CHAPTER 4

I/O Performance

BOTH client and server programs often use the `java.io` package. While this package is designed with ease of use in mind, developers new to the platform often make mistakes that can lead to poor I/O performance. Fortunately, a better understanding of this package can lead to major improvements in I/O performance.

Section 4.1 provides a brief overview of the `java.io` package, identifies the most common cause of poor I/O performance, and describes several different approaches to solving this problem. Section 4.2 focuses on object serialization, a key part of many of the newest features in the Java platform.

4.1 Basic I/O

Two abstract classes shape the architecture of the `java.io` package: `InputStream` and `OutputStream`. These interfaces define the key abstractions for I/O operations. Concrete implementations of `InputStream` and `OutputStream` provide access to different types of data sources such as disks and network connections.

The `java.io` package also provides several *filter* streams that don't point to a specific data source and are meant to be stacked on top of other streams. These filter streams are at the heart of the `java.io`

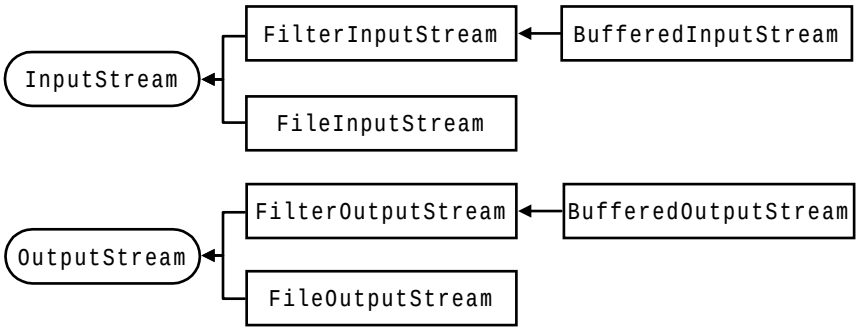

Figure 4-1 Simplified `java.io` class hierarchy

type, the buffering behavior is implemented in dedicated `BufferedInputStream` and `BufferedOutputStream` classes. This is very different from C, where the basic `stdio` operations are buffered by default. To better understand the effects of buffering streams, look at Listing 4-1. This example copies a file from one location to another.

```
public static void copy(String from, String to) throws IOException{
   InputStream in = null;
   OutputStream out = null;
   try {
      in = new FileInputStream(from);
      out = new FileOutputStream(to);
      while (true) {
         int data = in.read();
         if (data == -1) {
            break;
         }
         out.write(data);
      }
      in.close();
      out.close();
   } finally {
      if (in != null) {
         in.close();
      }
      if (out != null) {
         out.close();
      }
   }
}
```

Listing 4-1 Simple file copy

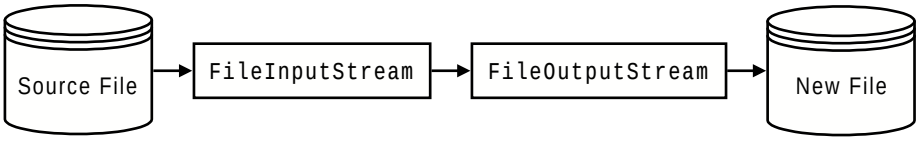

Figure 4-2 Copying with basic streams

The `copy` method opens a `FileInputStream` and a `FileOutputStream` and copies the contents of one directly into the other. Since the `read` and `write` methods work on individual bytes, this means an actual disk read and write occurs for each byte copied. Figure 4-2 illustrates how the data moves from one file to the other.

When the code in Listing 4-1 is run on our test configuration to copy a 370K JPEG test image, it takes almost 11 seconds to execute. Listing 4-2 shows a slightly modified version of the same code that uses buffered streams to improve performance. This code stacks buffered streams on top of the bare file streams. The buffered streams save up read and write requests and then execute them all at once—usually batching several thousand tiny requests into one larger request. When the code in Listing 4-2 is used to copy the same JPEG test file, it executes in a mere 130 milliseconds—almost 100 times faster!

```
public static void copy(String from, String to) throws IOException{
   InputStream in = null;
   OutputStream out = null;
   try {
      InputStream inFile = new FileInputStream(from);
      in = new BufferedInputStream(inFile);
      OutputStream outFile = new FileOutputStream(to);
      out = new BufferedOutputStream(outFile);
      while (true) {
         int data = in.read();
         if (data == -1) {
            break;
         }
         out.write(data);
      }
   } finally {
      if (in != null) {
         in.close();
      }
      if (out != null) {
         out.close();
      }
   }
}
```

Listing 4-2 Faster file copy

Figure 4-3 shows how the data flows from one file to another when you're using buffers. Although using buffered streams is much faster than using bare file streams, you can see from Figure 4-3 that the buffers add a level of indirection. This does add some overhead and in some cases it's possible to improve performance by implementing a custom buffering scheme.

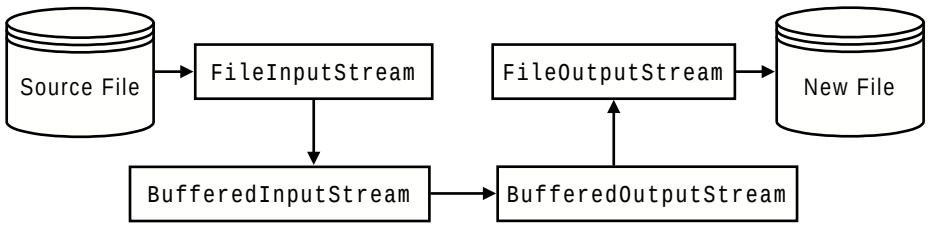

Figure 4-3 Copying with buffered streams

4.1.2 Custom Buffering

Copying data from one array to another using a `while` loop isn't as fast as you'd like it to be. Because the JVM enforces bounds checking, there is a good deal of overhead associated with such operations. When you're using buffered streams as shown in Listing 4-2, you essentially end up copying a lot of data from one array to another—several times, in fact. A large number of method calls are also made, many of them `synchronized`, with arguments that are passed and copied on the stack. Much of this overhead can be avoided, while still taking advantage of the fact that hard disks are good at reading and writing large chunks of data.

Listing 4-3 shows the `copy` method rewritten to use a custom buffering scheme. This block of code takes advantage of the fact that the `read` and `write` methods in `InputStream` and `OutputStream` are overloaded to work with `byte` arrays as well as individual bytes.

```
public int read();
public int read(byte[] bytes);

public void write();
public void write(byte[] bytes);
```

These methods allow you to work with large chunks of data, instead of just single bytes. In Listing 4-3 the code creates its own buffer, in the form of a `byte[]`, which it then uses to map the entire file into memory with a single `read` call. It then uses just one call to the `write` method to create the newly copied file.

```
public static void copy(String from, String to) throws IOException{
   InputStream in = null;
   OutputStream out = null;
   try {
      in = new FileInputStream(from);
      out = new FileOutputStream(to);
      int length = in.available(); // danger!
      byte[] bytes = new byte[length];
      in.read(bytes);
      out.write(bytes);
   } finally {
      if (in != null) {
         in.close();
      }
      if (out != null) {
         out.close();
      }
   }
}
```

Listing 4-3 Custom buffered copy

This code is very fast. Where the previous buffered stream version took about 130 milliseconds to copy the JPEG test file, using the single `read` and `write` operations reduces the time to about 33 milliseconds.

Note that there are two trade-offs to consider when using this strategy. First, this code creates a buffer the size of the original source file. If this code is used to copy large files, the buffer can get very large (perhaps larger than your available RAM). Second, this code creates a new buffer for each copy operation. If this code is used to copy a large number of files, the JVM has to allocate and collect many of these potentially large buffers. This is going to hurt performance.

It is possible to create a version of the custom buffered `copy` that avoids these two pitfalls and is even faster. To do this, you create a single, `static` buffer and then read or write blocks the size of that buffer. This results in more than one read or write operation for files larger than the buffer, but the cost is offset by the fact that a fresh buffer doesn't have to be allocated for each file that is copied. (The costs of such allocations are discussed further in Chapter 7, Object Mutability: Strings and Other Things.) The size of the buffer is known and can be optimized to achieve the best trade-off between speed and memory-use for each particular situation.

Listing 4-4 shows the code for the improved custom buffered `copy` function. In this version, a static 100K buffer is used for the copy operation. For copying the same JPEG test file, this implementation is even faster than the one in Listing 4-3. This is because a new buffer doesn't have to be allocated for every copy.

```
static final int BUFF_SIZE = 100000;
static final byte[] buffer = new byte[BUFF_SIZE];

public static void copy(String from, String to) throws IOException{
   InputStream in = null;
   OutputStream out = null;
   try {
      in = new FileInputStream(from);
      out = new FileOutputStream(to);
      while (true) {
         synchronized (buffer) {
            int amountRead = in.read(buffer);
            if (amountRead == -1) {
               break;
            }
            out.write(buffer, 0, amountRead);
         }
      }
   } finally {
      if (in != null) {
         in.close();
      }
      if (out != null) {
         out.close();
      }
   }
}
```

Listing 4-4 Improved custom buffered copy

One key item to note in Listing 4-4 is the `synchronized` block. In a single threaded environment, this code will work fine without the `synchronized` block. However, if you want to use this code in a multithreaded environment, you need to synchronize the buffer to prevent multiple threads from trying to write to it simultaneously. Although there are costs associated with synchronization, there is almost no performance impact because the number of iterations through the `while` loop is small. In our tests, both synchronized and unsynchronized versions of this code took the same amount of time to copy the test file.

Table 4-1 shows the approximate copy times for the different copy tactics. While the results from such micro-benchmarks don't show the whole performance picture, they provide a rough idea of how the different options compare.

Strategy	Time
Raw File Streams	10800 ms
Buffered Streams	130 ms
Custom Buffer	33 ms
Custom Buffer 2	22 ms

Table 4-1 Copy Times

4.1.3 Further Improvements

Typically, there are application-specific ways to further improve I/O performance. By examining an application's purpose and operation, you can often find opportunities for big performance improvements.

For example, consider an FTP or HTTP server. The primary job of such a server is to copy files from a disk to a network socket. Although a server might have access to thousands of files, a small fraction of these files typically represent the majority of the files served. A web site's main homepage is probably served much more often than other pages in the site. To improve performance, you could implement your server so that the most commonly accessed files are mapped into cached `byte[]` structures. That way, those files don't need to be read from disk each time; they can be copied directly from memory to the network socket.

4.2 Serialization

Object serialization is the process through which live objects are flattened into a form that can be written to and read from a stream. These flattened objects can be piped into files or even sent across a network. Together, object serialization and the Java platform's hardware-independent byte-codes enable revolutionary software solutions, such as advanced distributed system software and mobile agent software. Technologies such as Remote Method Invocation (RMI), and in turn Jini™ technology, rely on serialization. The `java.io` package provides classes that support object serialization.

One of the best features of serialization is that it is almost completely automatic. It takes very little work to use serialization to make your objects persistent—in general, all you need to do is implement the `Serializable` interface.

The outward simplicity of the serialization API hides its internal complexity. The internal machinery that enables transparent object serialization is very

complex and can be quite costly at runtime. Fortunately, there are tactics you can use to mitigate these costs.

4.2.1 Serialization Example

The `Serializable` interface is actually purely a tagging interface—it doesn't define any methods. `Serializable` is simply used to indicate that the class is designed to be serialized. Listing 4-5 shows a simple class that implements the `Serializable` interface.

```
public class TestObject implements Serializable {

    private int value;
    private String name;
    private Date timeStamp;
    private JPanel panel;

    public TestObject(int value) {
        this.value = value;
        name = new String("Object:" + value);
        timeStamp = new Date();
        panel = new JPanel();
        panel.add(new JTextField());
        panel.add(new JButton("Help"));
        panel.add(new JLabel("This is a text label"));
    }
}
```

Listing 4-5 Simple serializable class

Because this class implements the `Serializable` interface and the instance variables are serializable, any instance of this class can be written to an `ObjectOutputStream`. Listing 4-6 shows a code fragment that creates 50 instances of the `TestObject` class and writes them to an `ObjectOutputStream`.

```
for (int i =0;i <50; i++) {
   vector.addElement(new TestObject(i));
}
Stopwatch timer = new Stopwatch().start();
try {
   OutputStream file = new FileOutputStream("Out.test");
   OutputStream buffer = new BufferedOutputStream(file);
   ObjectOutputStream out = new ObjectOutputStream(buffer);
   out.writeObject(vector);
   out.close();
} catch (Exception e) {
   e.printStackTrace();
```

```
}
timer.stop();
System.out.println("elapsed = " + timer.getElapsedTime());
```

Listing 4-6 Writing objects to a stream

Once you've streamed these objects to disk, you can re-create them using an `ObjectInputStream`. Listing 4-7 shows a code fragment that loads objects from a file.

```
Stopwatch timer = new Stopwatch().start();
try {
   InputStream file = new FileInputStream("Out.test");
   InputStream buffer = new BufferedInputStream(file);
   ObjectInputStream in = new ObjectInputStream(buffer);
   vector = (Vector)in.readObject();
   in.close();
} catch (Exception e) {
   e.printStackTrace();
}
timer.stop();
System.out.println("elapsed = " + timer.getElapsedTime());
```

Listing 4-7 Reading objects from a stream

When the file created by writing a `Vector` that contains 50 of these test objects is written to disk, the resulting file occupies about 91K. That's almost 2K per instance of the class. That seems like a lot, considering the entire source file that defines the class takes up only about 500 bytes. Clearly, there is a lot of stuff being written into these files. This stems from the fact that serialization is a recursive process. When a single object is serialized, the `ObjectOutputStream` examines all the object's fields (even the private ones) and writes them out. If the fields contain other objects, those are also written out, and so on.

In the `TestObject` example, all of the Swing UI widgets and any objects they reference are written out along with the `TestObject`. Even fields set to the same value that is set by the class's default constructor are written out, because serialization has no knowledge of the default values for fields.

4.2.2 Improved Serialization Example

The good news is that the serialization mechanism provides tools that give you better control over what's written out. The primary tool for this is the `transient` keyword. This keyword allows you to specify values that are noncritical, or that can be reconstructed manually after the object is read into memory.

Listing 4-8 shows a new version of the `TestObject` class that uses the `transient` keyword to specify that two of the four fields defined by the class should not to be written out by the `ObjectOutputStream`. It also defines a `private` method called `readObject`. The `ObjectInputStream` class looks for a method with this signature (using machinery from within the JVM) and automatically calls this method while reading the object in, even though it is `private`. This hook can be used to reinitialize transient state in your object after it has been reconstructed from a stream. In this example, the user interface panel and the name are reinitialized after the class is read in. This dramatically improves performance for both reading and writing.

```
public class TestObjectTrans implements Serializable {

    private int value;
    private transient String name;
    private DatetimeStamp;
    private transient JPanel panel;

    public TestObjectTrans(int value) {
        this.value = value;
        timeStamp = new Date();
        initTransients();
    }

    public void initTransients() {
        name = new String("Object:" + value);
        panel = new JPanel();
        panel.add(new JTextField());
        panel.add(new JButton("Help"));
        panel.add(new JLabel("This is a text label"));
    }

    private void readObject(ObjectInputStream in)
                throws IOException, ClassNotFoundException {
        in.defaultReadObject();
        initTransients();
    }
}
```

Listing 4-8 Improved serializable object

Table 4-2 shows a comparison between the `TestObject` implementation and the modified version that uses the `transient` keyword to prevent selected fields from being written out.

	TestObject	TestObjectTrans
Save Time	990 ms	110 ms
Load Time	3,680 ms	1,040 ms
File Size	91.7K	1.6K

Table 4-2 Serialization Comparison

As you can see, using the `transient` keyword drastically improves the performance of serializing these objects:

- The time to flatten and save 50 of these objects was reduced by nine times.
- The time to load and reconstruct the objects was reduced by four times.
- The size of the resulting file was reduced by more than 50 times.

Note that this last figure is especially crucial if you are piping these objects over a network instead of just to a local file. Making a change like this in an application that uses RMI to move objects could result in a significant reduction in network traffic.

4.2.3 Analyzing Persistent State

If your software depends on the serialization mechanism, then you need to perform some analysis to determine what object state information needs to be persistent and what can be recalculated after an object has been reconstituted.

In the example in Listing 4-8, streaming out a simple object causes several Swing user interface components to be streamed out as well. While this example might seem artificial, it's actually a simplified version of a problem encountered by a group of developers who were working on a server program.

When the server process needed to be terminated, the program serialized a set of important objects and streamed them to a file. When the server was restarted, this enabled it to re-create these objects and begin running in the same state where it left off. The problem was that it took more than 30 minutes to stream these objects to disk, which was clearly unacceptable. When the developers analyzed the required persistent state of their objects, they found many things were being streamed that didn't need to be. In fact, because of the recursive nature of serialization, almost the entire object heap was being streamed to disk. Careful use of the `transient` keyword drastically reduced the time it took to write out their data.

Key Points

- Reading and writing data in small chunks can be very slow.
- Using `BufferedInputStream` and `BufferedOutputStream` to batch requests improves performance.
- In some situations, you can use custom buffering techniques to maximize performance.
- Serialization can be very costly.
- Using the `transient` keyword reduces the amount of data serialized.