# 【Java基础专题】IO与文件读写---使用Apache commons io包提高读写效率

## 【一】Apache commons IO简介

首先贴一段Apache commons IO官网上的介绍，来对这个著名的开源包有一个基本的了解：

Commons IO is a library of utilities to assist with developing IO functionality. There are four main areas included:

- Utility classes - with static methods to perform common tasks
- Filters - various implementations of file filters
- Comparators - various implementations of java.util.Comparator for files
- Streams - useful stream, reader and writer implementations

| Packages | |
|---|---|
| **org.apache.commons.io** | This package defines utility classes for working with streams, readers, writers and files. |
| **org.apache.commons.io.comparator** | This package provides various `Comparator` implementations for `File`s. |
| **org.apache.commons.io.filefilter** | This package defines an interface (IOFileFilter) that combines both `FileFilter` and `FilenameFilter`. |
| **org.apache.commons.io.input** | This package provides implementations of input classes, such as `InputStream` and `Reader`. |
| **org.apache.commons.io.output** | This |

## 【二】org.apache.comons.io.input包介绍

这个包针对SUN JDK IO包进行了扩展，实现了一些功能简单的IO类，主要包括了对字节/字符输入流接口的实现

这个包针对java.io.InputStream和Reader进行了扩展，其中比较实用的有以下几个：

**•AutoCloseInputStream**

*Proxy stream that closes and discards the underlying stream as soon as the end of input has been reached or when the stream is explicitly closed. Not even a reference to the underlying stream is kept after it has been closed, so any allocated in-memory buffers can be freed even if the client application still keeps a reference to the proxy stream*

*This class is typically used to release any resources related to an open stream as soon as possible even if the client application (by not explicitly closing the stream when no longer needed) or the underlying stream (by not releasing resources once the last byte has been read) do not do that.*

这个输入流是一个底层输入流的代理，它能够在数据源的内容被完全读取到输入流后，后者当用户调用close()方法时，立即关闭底层的输入流。释放底层的资源(例如文件的句柄)。这个类的好处就是避免我们在代码中忘记关闭底层的输入流而造成文件处于一直打开的状态。

我们知道对于某些文件，只允许由一个进程打开。如果我们使用后忘记关闭那么该文件将处于一直“打开”的状态，其它进程无法读写。例如下面的例子：

```
new BufferedInputStream(new FileInputStream(FILE))
```

里面的FileInputStream(FILE)在打开后不能被显式关闭，这将导致可能出现的问题。如果我们使用了AutoCloseInputStream，那么当数据读取完毕后，底层的输入流会被自动关闭，迅速地释放资源。

```
new BufferedInputStream(new AutoClosedInputStream(new FileInputStream));
```

那么这个类是如何做到自动关闭的呢？来看看这个非常简单的类的代码吧

```
package org.apache.commons.io.input;
import java.io.IOException;
import java.io.InputStream;
public class AutoCloseInputStream extends ProxyInputStream {
    public AutoCloseInputStream(InputStream in) {
        super(in);
    }
    public void close() throws IOException {
        in.close();
        in = new ClosedInputStream();
    }
   public int read() throws IOException {
        int n = in.read();
        if (n == -1) {
            close();
        }
        return n;
    }
    public int read(byte[] b) throws IOException {
        int n = in.read(b);
        if (n == -1) {
            close();
        }
        return n;
    }

    public int read(byte[] b, int off, int len) throws IOException {
        int n = in.read(b, off, len);
        if (n == -1) {
            close();
        }
        return n;
    }
    protected void finalize() throws Throwable {
        close();
        super.finalize();
```

```
    }
}
public class ClosedInputStream extends InputStream {

    /**
     * A singleton.
     */
    public static final ClosedInputStream CLOSED_INPUT_STREAM = new ClosedInputStream();
    /**
     * Returns -1 to indicate that the stream is closed.
     *
     * @return always -1
     */
    public int read() {
        return -1;
    }
}
```

可以看到这个类通过两个途径来保证底层的流能够被正确地关闭：
①每次调用read方法时，如果底层读到的是-1，立即关闭底层输入流。返回一个ClosedInputStream
②当这个类的对象被回收时，确保关闭底层的输入流

**●TeeInputStream**

*InputStream proxy that transparently writes a copy of all bytes read from the proxied stream to a given OutputStream. The proxied input stream is closed when the `close()` method is called on this proxy. It is configurable whether the associated output stream will also closed.*

可以看到这个类的作用是把输入流读入的数据原封不动地传递给输出流。这一点和JDK中提供的PipedInputStream的理念有些类似。在实际使用中可以非常方便地做到像：将从远程URL读入的数据写到输出流，保存到文件之类的动作。当输入流被关闭时，输出流不一定被关闭。可以依然保持打开的状态。

下面是这个类的部分源码

```
    public int read(byte[] bts, int st, int end) throws IOException {
        int n = super.read(bts, st, end);
        if (n != -1) {
            branch.write(bts, st, n);
        }
        return n;
    }
```

**●CharSequenceReader**

`Reader` implementation that can read from String, StringBuffer, StringBuilder or CharBuffer.

这个类可以看成是对StringReader的一个扩展，用于从内存中读取字符。

**●NullReader**

A functional, light weight `Reader` that emulates a reader of a specified size.

This implementation provides a light weight object for testing with an `Reader` where the contents don't matter.

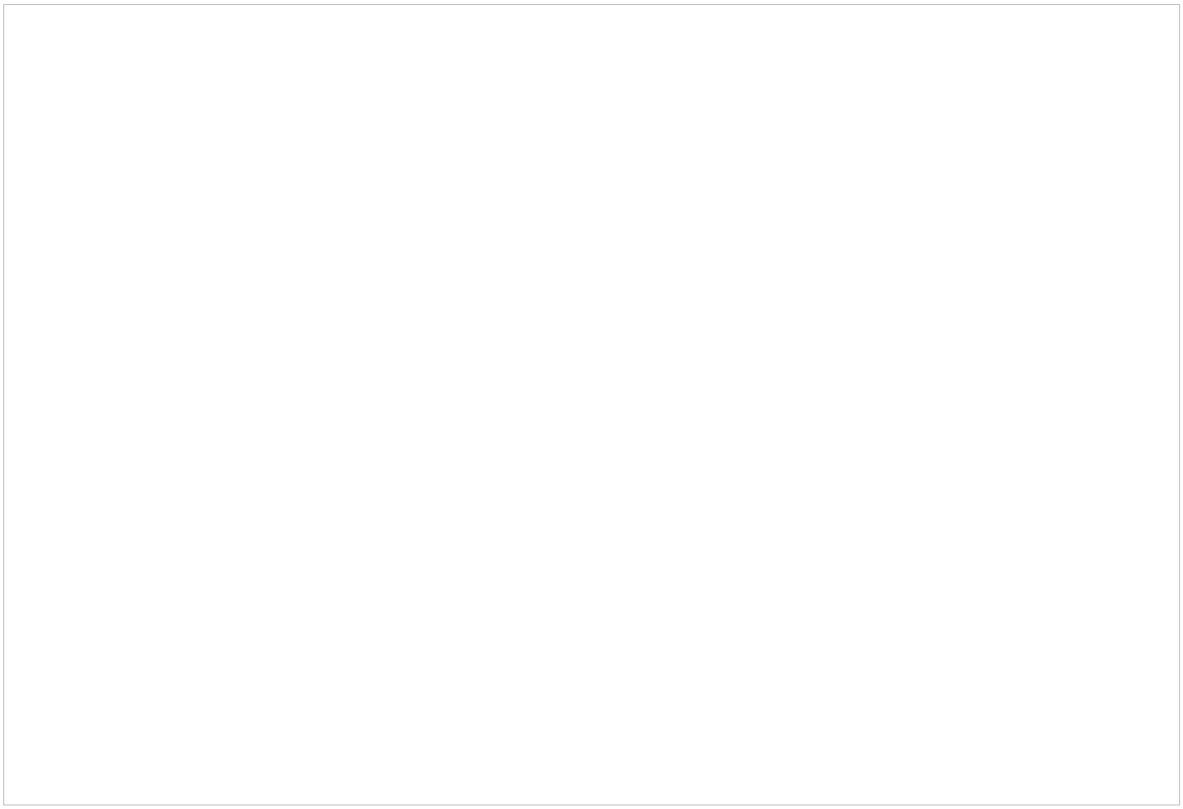
One use case would be for testing the handling of large `Reader` as it can emulate that scenario without the overhead of actually processing large numbers of characters - significantly speeding up test execution times.

从上面的文字描述来看，这个类显然是用来做测试辅助的，它的目标对象是“对读入内容不关心”的需求。它并不传递真正的数据，而是模拟这个过程。来看看下面的源代码

```
/**
 * Read the specified number characters into an array.
 *
 * @param chars The character array to read into.
 * @param offset The offset to start reading characters into.
 * @param length The number of characters to read.
 * @return The number of characters read or <code>-1</code>
 * if the end of file has been reached and
 * <code>throwEofException</code> is set to <code>false</code>.
 * @throws EOFException if the end of file is reached and
 * <code>throwEofException</code> is set to <code>true</code>.
 * @throws IOException if trying to read past the end of file.
 */
public int read(char[] chars, int offset, int length) throws IOException {
  if (eof) {
      throw new IOException("Read after end of file");
    }
  if (position == size) {
      return doEndOfFile();
    }
    position += length;
    int returnLength = length;
  if (position > size) {
      returnLength = length - (int)(position - size);
      position = size;
    }
    processChars(chars, offset, returnLength);
    return returnLength;
}
/**
 * Return a character value for the  <code>read()</code> method.
 * <p>
 * This implementation returns zero.
 *
 * @return This implementation always returns zero.
 */
protected int processChar() {
    // do nothing - overridable by subclass
    return 0;
}
/**
 * Process the characters for the <code>read(char[], offset, length)</code>
 * method.
 * <p>
 * This implementation leaves the character array unchanged.
 *
 * @param chars The character array
 * @param offset The offset to start at.
 * @param length The number of characters.
 */
protected void processChars(char[] chars, int offset, int length) {
    // do nothing - overridable by subclass
}
```

知道它是怎么模拟的了吗？呵呵~~。原来它只是模拟计数的过程，根本不传递、处理、存储任何数据。数组始终都是空的。

## 【三】org.apache.commons.io.output包介绍



和input包类似，output包也实现/继承了部分JDK IO包的类、接口。这里需要特别注意的有3个类，他们分别是：

①ByteArrayOutputStream
②FileWriterWithEncoding
③LockableFileWriter

**●ByteArrayOutputStream**

*This class implements an output stream in which the data is written into a byte array. The buffer automatically grows as data is written to it.*

*The data can be retrieved using `toByteArray()` and `toString()`.*

*Closing a `ByteArrayOutputStream` has no effect. The methods in this class can be called after the stream has been closed without generating an `IOException`.*

*This is an alternative implementation of the java.io.ByteArrayOutputStream class. The original implementation only allocates 32 bytes at the beginning. As this class is designed for heavy duty it starts at 1024 bytes. In contrast to the original it doesn't reallocate the whole memory block but allocates additional buffers. This way no buffers need to be garbage collected and the contents don't have to be copied to the new buffer. This class is designed to behave exactly like the original. The only exception is the deprecated toString(int) method that has been ignored.*

从上面的文档中，我们看到Apache commons io的ByteArrayOutputString比起SUN自带的ByteArrayOutputStream更加高效，原因在于：

①缓冲区的初始化大小比原始的JDK自带的ByteArrayOutputStream要大很多(1024:32)
②缓冲区的大小可以无限增加。当缓冲不够时动态增加分配，而非清空后再重新封闭
③减少write方法的调用次数，一次性将多个一级缓冲数据写出。减少堆栈调用的时间

那么为什么这个类可以做到这些呢？来看看他的源码吧：

```
/** The list of buffers, which grows and never reduces. */
private List buffers = new ArrayList();
```

```
    /** The current buffer. */
    private byte[] currentBuffer;
```

```
/**
     * Creates a new byte array output stream. The buffer capacity is
     * initially 1024 bytes, though its size increases if necessary.
     */
    public ByteArrayOutputStream() {
        this(1024);
    }
```

而JDK自带的Buffer则只有简单的一个byte[]

```
     /**
     * The buffer where data is stored.
     */
    protected byte buf[];
```

```
/**
     * Creates a new byte array output stream. The buffer capacity is
     * initially 32 bytes, though its size increases if necessary.
     */
    public ByteArrayOutputStream() {
    this(32);
    }
```

原来Apache commons 的io是采用了二级缓冲：首先一级缓冲是一个byte[]，随着每次写出的数据不同而不同。二级缓冲则是一个无限扩充的ArrayList，每次从byte[]中要写出的数据都会缓存到这里。当然效率上要高很多了。那么这个类是如何做到动态增加缓冲而不需要每次都回收已有的缓冲呢?

```
/**
     * Makes a new buffer available either by allocating
     * a new one or re-cycling an existing one.
     *
     * @param newcount  the size of the buffer if one is created
     */
    private void needNewBuffer(int newcount) {
        if (currentBufferIndex < buffers.size() - 1) {
            //Recycling old buffer
            filledBufferSum += currentBuffer.length;

            currentBufferIndex++;
            currentBuffer = getBuffer(currentBufferIndex);
        } else {
            //Creating new buffer
            int newBufferSize;
            if (currentBuffer == null) {
                newBufferSize = newcount;
                filledBufferSum = 0;
            } else {
                newBufferSize = Math.max(
                    currentBuffer.length << 1,
                    newcount - filledBufferSum);
                filledBufferSum += currentBuffer.length;
            }

            currentBufferIndex++;
            currentBuffer = new byte[newBufferSize];
            buffers.add(currentBuffer);
        }
    }
```

在初始化的情况下，currentBuffer == null，于是第一个一级缓冲区byte[]的大小就是默认的1024或者用户指定的值。然后filledBufferSum、currentBufferIndex分别进行初始化。创建第一个一级缓存区，添加到二级缓冲区buffers中。

当后续的缓冲请求到来后，根据剩下的缓冲大小和尚存的缓冲进行比较，然后选择较大的值作为缓冲扩展的大小。再次创建一个新的一级缓冲byte[]，添加到二级缓冲中。

相比于JDK自带的方法，这个类多了一个write(InputStream in)的方法，看看下面的源代码

```
public synchronized int write(InputStream in) throws IOException {
        int readCount = 0;
        int inBufferPos = count - filledBufferSum;
        int n = in.read(currentBuffer, inBufferPos, currentBuffer.length - inBufferPos);
        while (n != -1) {
            readCount += n;
            inBufferPos += n;
            count += n;
            if (inBufferPos == currentBuffer.length) {
                needNewBuffer(currentBuffer.length);
                inBufferPos = 0;
            }
            n = in.read(currentBuffer, inBufferPos, currentBuffer.length - inBufferPos);
        }
        return readCount;
    }
```

可以看到每次从InputStream读取当前一级缓冲剩余空间大小的字节，缓冲到剩下的空间。如果缓冲满了则继续分配新的一级缓冲。直至数据读完。对于写出到另外的输出流，则是：

```
public synchronized void writeTo(OutputStream out) throws IOException {
        int remaining = count;
        for (int i = 0; i < buffers.size(); i++) {
            byte[] buf = getBuffer(i);
            int c = Math.min(buf.length, remaining);
            out.write(buf, 0, c);
            remaining -= c;
            if (remaining == 0) {
                break;
            }
        }
    }
```

由于每次write的时候一次性地写出一级缓冲，而且是将二级缓冲全部写出，减少了调用的次数，所以提高了效率。

**●FileWriterWithEncoding**

从这个类的名称已经可以很清楚的知道它的作用了。在JDK自带的FileWriter中，是无法设置encoding的，这个类允许我们采用默认或者指定的encoding，以字符的形式写到文件。为什么这个类可以改变字符嗯？

原理很简单：无非使用了OutputStreamWriter。而且这个类并不是继承与FileWriter，而是直接继承于Writer。

```
        OutputStream stream = null;
        Writer writer = null;
        try {
            stream = new FileOutputStream(file, append);
            if (encoding instanceof Charset) {
                writer = new OutputStreamWriter(stream, (Charset)encoding);
            } else if (encoding instanceof CharsetEncoder) {
                writer = new OutputStreamWriter(stream, (CharsetEncoder)encoding);
            } else {
                writer = new OutputStreamWriter(stream, (String)encoding);
```

```
        }
...
```

剩下的各种write方法，无非就是decorator模式而已。

●LockableFileWriter

使用“文件锁”而非“对象锁”来限制多线程环境下的写动作。这个类采用在JDK默认的系统临时目录下写文件：java.io.tmpdir属性。而且允许我们设置encoding。

```
/**
     * Constructs a LockableFileWriter with a file encoding.
     *
     * @param file  the file to write to, not null
     * @param encoding  the encoding to use, null means platform default
     * @param append  true if content should be appended, false to overwrite
     * @param lockDir  the directory in which the lock file should be held
     * @throws NullPointerException if the file is null
     * @throws IOException in case of an I/O error
     */
    public LockableFileWriter(File file, String encoding, boolean append,
            String lockDir) throws IOException {
        super();
        // init file to create/append
        file = file.getAbsoluteFile();
      if (file.getParentFile() != null) {
            FileUtils.forceMkdir(file.getParentFile());
        }
      if (file.isDirectory()) {
            throw new IOException("File specified is a directory");
        }

        // init lock file
      if (lockDir == null) {
            lockDir = System.getProperty("java.io.tmpdir");
        }
        File lockDirFile = new File(lockDir);
        FileUtils.forceMkdir(lockDirFile);
        testLockDir(lockDirFile);
        lockFile = new File(lockDirFile, file.getName() + LCK);

        // check if locked
        createLock();

        // init wrapped writer
        out = initWriter(file, encoding, append);
    }
```

首先创建一个用于存放lock文件的目录，位于系统临时目录下。

接下来创建一个位于该目录下的名为xxxLCK的文件(锁文件)。

然后创建锁

最后则是初始化writer

显然我们关心的是如何创建这个锁，以及如何在写期间进行锁。首先来看创建锁的过程

```
private void createLock() throws IOException {
        synchronized (LockableFileWriter.class) {
            if (!lockFile.createNewFile()) {
                throw new IOException("Can't write file, lock " +
                        lockFile.getAbsolutePath() + " exists");
            }
```

```
            lockFile.deleteOnExit();
        }
    }
```

注意这里的deleteOnExit()方法很重要，它告诉JVM：当JV退出是要删除该文件。否则磁盘上将有无数无用的临时锁文件。

下面的问题则是如何实现锁呢？呵呵~~。还是回到这个上面这个类的构造方法吧，我们看到在构造这个LockableFileWriter时，会调用createLock()这个方法，而这个方法如果发现文件已经创建/被其它流引用时，会抛出一个IOException。于是创建不成功，也就无法继续后续的write操作了。

那么如果第一个进程创建了锁之后就不释放，那么后续的进程岂不是无法写了，于是在这个类的close方法中有这样一句代码：

```
public void close() throws IOException {
        try {
            out.close();
        } finally {
            lockFile.delete();
        }
    }
```

每个进程在完成数据的写动作后，必须调用close()方法，于是锁文件被删除，锁被解除。相比于JDK中自带Writer使用的object锁(synchronized(object))，这个方法确实要更加简便和高效。这个类当初就是被设计来替换掉原始的FileWriter的。

**但切记：一定要在结尾处调用close()方法，否则无法解锁。而且这里没有AutoCloseOutputStream这样的类哦！**

---

生活就像打牌，不是要抓一手好牌，而是要尽力打好一手烂牌。