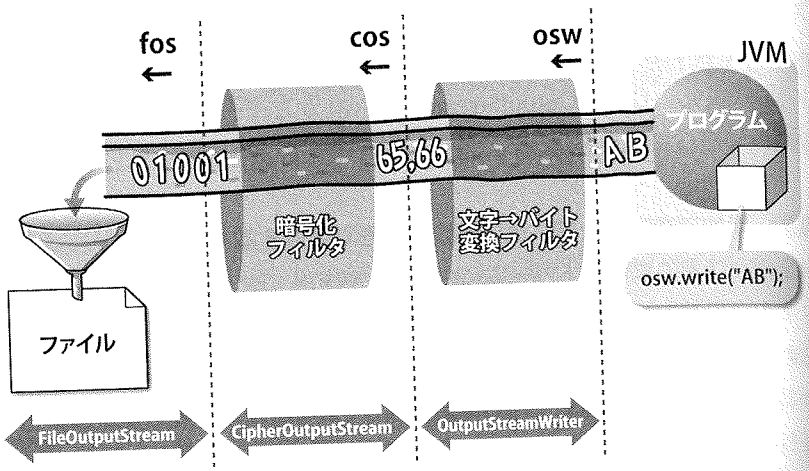

図 9-14　「多段フィルタ」の状態で利用する例

osw.close() を実行すれば、接続されている cos や fos の close() も連鎖的に実行されるよ。flush() も同様だ。

9.6.3　バッファリングフィルタ

JavaAPI では、たくさんの種類のフィルタが標準で提供されています。中でも利用頻度が高いのが java.io パッケージに準備された次の 4 つのフィルタです。

文字情報用　：BufferedReader、BufferedWriter
バイト情報用：BufferedInputStream、BufferedOutputStream

これら 4 つのクラスは**バッファリングフィルタ**（buffering filter）と総称されていますが、いずれも流れるデータの変換は行いません。つまり、このフィルタの上流と下流で流れるデータはまったく同一のものです。

挟み込んでも何の仕事もしてくれないフィルタなんて、存在価値がないんじゃないですか？

これらは「変換」をしないだけで、別の仕事をしてくれるんだよ。

バッファリングフィルタは、流れるデータを別の内容に変換することはしませんが、上流から少しずつ流れてくるデータを溜め込み、まとまった量になったところで一気に下流へ流す仕事をしてくれます（図 9-15）。

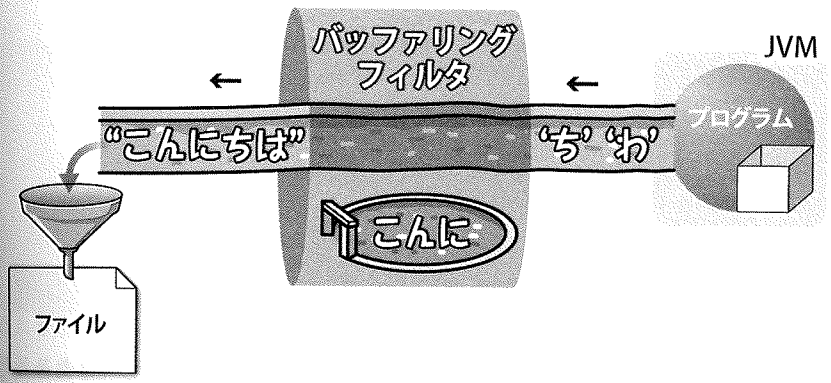

図 9-15　BufferedWriter の利用イメージ

データを溜めて一定量ずつまとめて処理する主なメリットは次の 2 点です。

メリット 1　処理性能の向上

ファイルを保存するために何気なく利用しているハードディスクは、メモリに比べて大容量ではあるものの読み書きの速度が遅いという欠点があります。私たちがファイルの読み書きを要求するたびに、ハードディスク内部のヘッドという

機械が動いて回転するディスク上のデータを読み書きしなければならないからです。よって、少量のデータを何度も書き込む要求をするより、データをひとまとめにして1回の書き込み要求をするほうが圧倒的に速く処理が終了します。

ファイル操作はバッファリングする

ファイルを読み書きするときは、通常はバッファリングフィルタを併用する。バッファリングなしでファイル操作をすることはほとんどない。

メリット2　まとまった単位でデータを読める

FileReaderはread()メソッドを使い1文字ずつ読み込むことが基本だと解説しました。しかしBufferedReaderを挟むとreadLine()メソッドが利用可能になります。このメソッドは改行までの1行分のデータをString型で返してくれる便利なもので、1行ずつファイルを読み込むプログラムを簡単に開発できます。

```
FileReader fr = new FileReader("rpgsave.dat");
BufferedReader br = new BufferedReader(fr);
String line = null;
while((line = br.readLine()) != null) {
  /* ここでlineの内容を利用した処理 */
}
```

このようなことが可能なのは、BufferedReaderがファイルのデータをある程度先読みして内部に溜め込んでいるからです。BufferedReaderは溜め込んだデータの中から次の改行部分までを取り出して返してくれます。

9.7 ファイルシステムの操作

9.7.1 ファイル自体を操作する

それでは章の最後に、ファイルの中身ではなくファイルの存在自体を操作するAPIを紹介しておこう。

これまで紹介してきたストリームクラスを用いれば、私たちは自由にファイルの中身を読み書きすることができます。しかし、ファイル自体を削除したり、ファイルサイズを取得するなど、ファイル自体に関して操作を行うためにはまた別のAPIを用いる必要があります。

ファイル自体の操作に古くから用いられてきたのは、**java.io.Fileクラス**です。Fileクラスは、次のようにして、ある特定のファイルやフォルダを指し示すインスタンスを生み出すことができます。

```
File file = new File("c:¥¥rpgdata.txt");
```

Fileインスタンスは、FileInputStreamやFileOutputStreamのコンストラクタ引数として、String型の代わりに利用することができます。さらに、表9-5にあるようなメソッドを用いて、指し示しているファイルやフォルダを操作することができます。

表9-5　java.io.Fileクラスが備える代表的なメソッド

操作	メソッドのシグニチャ
削除する	public boolean delete()
名前を変更する	public boolean renameTo(File dest)
存在するか確認する	public boolean exists()
ファイルであるか確認する	public boolean isFile()
フォルダであるか確認する	public boolean isDirectory()

ファイルサイズを取得する	public long length()
フォルダ内のファイルの一覧を取得する	public File[] listFiles()

9.7.2 File クラスの懸念点と解決策

ファイルの削除はできたけど…。うーん、コピーのための命令はないのかなぁ。

FileReader で 1 文字ずつ読み込みつつ、FileWriter で書き込んでいったらいいんじゃない？　とっても面倒だけど…。

java.io.File は広く利用されてきたクラスでしたが、次のような懸念点が指摘されていました。

懸念点①　備えている操作用の命令が少なく、中途半端にしか揃っていない。
懸念点②　ファイルやフォルダを「指し示す」という責務と、「各種の操作をする」という責務を抱えており、役割がわかりにくい。

そこで、この懸念点を解決するために Java7 から java.nio.file.Path インタフェースと java.nio.file.Files クラスが導入されました。

Path は、従来から用いられてきた File クラスの後継に当たるもので、特定のフォルダやファイルを指し示すためのものです。Path のインスタンスを取得するためには、new ではなく、java.nio.file.Paths クラスの get() メソッドを用います。また、すでに File インスタンスが存在する場合、その toPath() メソッドを呼ぶことで Path インスタンスに変換することも可能です。

```
Path p1 = Paths.get("c:¥¥rpgdata.txt");
Path p2 = file.toPath();
```


Path インスタンスは、あくまでファイルやフォルダを指し示す役割しか持ちません。実際に指し示す先を削除したりコピーしたりするための static メソッドは、Files クラスに豊富に準備されています (表 9-6)。

表 9-6　java.nio.file.Files クラスが備える代表的なメソッド

操作	メソッドのシグニチャ
コピーする	public static long copy(Path from, Path to)
移動 (または改名) する	public static long move(Path from, Path to)
削除する	public static void delete(Path path)
中身をすべて読み込む	public static byte[] readAllBytes(Path path)
全行を読み込む	public static List<String> readAllLines(Path path)
存在するか確認する	public static boolean exists(Path path)
フォルダであるか確認する	public static boolean isDirectory(Path path)
ファイルサイズを取得する	public static long size(Path path)

Files の API リファレンスを見たら、他にもたくさん便利そうなメソッドがありました！

どれも引数に Path インスタンスを渡して利用するのね。

9.8 この章のまとめ

ストリームの概念

・JVM がデータをやりとりする伝送路を表すストリームという概念がある。
・文字が流れる文字ストリーム、バイトが流れるバイトストリームがある。
・ファイルのほか、ネットワークやキーボードなどもストリームを用いてデータをやりとりできる。

ファイルの基本的な操作

・ファイルを前から順に読み書きする方法をシーケンシャルアクセスという。
・ファイルの読み書きには、次のようなクラスを利用する。

	読み込み	書き込み
テキストファイル	FileReader	FileWriter
バイナリファイル	FileInputStream	FileOutputStream

・ファイルは必ず finally ブロックで閉じなければならない。
・ファイルの読み書きには通常バッファリングフィルタを併用する。

フィルタ

・フィルタはほかのストリームやフィルタに連結して利用できる。
・Java の API では多様なフィルタが提供されている。

ファイルシステムの操作

・java.io.File を用いて、特定のファイルやフォルダを指し示すことができる。
・java.io.File では、ファイルの削除や情報取得など簡単な操作も行える。
・Java7 以降では、java.nio.Path と java.nio.Files を用いて、より高度な操作を簡単に行うことができる。

9.9 練習問題

練習 9-1

FileInputStream クラスと FileOutputStream クラスを使って、ファイルをコピーするプログラムを作成してください。コピー元ファイル名とコピー先ファイル名は Java プログラムの起動パラメータとして指定するものとし、バッファリングやエラー処理、例外処理は不要とします。

練習 9-2

練習 9-1 で作ったプログラムを下記に従って改造し、ファイル圧縮ソフトを作ってください。

・ファイルを書き込む際、java.util.zip.GZIPOutputStream を使って圧縮する。
・ファイルの書き込みには必ずバッファリングを行う。
・例外処理を正しく行う。

9.10 練習問題の解答

練習 9-1 の解答

Main.java

```
import java.io.*;

public class Main {
  public static void main(String[] args) throws Exception {
    String inFile = args[0];
    String outFile = args[1];
    FileInputStream fis = new FileInputStream(inFile);
    FileOutputStream fos = new FileOutputStream(outFile);
    int i = fis.read();
    while(i != -1) {
      fos.write(i); i = fis.read();
    }
    fos.flush();
    fos.close();
    fis.close();
  }
}
```

9.7.2 で紹介したクラスを使うと、main メソッドは次の 1 行で記述できます（要 import）。

```
Files.copy(Paths.get(args[0]), Paths.get(args[1]));
```

練習 9-2 の解答

Main.java

```
import java.io.*;
import java.util.zip.GZIPOutputStream;

public class Main {
  public static void main(String[] args) {
    String inFile = args[0];
    String outFile = args[1];
    FileInputStream fis = null;
    GZIPOutputStream gzos = null;
    try {
      fis = new FileInputStream(inFile);
      FileOutputStream fos = new FileOutputStream(outFile);
      BufferedOutputStream bos = new BufferedOutputStream(fos);
      gzos = new GZIPOutputStream(bos);
      int i = fis.read();
      while(i != -1) {
        gzos.write(i);
        i = fis.read();
      }
      gzos.flush();
      gzos.close();
      fis.close();
    } catch(IOException e) {
      System.err.println("ファイル操作に失敗しました");
      try {
        if(fis != null) {
          fis.close();
        }
        if(gzos != null) {
          gzos.close();
        }
      } catch(IOException ee) { }    // 何もしない
    }
  }
}
```