ストリームを反転させる: 第1回

出力ストリームからの読み取り

Merlin Hughes (merlin@merlin.org)
Cryptographer
Baltimore Technologies

2002年 7月 01日

一般にJava I/Oフレームワークはきわめて多くのことに使用できます。同一のフレームワークが、ファイル・アクセス、ネットワーク・アクセス、文字変換、圧縮、暗号化などをサポートします。しかし、柔軟性が十分であるとは言えない場合があります。たとえば、圧縮ストリームによってデータを圧縮形式で書き込むことはできますが、データを圧縮形式で読み取ることはできません。同様に、アプリケーションがデータを読み取る必要が生じる場合を考慮せずに、データを書き出すよう設計されたサード・パーティーのモジュールもあります。2回にわたる連載の第1回である今回の記事では、Java暗号研究者である著者のMerlin Hughesが、(出力ストリームへのデータ書込のみをサポートする) ソースからアプリケーションが効率的にデータを読み取れるようにするフレームワークについて説明します。

Javaプラットフォームは、初期のブラウザー・ベースのアプレットや単純なアプリケーション以降、大幅に拡張されています。現在、複数のプラットフォームとプロファイル、および数十もの新しいAPIがあり、さらに実に数百のAPIが開発中です。ますます複雑になっているにもかかわらず、Java言語は依然として日常のプログラミング作業における優れたツールです。日々のプログラミング上の問題にはまり込んでしまった場合に、その場から一歩退いてみると、以前から何度も出くわしている難問に対する名案がパッと閃くこともあるものです。

つい先日、私はネットワーク接続からデータを読み取る際に、そのデータを圧縮したいと考えました (UDPソケットの下で、TCPデータを圧縮形式で中継していたのです)。圧縮がバージョン1.1以降Javaプラットフォームによってサポートされていることを思い出したので、解決策が見つかることを期待して、`java.util.zip` パッケージを大至急手にとりました。しかし、解決策は見当たりませんでした。そのクラスは、データを読み取る際に解凍し、書き込む際に圧縮するという通常のケースに基づいて設計されており、私の望む「その逆のケース」ではなかったのです。I/Oクラスをバイパスすることもできましたが、私はストリーム方式に基づいた解決案を望んでおり、コンプレッサーを直接使用するという不本意な手段は取りたくありませんでした。

ほんの少し前にも別の状況でまったく同じ問題に出くわしていたことを思い出しました。私はベース64トランスコーディング・ライブラリーを持っていますが、これは圧縮パッケージと同様、ストリームから読み取られるデータのデコードと、ストリームに書き込まれるデータのエン

コードをサポートします。しかし私には、データをストリームから読み取る際にエンコードするライブラリーが必要だったのです。

この問題の解決に着手したとき、私はさらに別の状況でもこの問題に出くわしていたことに気付きました。XML文書をシリアル化する場合は、通常、文書全体にわたって、ノードをストリームに書き出す作業を繰り返します。しかし私は、そのサブセットを解析し直して新しい文書にするために、シリアル化された形式の文書を読み取らなければならない状況にありました。

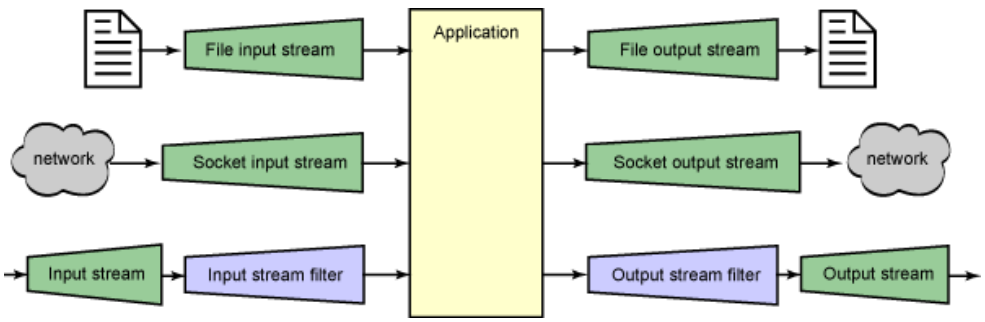
一歩退いて考えてみると、これらの個々の出来事はどうやら一般的な問題を表しているようです。つまり、出力ストリームにデータを逐次に書き込むデータ・ソースを考えた場合、さらにデータが必要になった際にはいつでも、透過的にデータ・ソースを呼び出すことで、これらのデータを読み取ることを可能にする入力ストリームが必要になるということです。

今回の記事では、この問題に対して考えうる3つのソリューションについて検討し、それらのソリューションの中で最善のものを実装する新しいフレームワークを決めることにします。そして、上述のそれぞれの問題に対してそのフレームワークを実際に使用してみます。パフォーマンスの問題に関しては簡単に触れますが、その説明の大半は次回の記事で取り上げることにします。

I/Oストリームの基本

まず、図1に示されているJavaプラットフォームの基本的なストリーム・クラスについて簡単におさらいしてみます。`OutputStream` は、データを書き込むことのできるストリームを表しています。一般的にこのストリームは、ファイルやネットワーク接続などのデバイスに直接接続されるか、あるいは別の出力ストリーム (このケースではフィルターと呼ばれる) に接続されます。一般に出力ストリーム・フィルターは、書き込まれるデータを変換した後で、接続されているストリームに変換結果のデータを書き込みます。`InputStream` は、それからデータを読み取ることができるデータ・ストリームを表しています。このストリームもまた、デバイスに直接接続されるか、あるいは別のストリームに接続されます。入力ストリーム・フィルターは、接続されているストリームからデータを読み取り、このデータを変換し、この変換したデータを入力ストリーム・フィルターから読み取れるようにします。

図1. I/Oストリームの基本



最初の問題に関しては、`GZIPOutputStream` クラスは、書き込まれるデータを圧縮し、その圧縮したデータを接続されているストリームに書き込む出力ストリーム・フィルターです。私は、ストリームからデータを読み取り、それらを圧縮し、その結果を読み取ることができる入力ストリーム・フィルターを必要としました。

Javaプラットフォーム、バージョン1.4は、新しいI/Oフレームワークである`java.nio`を導入しています。しかし、このフレームワークの大半は、オペレーティング・システムのI/Oリソースを効率的に利用するということに関するものであり、従来の`java.io`クラスの一部と類似のものを提供し、入力と出力の両方をサポートするという2つの目的に使用されるリソースを表すことはできるものの、標準のストリーム・クラスに完全に取って代わるものではなく、私が解決しなければならなかった問題に直接対処できるものでもありません。

力ずくのソリューション

問題に対するエンジニアリング・ソリューションを探し始める前に、洗練度と効率の点から標準のJava APIクラスに基づいたソリューションについて検討してみました。

この問題に対する力ずくのソリューションとは、単に入力ソースからすべてのデータを読み取って、トランスフォーマー (すなわち圧縮ストリーム、エンコード・ストリーム、またはXMLシリアライザー) を介してデータをメモリー・バッファーに押し込むことです。こうすれば、ストリームを開いてこのメモリー・バッファーから読み取ることが可能になり、問題が解決したことになるでしょう。

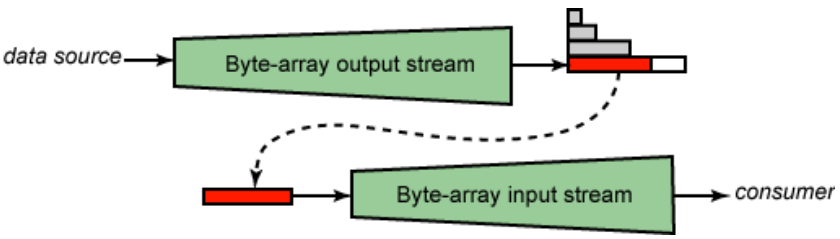
まず、汎用I/Oメソッドが必要です。リスト1のメソッドは、小さなバッファーを使用して`InputStream`からのデータをすべて`OutputStream`にコピーします。入力の最後に到達する (`read()`関数が0未満の値を返す) と、いずれのストリームも閉じずにメソッドは戻ります。

リスト1. 汎用I/Oメソッド

```
public static void io (InputStream in, OutputStream out) throws IOException {
  byte[] buffer = new byte[8192];
  int amount;
  while ((amount = in.read (buffer)) >= 0)
    out.write (buffer, 0, amount);
}
```

リスト2は、入力ストリームの圧縮形式を読み取ることができる力ずくのソリューションを示しています。メモリー・バッファーに書き込む`GZIPOutputStream`を開きます (`ByteArrayOutputStream`を使用)。次に、入力ストリームを圧縮ストリームにコピーします。これによって、メモリー・バッファーには圧縮データが書き込まれます。次に、図2に示すように、入力ストリームからの読み取りを可能にする`ByteArrayInputStream`を返します。

図2. 力ずくのソリューション



リスト2. 力ずくのソリューション

```
public static InputStream bruteForceCompress (InputStream in) throws IOException {
  ByteArrayOutputStream sink = new ByteArrayOutputStream ():
  OutputStream out = new GZIPOutputStream (sink);
  io (in, out);
  out.close ();
  byte[] buffer = sink.toByteArray ();
  return new ByteArrayInputStream (buffer);
}
```

このソリューションの明らかな欠点は、圧縮文書全体をメモリーに保存することです。文書が大きい場合、このアプローチは必要以上にシステム・リソースを浪費します。ストリームを使用する最大の特徴の1つは、使用しているシステムのメモリーよりも大きなデータを操作できるという点です。つまり、すべてのデータをメモリーに保持するのではなく、データを読み取りながら処理したり、あるいは書き込みながら生成したりすることができるということです。

効率の点から、バッファー間でのデータのコピーについてさらに詳しく見ることにしましょう。

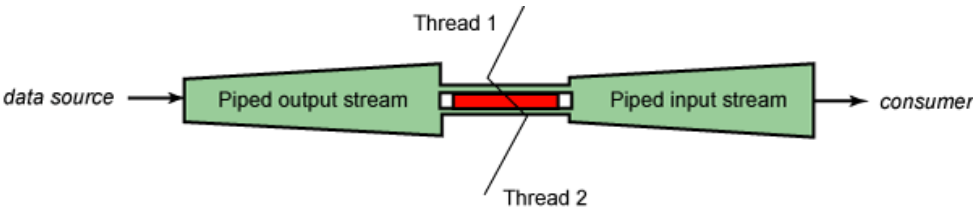
データは、`io()` メソッドによって入力ソースから1つのバッファーに読み込まれます。次に、そのバッファーから`ByteArrayOutputStream` 内のバッファーに書き込まれます (今の時点で、敢えて言及していない圧縮機能を介して)。しかし`ByteArrayOutputStream` クラスは拡張する内部バッファーを使用するため、バッファーがいっぱいになると、2倍のサイズの新たなバッファーが常に割り当てられ、既存のデータがそれにコピーされます。平均すると、各バイトはこのプロセスによって二度コピーされます。(この計算は簡単です。`ByteArrayOutputStream` に入ると、データは平均して二度コピーされます。すべてのデータが少なくとも一度コピーされ、半分が少なくとも二度、四分の一が少なくとも三度コピーされる、といった具合です。) 次に、データはそのバッファーから`ByteArrayInputStream` 用の新しいバッファーにコピーされます。これで、このデータはアプリケーションによって読み取れるようになります。合計すると、データはこのソリューションによって4つのバッファーを介して書き込まれることになります。これは、他の技法の効率を評価する際に役立つ基準です。

パイプ・ストリーム・ソリューション

パイプ・ストリーム (`PipedOutputStream` と`PipedInputStream`) は、Java仮想マシンのスレッド間におけるストリーム・ベースの接続を提供します。1つのスレッドによって`PipedOutputStream` に書き込まれるデータを、別のスレッドによって関連する`PipedInputStream` から並行して読み取ることができます。

そういうわけで、これらのクラスは私の問題に対するソリューションを提示してくれます。リスト3は、1つのスレッドを採用してデータを入力ストリームから`GZIPOutputStream` を介して`PipedOutputStream` にコピーするコードを示しています。そして図3に示すように、関連する`PipedInputStream` が別のスレッドからの圧縮データへの読み取りアクセスを提供します。

図3. パイプ・ストリーム・ソリューション



リスト3. パイプ・ストリーム・ソリューション

```
private static InputStream pipedCompress (final InputStream in) throws IOException {
  PipedInputStream source = new PipedInputStream ();
  final OutputStream out = new GZIPOutputStream (new PipedOutputStream (source));
  new Thread () {
    public void run () {
      try {
        Streams.io (in, out);
        out.close ();
      } catch (IOException ex) {
        ex.printStackTrace ();
      }
    }
  }.start ();
  return source;
}
```

理論上、これは優れた技法かもしれません。スレッド (圧縮を実行するものと、結果のデータを処理するもの) を採用することによって、アプリケーションはハードウェアのSMP (対称多重処理) またはSMT (対称マルチスレッド化) から恩恵を受けることができます。しかも、このソリューションには2つのバッファー書き込みのみが関与します。まずI/Oループがデータを入力ストリームからバッファーに読み込み、圧縮ストリームを介して`PipedOutputStream` に書き込みます。次に、出力ストリームがデータを内部バッファーに格納します。内部バッファーは、アプリケーションによる読み取り用として`PipedInputStream` と直接共用されます。さらに、データは固定バッファーを介して流し込まれるので、メモリーにすべてを読み込む必要はありません。その代わり、小さな作業セットだけが一定時間バッファーに入れられます。

同期の問題

この記事で示されているコードはどれも同期していません。つまり、2つのスレッドがこれらのクラスの1つの共用インスタンスに並行してアクセスするのは安全ではありません。

同期はNIOフレームワークやCollections APIなどのライブラリーで受け入れられるようになった手段ですが、依然としてアプリケーションにとっては負担となります。1つのオブジェクトに並行してアクセスしようとする場合、アプリケーションはアクセスを同期させるのに必要なステップを実行しなければなりません。

最近のJVMではスレッド・セーフティー・メカニズムのパフォーマンスが大きく向上しましたが、同期は依然として高価なオペレーションです。I/Oの場合、単一のストリームへの並行アクセスはほぼ必ずエラーとなります。結果のデータ・ストリームの順序が予想が付かないものになり、これで満足できるケースは稀です。そういうわけで、これらのクラスを同期させることは、何の明確な恩恵も望めない不要な費用を強いることになるでしょう。

マルチスレッドの考慮事項に関してはこの連載の第2回で詳しく説明します。今回は、ここで説明するストリームへの並行アクセスが、予想も付かないエラーをもたらすという点にのみ注意してください。

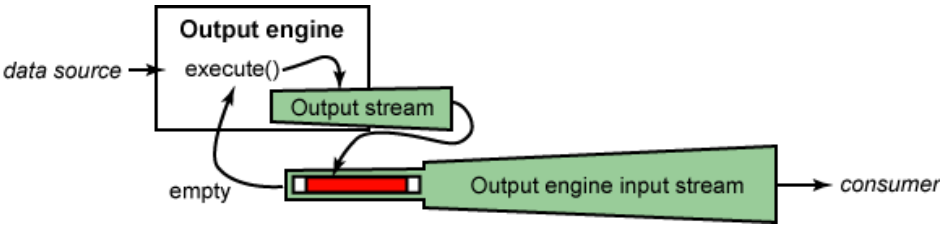
しかし実際には、パフォーマンスはひどいものです。パイプ・ストリームは同期を利用する必要があるので、2つのスレッド間で激しい競合が起きるでしょう。内部バッファーが小さすぎて、大量のデータを効果的に処理できないか、あるいはロック競合を隠すことができません。さらにバッファーを常に共用することによって、1つのSMPマシンでワークロードを分担するための多くの単純なキャッシュ方法が無効になるでしょう。最後に、スレッドの使用は例外処理を非常に困難にします。発生する可能性のある`IOException`を読み取り側による処理のためにパイプに押し込む方法がないからです。全体的に見て、このソリューションは重すぎて効果的ではありません。

エンジニアリング・ソリューション

では、この問題に対する代替ソリューションとなるエンジニアリング・ソリューションについて見てみましょう。このソリューションは、特にこの種の問題を解決するために設計されたフレームワークを提供します。このフレームワークは、データを逐次に`OutputStream`に書き込むソースから作成されるデータへの`InputStream`アクセスを提供するものです。データが逐次に書き込まれるという事実が重要です。ソースが単一のアトミック・オペレーションですべてのデータを`OutputStream`に書き込み、かつスレッドが使用されない場合は、基本的に力ずくの技法に戻ります。しかし、データを逐次に書き込むようソースに要求することができれば、力ずくのソリューションとパイプ・ストリーム・ソリューションの間でうまくバランスをとることができます。このソリューションは、メモリーには常に少量のデータしか保持しないというパイプ・ストリーム・ソリューションの利点を提供しつつ、スレッドを回避するという力ずくのソリューションの利点も提供します。

図4は完成したソリューションを示しています。

図4. エンジニアリング・ソリューション



出力エンジン

リスト4は、データ・ソースを記述する`OutputEngine`というインターフェースを提供します。すでに述べたように、これらのソースはデータを逐次に出力ストリームに書き込みます。

リスト4. 出力エンジン

```
package org.merlin.io;
import java.io.*;
/**
 * An incremental data source that writes data to an OutputStream.
 *
 * @author Copyright (c) 2002 Merlin Hughes <merlin@merlin.org>
 *
 * This program is free software; you can redistribute
 * it and/or modify it under the terms of the GNU
 * General Public License as published by the Free
 * Software Foundation; either version 2
 * of the License, or (at your option) any later version.
 */
public interface OutputEngine {
  public void initialize (OutputStream out) throws IOException;
  public void execute () throws IOException;
  public void finish () throws IOException;
}
```

`initialize()` メソッドは、エンジンに対し、データの書き込み先となるストリームを提示します。`execute()` メソッドが繰り返し呼び出され、このストリームにデータを書き込みます。データがなくなると、エンジンはストリームを閉じるはずです。最後に、エンジンがシャットダウンする際に`finish()` が呼び出されます。これは、エンジンがその出力ストリームを閉じる前に起こる場合もあれば、閉じた後に起こる場合もあります。

I/Oストリーム・エンジン

このような作業を始めるきっかけとなった問題に対処する出力エンジンは、入力ストリームから出力ストリーム・フィルターを介してターゲットの出力ストリームにデータをコピーします。これは一度に1つずつバッファーに対する読み書きを行うことができるので、逐次に進めるという性質を満たすことができます。

リスト5から10までのコードは、そのようなエンジンを実装しています。これは、入力ストリームおよび出力ストリーム・ファクトリーから構築されます。リスト11は、フィルター出力ストリームを生成するファクトリーです。たとえばこれは、ターゲットの出力ストリームにラップされた`GZIPOutputStream` を返すことができます。

リスト5. I/Oストリーム・エンジン

```
package org.merlin.io;
import java.io.*;
/**
 * An output engine that copies data from an InputStream through
 * a FilterOutputStream to the target OutputStream.
 *
 * @author Copyright (c) 2002 Merlin Hughes <merlin@merlin.org>
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version 2
 * of the License, or (at your option) any later version.
 */
public class IOStreamEngine implements OutputEngine {
  private static final int DEFAULT_BUFFER_SIZE = 8192;
  private InputStream in;
  private OutputStreamFactory factory;
  private byte[] buffer;
  private OutputStream out;
```

このクラスのコンストラクターは、さまざまな変数と、データの転送に使用されるバッファーを単に初期化します。

リスト6. コンストラクター

```
public IOStreamEngine (InputStream in, OutputStreamFactory factory) {
    this (in, factory, DEFAULT_BUFFER_SIZE);
  }
  public IOStreamEngine (InputStream in, OutputStreamFactory factory, int bufferSize) {
    this.in = in;
    this.factory = factory;
    buffer = new byte[bufferSize];
  }
```

`initialize()` メソッドでは、このエンジンは、提供された`OutputStream` をラップするようファクトリーに要求します。このファクトリーは通常、フィルターを`OutputStream` に接続します。

リスト7. initialize() メソッド

```
  public void initialize (OutputStream out) throws IOException {
    if (this.out != null) {
      throw new IOException ("Already initialised");
    } else {
      this.out = factory.getOutputStream (out);
    }
  }
```

`execute()` メソッドでは、エンジンは`InputStream` からバッファーのデータを読み取り、ラップされた`OutputStream` にそれを書き込みます。あるいは入力が終了している場合は、`OutputStream` を閉じます。

リスト8. execute() メソッド

```
  public void execute () throws IOException {
    if (out == null) {
      throw new IOException ("Not yet initialised");
    } else {
      int amount = in.read (buffer);
      if (amount < 0) {
        out.close ();
      } else {
        out.write (buffer, 0, amount);
      }
    }
  }
```

最後に、エンジンがシャットダウンするとき、エンジンはその`InputStream` を閉じます。

リスト9. InputStreamの終了

```
  public void finish () throws IOException {
    in.close ();
  }
```

下のリスト10に示す内部`OutputStreamFactory` インターフェースは、フィルター`OutputStream` を返すことができるクラスを示しています。

リスト10. 内部出力ストリーム・ファクトリー・インターフェース

```
public static interface OutputStreamFactory {
    public OutputStream getOutputStream (OutputStream out) throws IOException;
  }
}
```

リスト11は、`GZIPOutputStream` で提供されるストリームをラップするファクトリーの例を示しています。

リスト11. GZIP出力ストリーム・ファクトリー

```
public class GZIPOutputStreamFactory implements IOStreamEngine.OutputStreamFactory {
  public OutputStream getOutputStream (OutputStream out) throws IOException {
    return new GZIPOutputStream (out);
  }
}
```

出力ストリーム・ファクトリー・フレームワークを有するこのI/Oストリーム・エンジンは、出力ストリーム・フィルター操作のニーズの大半をサポートするだけの十分な汎用性を備えています。

出力エンジン入力ストリーム

最後に、このソリューションを完成させるには、さらにもう少しコードが必要です。リスト12から16までのコードは、出力エンジンによって書き込まれるデータを読み取る入力ストリームを表しています。実際には、このコードには2つの部分があります。メインのクラスは、内部バッファーからデータを読み取る入力ストリームです。リスト17に示すように、これと密結合されているのが出力ストリームです。これは、出力エンジンによって書き込まれるデータで内部読み取りバッファーを満たします。

メインの入力ストリーム・クラスは、出力エンジンを自分の内部出力ストリームで初期化します。そしてバッファーが空になった場合は、いつでも自動的にエンジンを実行して、新たなデータを受け取ることができます。出力エンジンはその出力ストリームにデータを書き込み、これが入力ストリームの内部バッファーを再び満たします。これによって、データを処理するアプリケーションはデータを効率よく読み取ることができます。

リスト12. 出力エンジン入力ストリーム

```
package org.merlin.io;
import java.io.*;
/**
 * An input stream that reads data from an OutputEngine.
 *
 * @author Copyright (c) 2002 Merlin Hughes <merlin@merlin.org>
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version 2
 * of the License, or (at your option) any later version.
 */
public class OutputEngineInputStream extends InputStream {
  private static final int DEFAULT_INITIAL_BUFFER_SIZE = 8192;
  private OutputEngine engine;
  private byte[] buffer;
  private int index, limit, capacity;
  private boolean closed, eof;
```

この入力ストリームのコンストラクターは、データの読み取り元である出力エンジンと、オプションのバッファー・サイズを受け取ります。まずストリームはストリーム自体を初期化し、次に出力エンジンを初期化します。

リスト13. コンストラクター

```
public OutputEngineInputStream (OutputEngine engine) throws IOException {
  this (engine, DEFAULT_INITIAL_BUFFER_SIZE);
}
public OutputEngineInputStream (OutputEngine engine, int initialBufferSize) throws IOException {
  this.engine = engine;
  capacity = initialBufferSize;
  buffer = new byte[capacity];
  engine.initialize (new OutputStreamImpl ());
}
```

コードの主な読み取り部分は、比較的簡単なバイト配列ベースの入力ストリームであり、`ByteArrayInputStream` クラスとほぼ同じです。しかしデータが要求されてこのストリームが空になると、それは常に出力エンジンの`execute()` メソッドを呼び出し、読み取りバッファーを再び満たします。このような新しいデータは、その後呼び出し側に返すことができます。したがってこのクラスは、出力エンジンによって書き込まれるデータを最後まで繰り返し読み取った後で、`eof` フラグが設定され、ファイルの最後に達したということをこのストリームが返します。

リスト14. データの読み取り

```
private byte[] one = new byte[1];
  public int read () throws IOException {
    int amount = read (one, 0, 1);
    return (amount < 0) ? -1 : one[0] & 0xff;
  }
  public int read (byte data[], int offset, int length) throws IOException {
    if (data == null) {
      throw new NullPointerException ();
    } else if ((offset < 0) || (length < 0) || (offset + length > data.length)) {
      throw new IndexOutOfBoundsException ();
    } else if (closed) {
      throw new IOException ("Stream closed");
    } else {
      while (index >= limit) {
        if (eof)
          return -1;
        engine.execute ();
      }
      if (limit - index < length)
        length = limit - index;
      System.arraycopy (buffer, index, data, offset, length);
      index += length;
      return length;
    }
  }
  public long skip (long amount) throws IOException {
    if (closed) {
      throw new IOException ("Stream closed");
    } else if (amount <= 0) {
      return 0;
    } else {
      while (index >= limit) {
        if (eof)
```

```
          return 0;
        engine.execute ();
      }
      if (limit - index < amount)
        amount = limit - index;
      index += (int) amount;
      return amount;
    }
  }
  public int available () throws IOException {
    if (closed) {
      throw new IOException ("Stream closed");
    } else {
      return limit - index;
    }
  }
```

データを消費するアプリケーションがこのストリームを閉じると、使用しているリソースをすべて解放できるように出力エンジンの`finish()` メソッドが呼び出されます。

リスト15. リソースの解放

```
public void close () throws IOException {
    if (!closed) {
      closed = true;
      engine.finish ();
    }
  }
```

出力エンジンがデータをその出力ストリームに書き込むと、`writeImpl()` メソッドが呼び出されます。そのメソッドは、これらのデータを読み取りバッファーにコピーし、読み取り限界インデックスを更新します。これによって、読み取りメソッドが自動的に新しいデータを利用できるようになります。

出力エンジンが単一の反復の中で、バッファーに保持できる以上のデータを書き込むと、バッファー容量が2倍になります。しかし、これがあまり頻繁に起きるのは望ましくありません。バッファーは、安定した状態で動作するように十分なサイズまで急速に拡張する必要があります。

リスト16. writeImpl() メソッド

```
private void writeImpl (byte[] data, int offset, int length) {
    if (index >= limit)
      index = limit = 0;
    if (limit + length > capacity) {
      capacity = capacity * 2 + length;
      byte[] tmp = new byte[capacity];
      System.arraycopy (buffer, index, tmp, 0, limit - index);
      buffer = tmp;
      limit -= index;
      index = 0;
    }
    System.arraycopy (data, offset, buffer, limit, length);
    limit += length;
  }
```

下のリスト17に示されている内部出力ストリーム実装は、データを内部入力ストリーム・バッファーに書き込むストリームを表しています。コードはパラメーターが受け入れ可能かどうかを確認し、受け入れ可能であれば、`writeImpl()` メソッドを呼び出します。

リスト17. 内部出力ストリーム実装

```
private class OutputStreamImpl extends OutputStream {
    public void write (int datum) throws IOException {
      one[0] = (byte) datum;
      write (one, 0, 1);
    }
    public void write (byte[] data, int offset, int length) throws IOException {
      if (data == null) {
        throw new NullPointerException ();
      } else if ((offset < 0) || (length < 0) || (offset + length > data.length)) {
        throw new IndexOutOfBoundsException ();
      } else if (eof) {
        throw new IOException ("Stream closed");
      } else {
        writeImpl (data, offset, length);
      }
    }
```

最後に出力エンジンがその出力ストリームを閉じ、それ以上書き込むデータがないことを示すと、この出力ストリームは入力ストリームの`eof` フラグを設定し、それ以上読み取るデータがないことを示します。

リスト18. 入力ストリームのeofフラグの設定

```
    public void close () {
      eof = true;
    }
  }
}
```

鋭い読者なら、この出力ストリーム実装に直接`writeImpl()` メソッドの本体を置くこともできたのではないかとお気付きかもしれません。内部クラスは、エンクロージング・クラスのすべてのプライベート・メンバーにアクセスすることができるからです。しかし、そのようなフィールドへの内部クラス・アクセスは、エンクロージング・クラスの直接メソッドによるアクセスよりも若干非効率です。そこで、効率化のために、またクラス間の依存関係を最小限に抑えるために、私は追加のヘルパー・メソッドを使用しています。

エンジニアリング・ソリューションの適用: 読み取り中のデータの圧縮

リスト19は、データを読み取る際に圧縮するという最初の問題を解決するためのクラスのフレームワークの使用を示しています。このソリューションをひとことで言えば、入力ストリームに関連する`IOStreamEngine` と`GZIPOutputStreamFactory` を作成し、`OutputEngineInputStream` をこれに接続することです。これらのストリームの初期化および接続は自動的に実行され、圧縮データはその結果のストリームから直接読み取ることができます。処理が完了し、ストリームを閉じると、出力エンジンが自動的にシャットダウンし、元の入力ストリームを閉じます。

リスト19. エンジニアリング・ソリューションの適用

```
  private static InputStream engineCompress (InputStream in) throws IOException {
    return new OutputEngineInputStream
      (new IOStreamEngine (in, new GZIPOutputStreamFactory ()));
  }
```

この種の問題に対応するように設計されたソリューションが、結果としてはるかに明快なコードになることはまさに驚くべきことですが、ここからは、より広く当てはまる教訓が得られます。

つまり問題の大小を問わず、優れた設計技法を適用すれば、その結果は必ずと言ってよいほどより明快で保守しやすいコードになるということです。

テスト・パフォーマンス

効率の点から見ると、`IOStreamEngine` はデータをその内部バッファーに読み込み、圧縮フィルターを介してそれらを`OutputStreamImpl` に書き込みます。これはデータを直接`OutputEngineInputStream` に書き込み、そこでデータを読み取り用に利用できるようにします。バッファー・コピーは全部で2つしか実行されません。つまり、パイプ・ストリーム・ソリューションにおけるバッファー・コピーの効率性と、力ずくのソリューションにおけるスレッドレスの効率性を組み合わせることによって、恩恵を受けることができるということです。

実際にパフォーマンスをテストするために、提示した3つのソリューションを使用し、ヌル・フィルターを用いてダミー・データのブロック全体を読み取る単純なテスト・プログラムを作成しました。Java 2 SDK、バージョン1.4.0を稼働する800 MHz Linuxボックスで、以下のパフォーマンスを達成しました。

パイプ・ストリーム・ソリューション
　　15KB: 23ms、15MB: 22100ms
力ずくのソリューション
　　15KB: 0.35ms、15MB: 745ms
エンジニアリング・ソリューション
　　15KB: 0.16ms、15MB: 73ms
この問題に対するエンジニアリング・ソリューションは、標準のJava APIに基づいた他のソリューションのいずれよりも明らかに効率的です。

余談ですが、出力エンジンが、データを出力ストリームに書き込んだ後、データの書き込み元である配列を修正することなく戻るというコントラクトに従うことができれば、単一のバッファー・コピー操作のみを使用するソリューションを提供することも可能です。しかし、このコントラクトが守られることはめったにありません。必要な場合は、適切なマーカー・インターフェースを実装するだけで、出力エンジンはこの操作モードに対するサポートを知らせることも可能です。

エンジニアリング・ソリューションの適用: エンコードされた文字データの読み取り

データを繰り返し`OutputStream` に書き込むエンティティーに対して読み取りアクセスを提供するという観点で表すことができる問題は、すべてこのフレームワークによって解決できます。このセクションと次のセクションでは、そのような問題の例とその効率的なソリューションについて考えてみます。

まず、文字ストリームのUTF-8エンコード形式を読み取る必要のある場合について考えてみましょう。`InputStreamReader` クラスによって、バイナリー・エンコード文字データをUnicode文字のシーケンスとして読み取ることができます。これは、バイト入力ストリームから文字入力ストリームへのゲートウェイを表します。また`OutputStreamWriter` クラスによって、Unicode文字のシーケンスをバイナリー・エンコード形式で出力ストリームに書き込むことができます。これ

は、文字出力ストリームからバイト出力ストリームへのゲートウェイを表します。`String` クラスの`getBytes()` メソッドは、ストリングをエンコードされたバイト配列に変換します。しかしこれらのクラスはどれも、文字ストリームとしてのUTF-8エンコード形式を直接読み取ることはできません。

リスト20から24までのコードは、`IOStreamEngine` クラスと非常によく似た方法で`OutputEngine` フレームワークを使用するソリューションを示しています。入力ストリームから読み取り、出力ストリーム・フィルターを介して書き込むのではなく、文字ストリームから読み取り、選択した文字エンコード方式を使用して`OutputStreamWriter` を介して書き込みます。

リスト20. エンコードされた文字データの読み取り

```
package org.merlin.io;
import java.io.*;
/**
 * An output engine that copies data from a Reader through
 * a OutputStreamWriter to the target OutputStream.
 *
 * @author Copyright (c) 2002 Merlin Hughes <merlin@merlin.org>
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version 2
 * of the License, or (at your option) any later version.
 */
public class ReaderWriterEngine implements OutputEngine {
  private static final int DEFAULT_BUFFER_SIZE = 8192;
  private Reader reader;
  private String encoding;
  private char[] buffer;
  private Writer writer;
```

このクラスのコンストラクターは、読み取り元の文字ストリーム、使用するエンコード方式、およびオプションのバッファー・サイズを受け入れます。

リスト21. コンストラクター

```
public ReaderWriterEngine (Reader in, String encoding) {
    this (in, encoding, DEFAULT_BUFFER_SIZE);
  }
  public ReaderWriterEngine (Reader reader, String encoding, int bufferSize) {
    this.reader = reader;
    this.encoding = encoding;
    buffer = new char[bufferSize];
  }
```

初期化されるとこのエンジンは、提供される出力ストリームに選択されたエンコード方式で文字を書き込む`OutputStreamWriter` を接続します。

リスト22. 出力ストリーム・ライターの初期化

```
  public void initialize (OutputStream out) throws IOException {
    if (writer != null) {
      throw new IOException ("Already initialised");
    } else {
      writer = new OutputStreamWriter (out, encoding);
    }
  }
```

実行されると、このエンジンはデータを入力文字ストリームから読み取り、`OutputStreamWriter` に書き込みます。この`OutputStreamWriter` は、接続されている出力ストリームに、選択されたエンコード方式でデータを渡します。ここから、このフレームワークはデータを読み取り用に利用できるようにします。

リスト23. データの読み取り

```
  public void execute () throws IOException {
    if (writer == null) {
      throw new IOException ("Not yet initialised");
    } else {
      int amount = reader.read (buffer);
      if (amount < 0) {
        writer.close ();
      } else {
        writer.write (buffer, 0, amount);
      }
    }
  }
```

エンジンが終了すると、入力も終了します。

リスト24. 入力の終了

```
  public void finish () throws IOException {
    reader.close ();
  }
}
```

この場合は、圧縮のケースと異なり、Java I/Oパッケージは、`OutputStreamWriter` の下にある文字エンコード・クラスへの低レベルのアクセスを提供しません。その結果、これはJavaプラットフォームの1.4より前のリリースにおける文字ストリームのエンコード形式読み取りに対する唯一の効果的なソリューションとなります。バージョン1.4の時点で、`java.nio.charset` パッケージは、ストリームに依存しない文字エンコード/デコード機能を提供します。しかしこのパッケージは、入力ストリーム・ベースのソリューションに対する我々の要件を満たしていません。

エンジニアリング・ソリューションの適用: シリアル化されたDOM文書の読み取り

最後に、このフレームワークのもう1つの使用方法を見てみましょう。リスト25から29までのコードは、DOM文書または文書サブセットのシリアル化された形式を読み取るためのソリューションを示しています。このコードの潜在的な用途は、DOM文書の一部に対して検証の再解析を実行することかもしれません。

リスト25. シリアル化されたDOM文書の読み取り

```
package org.merlin.io;
import java.io.*;
import java.util.*;
import org.w3c.dom.*;
import org.w3c.dom.traversal.*;
/**
 * An output engine that serializes a DOM tree using a specified
 * character encoding to the target OutputStream.
 *
 * @author Copyright (c) 2002 Merlin Hughes <merlin@merlin.org>
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * as published by the Free Software Foundation; either version 2
 * of the License, or (at your option) any later version.
 */
public class DOMSerializerEngine implements OutputEngine {
  private NodeIterator iterator;
  private String encoding;
  private OutputStreamWriter writer;
```

コンストラクターは、反復処理対象のDOMノード、または事前に作成されたノード反復子 (これはDOM 2の一部)、およびシリアル化された形式用に用いるエンコード方式を受取ります。

リスト26. コンストラクター

```
public DOMSerializerEngine (Node root) {
    this (root, "UTF-8");
  }
  public DOMSerializerEngine (Node root, String encoding) {
    this (getIterator (root), encoding);
  }
  private static NodeIterator getIterator (Node node) {
    DocumentTraversal dt= (DocumentTraversal)
      (node.getNodeType () == Node.DOCUMENT_NODE) ? node : node.getOwnerDocument ();
    return dt.createNodeIterator (node, NodeFilter.SHOW_ALL, null, false);
  }  public DOMSerializerEngine (NodeIterator iterator, String encoding) {
    this.iterator = iterator;
    this.encoding = encoding;
  }
```

初期化中にエンジンは、適切な`OutputStreamWriter` をターゲットの出力ストリームに接続します。

リスト27. initialize() メソッド

```
  public void initialize (OutputStream out) throws IOException {
    if (writer != null) {
      throw new IOException ("Already initialised");
    } else {
      writer = new OutputStreamWriter (out, encoding);
    }
  }
```

実行フェーズの間、エンジンはノード反復子から次のノードを取得し、それを`OutputStreamWriter` にシリアル化します。ノードがなくなると、エンジンはそのストリームを閉じます。

リスト28. execute() メソッド

```
  public void execute () throws IOException {
    if (writer == null) {
      throw new IOException ("Not yet initialised");
    } else {
      Node node = iterator.nextNode ();
      closeElements (node);
      if (node == null) {
        writer.close ();



      } else {
        writeNode (node);
        writer.flush ();
      }
    }
  }
```

このエンジンがシャットダウンするときに、解放するリソースはありません。

リスト29. シャットダウン

```
  public void finish () throws IOException {
  }
  // private void closeElements (Node node) throws IOException ...
  // private void writeNode (Node node) throws IOException ...
}
```

各ノードのシリアル化に関する内部処理の残りの部分は、あまり興味を引くものではありません。基本的にこのプロセスは、そのタイプとXML 1.0仕様に従ってノードを書き出すことに関係しているので、この記事ではコードのその部分は省略します。

結論

これまで説明してきたのは、出力ストリームに書き込むことしかできないシステムによって作成されたデータを、標準の入力ストリームAPIを使用して効率的に読み取ることを可能にする有用なフレームワークです。これによって、圧縮またはエンコードされたデータ、シリアル化された文書などを読み取ることができます。この機能は標準のJava APIでも不可能ではありませんでしたが、それらのクラスを使用することは決して効率的ではありませんでした。このソリューションは、小さなデータ・サイズの場合でさえ、最も単純な力ずくのソリューションよりも効率的だという点に留意してください。後続の処理のためにデータを`ByteArrayOutputStream` に書き込むアプリケーションは、すべてこのフレームワークから恩恵を受けることができます。

バイト配列ストリームの貧弱なパフォーマンスと、パイプ・ストリームの信じられないほど貧弱なパフォーマンスは、実は次回の記事のトピックです。その中で私は、そのクラスの元の作成者よりもパフォーマンスに一層の重点を置いて、それらのクラスの再実装について説明したいと思います。APIコントラクトをわずかに緩めるだけで、100倍のパフォーマンス向上が可能になります。

私は皿洗いが嫌いです。しかし、これらのクラスの背景にあるアイデアを思いついたのは、私が良い出来だと考えるアイデア (多くの場合はつまらないアイデアですが) のほとんどの場合と同じ

ように、皿洗いの最中でした。たいていの場合、実際のコードから離れ、一歩退いて問題についてより広い視野で考えることが、最終的には、安易な解決手段をとるよりもはるかによい結果をもたらす、より優れたソリューションを生み出すということがわかりました。多くの場合、これらのソリューションは、より明快で効率的かつ保守しやすいコードになります。

正直なところ、食器洗い機を使う日が来るのを恐れています。

著者について

Merlin Hughes

Merlinは、アイルランドのE-security企業であるBaltimore Technologies の暗号研究者および主任技術指導者であり、ときには執筆を行うこともあり、家庭では掃除人や皿洗いの役目も果たします。JDK 1.4のMerlinと混同しないでください。彼は、ニューヨーク州ニューヨークに住んでいます (とてもすばらしい町なので、同じ名前を二度使っているのです)。彼の連絡先はmerlin@merlin.org です。