Java IO - 源码: OutputStream

主要从JDK源码角度分析 OutputStream。

OutputStream 类实现关系

OutputStream 抽象类

OutputStream 类

```
public abstract void write(int b)
// 写入一个字节，可以看到这里的参数是一个 int 类型，对应上面的读方法，int 类型的 32 位，只有低 8 位才写入，高 24 位将舍弃。

public void write(byte b[])
// 将数组中的所有字节写入，和上面对应的 read() 方法类似，实际调用的也是下面的方法。

public void write(byte b[], int off, int len)
// 将 byte 数组从 off 位置开始，len 长度的字节写入

public void flush()
// 强制刷新，将缓冲中的数据写入

public void close()
// 关闭输出流，流被关闭后就不能再输出数据了
```

源码实现

FilterOutputStream

```
/**
 * This class is the superclass of all classes that filter output
 * streams. These streams sit on top of an already existing output
 * stream (the <i>underlying</i> output stream) which it uses as its
 * basic sink of data, but possibly transforming the data along the
 * way or providing additional functionality.
 * <p>
 * The class <code>FilterOutputStream</code> itself simply overrides
 * all methods of <code>OutputStream</code> with versions that pass
 * all requests to the underlying output stream. Subclasses of
 * <code>FilterOutputStream</code> may further override some of these
 * methods as well as provide additional methods and fields.
```

```java
 *
 * @author  Jonathan Payne
 * @since   JDK1.0
 */
public
class FilterOutputStream extends OutputStream {
    /**
     * The underlying output stream to be filtered.
     */
    protected OutputStream out;

    /**
     * Creates an output stream filter built on top of the specified
     * underlying output stream.
     *
     * @param   out   the underlying output stream to be assigned to
     *                the field <tt>this.out</tt> for later use, or
     *                <code>null</code> if this instance is to be
     *                created without an underlying stream.
     */
    public FilterOutputStream(OutputStream out) {
        this.out = out;
    }

    /**
     * Writes the specified <code>byte</code> to this output stream.
     * <p>
     * The <code>write</code> method of <code>FilterOutputStream</code>
     * calls the <code>write</code> method of its underlying output stream,
     * that is, it performs <tt>out.write(b)</tt>.
     * <p>
     * Implements the abstract <tt>write</tt> method of <tt>OutputStream</tt>.
     *
     * @param      b   the <code>byte</code>.
     * @exception  IOException  if an I/O error occurs.
     */
    public void write(int b) throws IOException {
        out.write(b);
    }

    /**
     * Writes <code>b.length</code> bytes to this output stream.
     * <p>
     * The <code>write</code> method of <code>FilterOutputStream</code>
     * calls its <code>write</code> method of three arguments with the
     * arguments <code>b</code>, <code>0</code>, and
     * <code>b.length</code>.
     * <p>
     * Note that this method does not call the one-argument
     * <code>write</code> method of its underlying stream with the single
     * argument <code>b</code>.
     *
     * @param      b   the data to be written.
     * @exception  IOException  if an I/O error occurs.
     * @see        java.io.FilterOutputStream#write(byte[], int, int)
     */
    public void write(byte b[]) throws IOException {
        write(b, 0, b.length);
    }

    /**
```

```
     * Writes <code>len</code> bytes from the specified
     * <code>byte</code> array starting at offset <code>off</code> to
     * this output stream.
     * <p>
     * The <code>write</code> method of <code>FilterOutputStream</code>
     * calls the <code>write</code> method of one argument on each
     * <code>byte</code> to output.
     * <p>
     * Note that this method does not call the <code>write</code> method
     * of its underlying input stream with the same arguments. Subclasses
     * of <code>FilterOutputStream</code> should provide a more efficient
     * implementation of this method.
     *
     * @param      b     the data.
     * @param      off   the start offset in the data.
     * @param      len   the number of bytes to write.
     * @exception  IOException  if an I/O error occurs.
     * @see        java.io.FilterOutputStream#write(int)
     */
    public void write(byte b[], int off, int len) throws IOException {
        if ((off | len | (b.length - (len + off)) | (off + len)) < 0)
            throw new IndexOutOfBoundsException();

        for (int i = 0 ; i < len ; i++) {
            write(b[off + i]);
        }
    }

    /**
     * Flushes this output stream and forces any buffered output bytes
     * to be written out to the stream.
     * <p>
     * The <code>flush</code> method of <code>FilterOutputStream</code>
     * calls the <code>flush</code> method of its underlying output stream.
     *
     * @exception  IOException  if an I/O error occurs.
     * @see        java.io.FilterOutputStream#out
     */
    public void flush() throws IOException {
        out.flush();
    }

    /**
     * Closes this output stream and releases any system resources
     * associated with the stream.
     * <p>
     * The <code>close</code> method of <code>FilterOutputStream</code>
     * calls its <code>flush</code> method, and then calls the
     * <code>close</code> method of its underlying output stream.
     *
     * @exception  IOException  if an I/O error occurs.
     * @see        java.io.FilterOutputStream#flush()
     * @see        java.io.FilterOutputStream#out
     */
    @SuppressWarnings("try")
    public void close() throws IOException {
        try (OutputStream ostream = out) {
            flush();
        }
    }
}
```

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 <code>toByteArray()</code> and
 * <code>toString()</code>.
 * <p>
 * Closing a <tt>ByteArrayOutputStream</tt> has no effect. The methods in
 * this class can be called after the stream has been closed without
 * generating an <tt>IOException</tt>.
 *
 * @author  Arthur van Hoff
 * @since   JDK1.0
 */

public class ByteArrayOutputStream extends OutputStream {

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

    /**
     * The number of valid bytes in the buffer.
     */
    protected int count;

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

    /**
     * Creates a new byte array output stream, with a buffer capacity of
     * the specified size, in bytes.
     *
     * @param   size   the initial size.
     * @exception  IllegalArgumentException if size is negative.
     */
    public ByteArrayOutputStream(int size) {
        if (size < 0) {
            throw new IllegalArgumentException("Negative initial size: "
                                               + size);
        }
        buf = new byte[size];
    }

    /**
     * Increases the capacity if necessary to ensure that it can hold
     * at least the number of elements specified by the minimum
     * capacity argument.
     *
     * @param minCapacity the desired minimum capacity
     * @throws OutOfMemoryError if {@code minCapacity < 0}.  This is
     * interpreted as a request for the unsatisfiably large capacity
     * {@code (long) Integer.MAX_VALUE + (minCapacity - Integer.MAX_VALUE)}.
     */
    private void ensureCapacity(int minCapacity) {
        // overflow-conscious code
        if (minCapacity - buf.length > 0)
            grow(minCapacity);
    }

    /**
     * The maximum size of array to allocate.
     * Some VMs reserve some header words in an array.
     * Attempts to allocate larger arrays may result in
     * OutOfMemoryError: Requested array size exceeds VM limit
     */
    private static final int MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;

    /**
     * Increases the capacity to ensure that it can hold at least the
     * number of elements specified by the minimum capacity argument.
     *
     * @param minCapacity the desired minimum capacity
     */
    private void grow(int minCapacity) {
        // overflow-conscious code
        int oldCapacity = buf.length;
        int newCapacity = oldCapacity << 1;
        if (newCapacity - minCapacity < 0)
            newCapacity = minCapacity;
        if (newCapacity - MAX_ARRAY_SIZE > 0)
            newCapacity = hugeCapacity(minCapacity);
        buf = Arrays.copyOf(buf, newCapacity);
    }

    private static int hugeCapacity(int minCapacity) {
        if (minCapacity < 0) // overflow
            throw new OutOfMemoryError();
        return (minCapacity > MAX_ARRAY_SIZE) ?
            Integer.MAX_VALUE :
            MAX_ARRAY_SIZE;
    }

    /**
     * Writes the specified byte to this byte array output stream.
     *
     * @param   b   the byte to be written.
     */
    public synchronized void write(int b) {
        ensureCapacity(count + 1);
        buf[count] = (byte) b;
        count += 1;
    }

    /**
     * Writes <code>len</code> bytes from the specified byte array
     * starting at offset <code>off</code> to this byte array output stream.
     *
     * @param   b     the data.
     * @param   off   the start offset in the data.
     * @param   len   the number of bytes to write.
     */
    public synchronized void write(byte b[], int off, int len) {
        if ((off < 0) || (off > b.length) || (len < 0) ||
            ((off + len) - b.length > 0)) {
            throw new IndexOutOfBoundsException();
        }
        ensureCapacity(count + len);
        System.arraycopy(b, off, buf, count, len);
        count += len;
    }

    /**
     * Writes the complete contents of this byte array output stream to
     * the specified output stream argument, as if by calling the output
     * stream's write method using <code>out.write(buf, 0, count)</code>.
     *
     * @param      out   the output stream to which to write the data.
     * @exception  IOException  if an I/O error occurs.
     */
    public synchronized void writeTo(OutputStream out) throws IOException {
        out.write(buf, 0, count);
    }

    /**
     * Resets the <code>count</code> field of this byte array output
     * stream to zero, so that all currently accumulated output in the
     * output stream is discarded. The output stream can be used again,
     * reusing the already allocated buffer space.
     *
     * @see     java.io.ByteArrayInputStream#count
     */
    public synchronized void reset() {
        count = 0;
    }

    /**
     * Creates a newly allocated byte array. Its size is the current
     * size of this output stream and the valid contents of the buffer
     * have been copied into it.
     *
     * @return  the current contents of this output stream, as a byte array.
     * @see     java.io.ByteArrayOutputStream#size()
     */
    public synchronized byte toByteArray()[] {
        return Arrays.copyOf(buf, count);
    }

    /**
     * Returns the current size of the buffer.
     *
     * @return  the value of the <code>count</code> field, which is the number
     *          of valid bytes in this output stream.
     * @see     java.io.ByteArrayOutputStream#count
     */
    public synchronized int size() {
        return count;
    }

    /**
     * Converts the buffer's contents into a string decoding bytes using the
     * platform's default character set. The length of the new <tt>String</tt>
     * is a function of the character set, and hence may not be equal to the
     * size of the buffer.
```

```java
     *
     * <p> This method always replaces malformed-input and unmappable-character
     * sequences with the default replacement string for the platform's
     * default character set. The {@linkplain java.nio.charset.CharsetDecoder}
     * class should be used when more control over the decoding process is
     * required.
     *
     * @return String decoded from the buffer's contents.
     * @since  JDK1.1
     */
    public synchronized String toString() {
        return new String(buf, 0, count);
    }

    /**
     * Converts the buffer's contents into a string by decoding the bytes using
     * the named {@link java.nio.charset.Charset charset}. The length of the new
     * <tt>String</tt> is a function of the charset, and hence may not be equal
     * to the length of the byte array.
     *
     * <p> This method always replaces malformed-input and unmappable-character
     * sequences with this charset's default replacement string. The {@link
     * java.nio.charset.CharsetDecoder} class should be used when more control
     * over the decoding process is required.
     *
     * @param      charsetName  the name of a supported
     *             {@link java.nio.charset.Charset charset}
     * @return     String decoded from the buffer's contents.
     * @exception  UnsupportedEncodingException
     *             If the named charset is not supported
     * @since      JDK1.1
     */
    public synchronized String toString(String charsetName)
        throws UnsupportedEncodingException
    {
        return new String(buf, 0, count, charsetName);
    }

    /**
     * Creates a newly allocated string. Its size is the current size of
     * the output stream and the valid contents of the buffer have been
     * copied into it. Each character <i>c</i> in the resulting string is
     * constructed from the corresponding element <i>b</i> in the byte
     * array such that:
     * <blockquote><pre>
     *     c == (char)(((hibyte &amp; 0xff) &lt;&lt; 8) | (b &amp; 0xff))
     * </pre></blockquote>
     *
     * @deprecated This method does not properly convert bytes into characters.
     * As of JDK&nbsp;1.1, the preferred way to do this is via the
     * <code>toString(String enc)</code> method, which takes an encoding-name
     * argument, or the <code>toString()</code> method, which uses the
     * platform's default character encoding.
     *
     * @param      hibyte    the high byte of each resulting Unicode character.
     * @return     the current contents of the output stream, as a string.
     * @see        java.io.ByteArrayOutputStream#size()
     * @see        java.io.ByteArrayOutputStream#toString(String)
     * @see        java.io.ByteArrayOutputStream#toString()
     */
    @Deprecated
    public synchronized String toString(int hibyte) {
        return new String(buf, hibyte, 0, count);
    }

    /**
     * Closing a <tt>ByteArrayOutputStream</tt> has no effect. The methods in
     * this class can be called after the stream has been closed without
     * generating an <tt>IOException</tt>.
     */
    public void close() throws IOException {
    }

}
```

BufferedOutputStream

```
/**
 * The class implements a buffered output stream. By setting up such
 * an output stream, an application can write bytes to the underlying
 * output stream without necessarily causing a call to the underlying
 * system for each byte written.
 *
 * @author  Arthur van Hoff
 * @since   JDK1.0
 */
public
class BufferedOutputStream extends FilterOutputStream {
    /**
     * The internal buffer where data is stored.
     */
    protected byte buf[];

    /**
     * The number of valid bytes in the buffer. This value is always
     * in the range <tt>0</tt> through <tt>buf.length</tt>; elements
     * <tt>buf[0]</tt> through <tt>buf[count-1]</tt> contain valid
     * byte data.
     */
    protected int count;

    /**
     * Creates a new buffered output stream to write data to the
     * specified underlying output stream.
     *
     * @param   out   the underlying output stream.
     */
    public BufferedOutputStream(OutputStream out) {
        this(out, 8192);
    }

    /**
     * Creates a new buffered output stream to write data to the
     * specified underlying output stream with the specified buffer
     * size.
     *
     * @param   out    the underlying output stream.
     * @param   size   the buffer size.
```

```java
     * @exception IllegalArgumentException if size &lt;= 0.
     */
    public BufferedOutputStream(OutputStream out, int size) {
        super(out);
        if (size <= 0) {
            throw new IllegalArgumentException("Buffer size <= 0");
        }
        buf = new byte[size];
    }

    /** Flush the internal buffer */
    private void flushBuffer() throws IOException {
        if (count > 0) {
            out.write(buf, 0, count);
            count = 0;
        }
    }

    /**
     * Writes the specified byte to this buffered output stream.
     *
     * @param      b   the byte to be written.
     * @exception  IOException  if an I/O error occurs.
     */
    public synchronized void write(int b) throws IOException {
        if (count >= buf.length) {
            flushBuffer();
        }
        buf[count++] = (byte)b;
    }

    /**
     * Writes <code>len</code> bytes from the specified byte array
     * starting at offset <code>off</code> to this buffered output stream.
     *
     * <p> Ordinarily this method stores bytes from the given array into this
     * stream's buffer, flushing the buffer to the underlying output stream as
     * needed.  If the requested length is at least as large as this stream's
     * buffer, however, then this method will flush the buffer and write the
     * bytes directly to the underlying output stream.  Thus redundant
     * <code>BufferedOutputStream</code>s will not copy data unnecessarily.
     *
     * @param      b     the data.
     * @param      off   the start offset in the data.
     * @param      len   the number of bytes to write.
     * @exception  IOException  if an I/O error occurs.
     */
    public synchronized void write(byte b[], int off, int len) throws IOException {
        if (len >= buf.length) {
            /* If the request length exceeds the size of the output buffer,
               flush the output buffer and then write the data directly.
               In this way buffered streams will cascade harmlessly. */
            flushBuffer();
            out.write(b, off, len);
            return;
        }
        if (len > buf.length - count) {
            flushBuffer();
        }
        System.arraycopy(b, off, buf, count, len);
        count += len;
    }

    /**
     * Flushes this buffered output stream. This forces any buffered
     * output bytes to be written out to the underlying output stream.
     *
     * @exception  IOException  if an I/O error occurs.
     * @see        java.io.FilterOutputStream#out
     */
    public synchronized void flush() throws IOException {
        flushBuffer();
        out.flush();
    }
}
```

PipedOutputStream

```java
/**
 * A piped output stream can be connected to a piped input stream
 * to create a communications pipe. The piped output stream is the
 * sending end of the pipe. Typically, data is written to a
 * <code>PipedOutputStream</code> object by one thread and data is
 * read from the connected <code>PipedInputStream</code> by some
 * other thread. Attempting to use both objects from a single thread
 * is not recommended as it may deadlock the thread.
 * The pipe is said to be <a name=BROKEN> <i>broken</i> </a> if a
 * thread that was reading data bytes from the connected piped input
 * stream is no longer alive.
 *
 * @author  James Gosling
 * @see     java.io.PipedInputStream
 * @since   JDK1.0
 */
public
class PipedOutputStream extends OutputStream {

        /* REMIND: identification of the read and write sides needs to be
           more sophisticated.  Either using thread groups (but what about
           pipes within a thread?) or using finalization (but it may be a
           long time until the next GC). */
    private PipedInputStream sink;

    /**
     * Creates a piped output stream connected to the specified piped
     * input stream. Data bytes written to this stream will then be
     * available as input from <code>snk</code>.
     *
     * @param      snk   The piped input stream to connect to.
     * @exception  IOException  if an I/O error occurs.
     */
    public PipedOutputStream(PipedInputStream snk)  throws IOException {
        connect(snk);
    }

    /**
     * Creates a piped output stream that is not yet connected to a
     * piped input stream. It must be connected to a piped input stream,
     * either by the receiver or the sender, before being used.
     *
     * @see     java.io.PipedInputStream#connect(java.io.PipedOutputStream)
     * @see     java.io.PipedOutputStream#connect(java.io.PipedInputStream)
     */
    public PipedOutputStream() {
    }

    /**
     * Connects this piped output stream to a receiver. If this object
     * is already connected to some other piped input stream, an
     * <code>IOException</code> is thrown.
     * <p>
     * If <code>snk</code> is an unconnected piped input stream and
     * <code>src</code> is an unconnected piped output stream, they may
     * be connected by either the call:
     * <blockquote><pre>
     * src.connect(snk)</pre></blockquote>
     * or the call:
     * <blockquote><pre>
     * snk.connect(src)</pre></blockquote>
     * The two calls have the same effect.
     *
     * @param      snk   the piped input stream to connect to.
     * @exception  IOException  if an I/O error occurs.
     */
    public synchronized void connect(PipedInputStream snk) throws IOException {
        if (snk == null) {
            throw new NullPointerException();
        } else if (sink != null || snk.connected) {
            throw new IOException("Already connected");
        }
        sink = snk;
        snk.in = -1;
        snk.out = 0;
        snk.connected = true;
    }

    /**
     * Writes the specified <code>byte</code> to the piped output stream.
     * <p>
     * Implements the <code>write</code> method of <code>OutputStream</code>.
     *
     * @param      b   the <code>byte</code> to be written.
     * @exception IOException if the pipe is <a href=#BROKEN> broken</a>,
     *          {@link #connect(java.io.PipedInputStream) unconnected},
     *          closed, or if an I/O error occurs.
     */
    public void write(int b)  throws IOException {
        if (sink == null) {
            throw new IOException("Pipe not connected");
        }
        sink.receive(b);
    }

    /**
     * Writes <code>len</code> bytes from the specified byte array
     * starting at offset <code>off</code> to this piped output stream.
     * This method blocks until all the bytes are written to the output
     * stream.
     *
     * @param      b     the data.
     * @param      off   the start offset in the data.
     * @param      len   the number of bytes to write.
     * @exception IOException if the pipe is <a href=#BROKEN> broken</a>,
     *          {@link #connect(java.io.PipedInputStream) unconnected},
     *          closed, or if an I/O error occurs.
     */
    public void write(byte b[], int off, int len) throws IOException {
        if (sink == null) {
            throw new IOException("Pipe not connected");
        } else if (b == null) {
            throw new NullPointerException();
        } else if ((off < 0) || (off > b.length) || (len < 0) ||
                   ((off + len) > b.length) || ((off + len) < 0)) {
            throw new IndexOutOfBoundsException();
        } else if (len == 0) {
            return;
        }
        sink.receive(b, off, len);
    }

    /**
     * Flushes this output stream and forces any buffered output bytes
     * to be written out.
     * This will notify any readers that bytes are waiting in the pipe.
     *
     * @exception IOException if an I/O error occurs.
     */
    public synchronized void flush() throws IOException {
        if (sink != null) {
            synchronized (sink) {
                sink.notifyAll();
            }
        }
    }

    /**
     * Closes this piped output stream and releases any system resources
     * associated with this stream. This stream may no longer be used for
     * writing bytes.
     *
     * @exception  IOException  if an I/O error occurs.
     */
    public void close()  throws IOException {
        if (sink != null) {
            sink.receivedLast();
        }
    }
}
```