OEP

AIM: Compression and decompression to realize the use of Java API in data compression.

Overview of Data Compression

The simplest type of redundancy in a file is the repetition of characters. For example, consider the following string:

```
BBBBHHDDXXXXKKKKWWZZZZ
```

This string can be encoded more compactly by replacing each repeated string of characters by a single instance of the repeated character and a number that represents the number of times it is repeated. The earlier string can be encoded as follows:

```
4B2H2D4X4K2W4Z
```

Here "4B" means four B's, and 2H means two H's, and so on. Compressing a string in this way is called run-length encoding.

As another example, consider the storage of a rectangular image. As a single-color bitmapped image, it can be stored as shown in Figure 1.

```
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000001111111111111111111100000000000
00000000001000000000000000000100000000000
00000000001000000000000000000100000000000
00000000001000000000000000000100000000000
00000000001111111111111111111100000000000
00000000000000000000000000000000000000000
```

Figure 1: A bitmap with information for run-length encoding

Another approach might be to store the image as a graphics metafile:

```
Rectangle 11, 3, 20, 5
```

This says, the rectangle starts at coordinate (11, 3) of width 20 and length 5 pixels.

The rectangular image can be compressed with run-length encoding by counting identical bits as follows:

0, 40

0, 40

0,10 1,20 0,10

0,10 1,1 0,18 1,1 0,10

0,10 1,1 0,18 1,1 0,10

0,10 1,1 0,18 1,1 0,10

0,10 1,20 0,10

0,40

The first line above says that the first line of the bitmap consists of 40 0's. The third line says that the third line of the bitmap consists of 10 0's followed by 20 1's followed by 10 more 0's, and so on for the other lines.

There are many benefits to data compression. The main advantage of it, however, is to reduce storage requirements. Also, for data communications, the transfer of compressed data over a medium result in an increase in the rate of information transfer. Note that data compression can be implemented on existing hardware by software or through the use of special hardware devices that incorporate compression techniques. Figure 2 shows a basic data-compression block diagram.

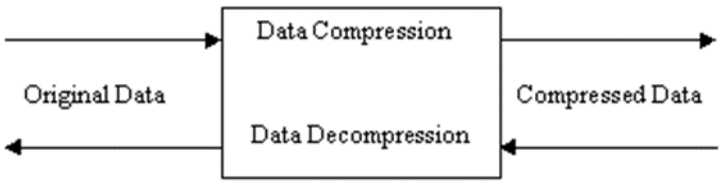

Figure 2: Data-compression block diagram

ZIP vs. GZIP

If you are working on Windows, you might be familiar with the WinZip tool, which is used to create a compressed archive and to extract files from a compressed archive. On UNIX, however, things are done a bit differently. The tar command is used to create an archive (not compressed) and another program (gzip or compress) is used to compress the archive.

Tools such as WinZip and PKZIP act as both an archiver and a compressor. They compress files and store them in an archive. On the other hand, gzip does not archive files. Therefore, on UNIX, the tar command is usually used to create an archive then the gzipcommand is used to compress the archived file.

The java.util.zip Package

Java provides the java.util.zip package for zip-compatible data compression. It provides classes that allow you to read, create, and modify ZIP and GZIP file formats. It also provides utility classes for computing checksums of arbitrary input streams that can be used to validate input data. This package provides one interface, fourteen classes, and two exception classes as shown in Table 1.

Table 1: The java.util.zip package

Item	Type	Description
Checksum	Interface	Represents a data checksum. Implemented by the classes Adler32 and CRC32

Adler32	Class	Used to compute the Adler32 checksum of a data stream
CheckedInputStream	Class	An input stream that maintains the checksum of the data being read
CheckedOutputStream	Class	An output stream that maintains the checksum of the data being written
CRC32	Class	Used to compute the CRC32 checksum of a data stream
Deflater	Class	Supports general compression using the ZLIB compression library
DeflaterOutputStream	Class	An output stream filter for compressing data in the deflate compression format
GZIPInputStream	Class	An input stream filter for reading compressed data in the GZIP file format
GZIPOutputStream	Class	An output stream filter for writing compressed data in the GZIP file format
Inflater	Class	Supports general decompression using the ZLIB compression library
InlfaterInputStream	Class	An input stream filter for decompressing data in the deflate compression format
ZipEntry	Class	Represents a ZIP file entry
ZipFile	Class	Used to read entries from a ZIP file
ZipInputStream	Class	An input stream filter for reading files in the ZIP file format
ZipOutputStream	Class	An output stream filter for writing files in the ZIP file format
DataFormatException	Exception Class	Thrown to signal a data format error
ZipException	Exception Class	Thrown to signal a zip error

Decompressing and Extracting Data from a ZIP file

The java.util.zip package provides classes for data compression and decompression. Decompressing a ZIP file is a matter of reading data from an input stream. The java.util.zip package provides a ZipInputStream class for reading ZIP files. A ZipInputStream can be created just like any other input stream. For example, the following segment of code can be used to create an input stream for reading data from a ZIP file format:

FileInputStream fis = new FileInputStream("figs.zip"); ZipInputStream zin = new ZipInputStream(new BufferedInputStream(fis));

Once a ZIP input stream is opened, you can read the zip entries using the getNextEntry method which returns a ZipEntry object. If the end-of-file is reached, getNextEntry returns null:

```
ZipEntry entry;

while((entry = zin.getNextEntry()) != null) {          }
```

Now, it is time to set up a decompressed output stream, which can be done as follows:

```
int BUFFER = 2048;

FileOutputStream fos = new FileOutputStream(entry.getName());

BufferedOutputStream dest = new BufferedOutputStream(fos, BUFFER);
```

In this segment of code, a file output stream is created using the entry's name, which can be retrieved using the entry.getName method. Source zipped data is then read and written to the decompressed stream:

```
while ((count = zin.read(data, 0, BUFFER)) != -1) {

  //System.out.write(x);

  dest.write(data, 0, count);

}
```

And finally, close the input and output streams:

```
dest.flush();

dest.close();

zin.close();
```

The source program in Code Sample 1 shows how to decompress and extract files from a ZIP archive. To test this sample, compile the class and run it by passing a compressed file in ZIP format:

```
prompt> java UnZip somefile.zip
```

Note that somefile.zip could be a ZIP archive created using any ZIP-compatible tool, such as WinZip.

Code Sample 1: UnZip.java

```
import java.io.*;

import java.util.zip.*;

public class UnZip {

  final int BUFFER = 2048;
```

```
  public static void main (String argv[]) {
    try {
      BufferedOutputStream dest = null;
      FileInputStream fis = new
            FileInputStream(argv[0]);
      ZipInputStream zis = new
            ZipInputStream(new BufferedInputStream(fis));
      ZipEntry entry;
      while((entry = zis.getNextEntry()) != null) {
        System.out.println("Extracting: " +entry);
        int count;
        byte data[] = new byte[BUFFER];
        // write the files to the disk
        FileOutputStream fos = new
              FileOutputStream(entry.getName());
        dest = new
         BufferedOutputStream(fos, BUFFER);
        while ((count = zis.read(data, 0, BUFFER))
          != -1) {
          dest.write(data, 0, count);
        }
        dest.flush();
        dest.close();
      }
      zis.close();
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

What about JAR Files?

The Java Archive (JAR) format is based on the standard ZIP file format with an optional manifest file. If you wish to create JAR files or extract files from a JAR file from within your Java applications, use the java.util.jar package, which provides classes for reading and writing JAR files. Using the classes provided by the java.util.jar package is very similar to using the classes provided by the java.util.zip package as described in this article. Therefore, you should be able to adapt much of the code in this article if you wish to use the java.util.jar package.

Conclusion

This article discussed the APIs that you can use to compress and decompress data from within your applications, with code samples throughout the article to show how to use the java.util.zip package to compress and decompress data. Now you have the tools to utilize data compression and decompression in your applications.

The article also shows how to compress and decompress data on the fly in order to reduce network traffic and improve the performance of your client/server applications. Compressing data on the fly, however, improves the performance of client/server applications only when the objects being compressed are more than a couple of hundred bytes.