在 Java 开发中，控制台输出仍是一个重要的工具，但默认的控制台输出有着各种各样的局限。本文介绍如何用 Java 管道流截取控制台输出，分析管道流应用中应该注意的问题，提供了截取 Java 程序和非 Java 程序控制台输出的实例。

即使在图形用户界面占统治地位的今天，控制台输出仍旧在 Java 程序中占有重要地位。控制台不仅是 Java 程序默认的堆栈跟踪和错误信息输出窗口，而且还是一种实用的调试工具（特别是对习惯于使用 println()的人来说）。然而，控制台窗口有着许多局限。例如在 Windows 9x 平台上，DOS 控制台只能容纳 50 行输出。如果 Java 程序一次性向控制台输出大量内容，要查看这些内容就很困难了。

对于使用 javaw 这个启动程序的开发者来说，控制台窗口尤其宝贵。因为用 javaw 启动 java 程序时，根本不会有控制台窗口出现。如果程序遇到了问题并抛出异常，根本无法查看 Java 运行时环境写入到 System.out 或 System.err 的调用堆栈跟踪信息。为了捕获堆栈信息，一些人采取了用 try/catch()块封装 main()的方式，但这种方式不一定总是有效，在 Java 运行时的某些时刻，一些描述性错误信息会在抛出异常之前被写入 System.out 和 System.err；除非能够监测这两个控制台流，否则这些信息就无法看到。

因此，有些时候检查 Java 运行时环境（或第三方程序）写入到控制台流的数据并采取合适的操作是十分必要的。本文讨论的主题之一就是创建这样一个输入流，从这个输入流中可以读入以前写入 Java 控制台流（或任何其他程序的输出流）的数据。我们可以想象写入到输出流的数据立即以输入的形式“回流”到了 Java 程序。

本文的目标是设计一个基于 Swing 的文本窗口显示控制台输出。在此期间，我们还将讨论一些和 Java 管道流（PipedInputStream 和 PipedOutputStream）有关的重要注意事项。

图一显示了用来截取和显示控制台文本输出的 Java 程序，用户界面的核心是一个 JTextArea。最后，我们还要创建一个能够捕获和显示其他程序（可以是非 Java 的程序）控制台输出的简单程序。

一、Java 管道流

要在文本框中显示控制台输出，我们必须用某种方法“截取”控制台流。换句话说，我们要有一种高效地读取写入到 System.out 和 System.err 所有内容的方法。如果你熟悉 Java 的管道流 PipedInputStream 和 PipedOutputStream，就会相信我们已经拥有最有效的工具。

写入到 PipedOutputStream 输出流的数据可以从对应的 PipedInputStream 输入流读取。Java 的管道流极大地方便了我们截取控制台输出。Listing 1 显示了一种非常简单的截取控制台输出方案。

【Listing 1：用管道流截取控制台输出】

```
PipedInputStream pipedIS = new PipedInputStream();
PipedOutputStream pipedOS = new PipedOutputStream();
try {
   pipedOS.connect(pipedIS);
}
catch(IOException e) {
   System.err.println("连接失败");
   System.exit(1);
}
PrintStream ps = new PrintStream(pipedOS);
System.setOut(ps);
System.setErr(ps);
```

可以看到，这里的代码极其简单。我们只是建立了一个 PipedInputStream，把它设置为所有写入控制台流的数据的最终目的地。所有写入到控制台流的数据都被转到

PipedOutputStream，这样，从相应的 PipedInputStream 读取就可以迅速地截获所有写入控制台流的数据。接下来的事情似乎只剩下在 Swing JTextArea 中显示从 pipedIS 流读取的数据，得到一个能够在文本框中显示控制台输出的程序。遗憾的是，在使用 Java 管道流时有一些重要的注意事项。只有认真对待所有这些注意事项才能保证 Listing 1 的代码稳定地运行。下面我们来看第一个注意事项。

1.1 注意事项一

PipedInputStream 运用的是一个 1024 字节固定大小的循环缓冲区。写入 PipedOutputStream 的数据实际上保存到对应的 PipedInputStream 的内部缓冲区。从 PipedInputStream 执行读操作时，读取的数据实际上来自这个内部缓冲区。如果对应的 PipedInputStream 输入缓冲区已满，任何企图写入 PipedOutputStream 的线程都将被阻塞。而且这个写操作线程将一直阻塞，直至出现读取 PipedInputStream 的操作从缓冲区删除数据。

这意味着，向 PipedOutputStream 写数据的线程不应该是负责从对应 PipedInputStream 读取数据的唯一线程。从图二可以清楚地看出这里的问题所在：假设线程 t 是负责从 PipedInputStream 读取数据的唯一线程；另外，假定 t 企图在一次对 PipedOutputStream 的 write()方法的调用中向对应的 PipedOutputStream 写入 2000 字节的数据。在 t 线程阻塞之前，它最多能够写入 1024 字节的数据（PipedInputStream 内部缓冲区的大小）。然而，一旦 t 被阻塞，读取 PipedInputStream 的操作就再也不会出现，因为 t 是唯一读取 PipedInputStream 的线程。这样，t 线程已经完全被阻塞，同时，所有其他试图向 PipedOutputStream 写入数据的线程也将遇到同样的情形。

这并不意味着在一次 write()调用中不能写入多于 1024 字节的数据。但应当保证，在写入数

据的同时，有另一个线程从 PipedInputStream 读取数据。

Listing 2 示范了这个问题。这个程序用一个线程交替地读取 PipedInputStream 和写入 PipedOutputStream。每次调用 write()向 PipedInputStream 的缓冲区写入 20 字节，每次调用 read()只从缓冲区读取并删除 10 个字节。内部缓冲区最终会被写满，导致写操作阻塞。由于我们用同一个线程执行读、写操作，一旦写操作被阻塞，就不能再从 PipedInputStream 读取数据。

【Listing 2：用同一个线程执行读/写操作导致线程阻塞】

```
import java.io.*;
public class Listing2 {
    static PipedInputStream pipedIS = new PipedInputStream();
    static PipedOutputStream pipedOS =
        new PipedOutputStream();
    public static void main(String[] a){
        try {
            pipedIS.connect(pipedOS);
        }
        catch(IOException e) {
            System.err.println("连接失败");
                System.exit(1);
            }
        byte[] inArray    = new byte[10];
        byte[] outArray = new byte[20];
        int bytesRead = 0;
        try {
            // 向 pipedOS 发送 20 字节数据
            pipedOS.write(outArray, 0, 20);
            System.out.println("      已发送 20 字节...");
           // 在每一次循环迭代中，读入 10 字节
           // 发送 20 字节
            bytesRead = pipedIS.read(inArray, 0, 10);
            int i=0;
            while(bytesRead != -1) {
                pipedOS.write(outArray, 0, 20);
```

```
                System.out.println("      已发送 20 字节..."+i);
                i++;
                bytesRead = pipedIS.read(inArray, 0, 10);
            }
        }
        catch(IOException e) {
                System.err.println("读取 pipedIS 时出现错误: " + e);
                System.exit(1);
        }
    } // main()
}
```

只要把读/写操作分开到不同的线程，Listing 2 的问题就可以轻松地解决。Listing 3 是 Listing 2 经过修改后的版本，它在一个单独的线程中执行写入 PipedOutputStream 的操作（和读取线程不同的线程）。为证明一次写入的数据可以超过 1024 字节，我们让写操作线程每次调用 PipedOutputStream 的 write()方法时写入 2000 字节。那么，在 startWriterThread()方法中创建的线程是否会阻塞呢？按照 Java 运行时线程调度机制，它当然会阻塞。写操作在阻塞之前实际上最多只能写入 1024 字节的有效载荷（即 PipedInputStream 缓冲区的大小）。但这并不会成为问题，因为主线程（main）很快就会从 PipedInputStream 的循环缓冲区读取数据，空出缓冲区空间。最终，写操作线程会从上一次中止的地方重新开始，写入 2000 字节有效载荷中的剩余部分。

【Listing 3：把读/写操作分开到不同的线程】

```
import java.io.*;
public class Listing3 {
    static PipedInputStream pipedIS =
        new PipedInputStream();
    static PipedOutputStream pipedOS =
        new PipedOutputStream();
    public static void main(String[] args) {
        try {
```

```
            pipedIS.connect(pipedOS);
        }
        catch(IOException e) {
            System.err.println("连接失败");
            System.exit(1);
        }
        byte[] inArray = new byte[10];
        int bytesRead = 0;
        // 启动写操作线程
        startWriterThread();
        try {
            bytesRead = pipedIS.read(inArray, 0, 10);
            while(bytesRead != -1) {
                System.out.println("已经读取" +
                    bytesRead + "字节...");
                bytesRead = pipedIS.read(inArray, 0, 10);
            }
        }
        catch(IOException e) {
            System.err.println("读取输入错误.");
            System.exit(1);
        }
    } // main()
    // 创建一个独立的线程
    // 执行写入 PipedOutputStream 的操作
    private static void startWriterThread() {
        new Thread(new Runnable() {
            public void run() {
                byte[] outArray = new byte[2000];
                while(true) { // 无终止条件的循环
                    try {
                        // 在该线程阻塞之前，有最多 1024 字节的数据被写入
                        pipedOS.write(outArray, 0, 2000);
                    }
                    catch(IOException e) {
                        System.err.println("写操作错误");
                        System.exit(1);
                    }
                    System.out.println("      已经发送 2000 字节...");
                }
            }
```

```
        }).start();
    } // startWriterThread()
} // Listing3
```

也许我们不能说这个问题是 Java 管道流设计上的缺陷，但在应用管道流时，它是一个必须密切注意的问题。下面我们来看看第二个更重要（更危险的）问题。

1.2 注意事项二

从 PipedInputStream 读取数据时，如果符合下面三个条件，就会出现 IOException 异常：

1. 试图从 PipedInputStream 读取数据，

2. PipedInputStream 的缓冲区为“空”（即不存在可读取的数据），

3. 最后一个向 PipedOutputStream 写数据的线程不再活动（通过 Thread.isAlive()检测）。

这是一个很微妙的时刻，同时也是一个极其重要的时刻。假定有一个线程 w 向 PipedOutputStream 写入数据；另一个线程 r 从对应的 PipedInputStream 读取数据。下面一系列的事件将导致 r 线程在试图读取 PipedInputStream 时遇到 IOException 异常：

1. w 向 PipedOutputStream 写入数据。

2. w 结束（w.isAlive()返回 false）。

3. r 从 PipedInputStream 读取 w 写入的数据，清空 PipedInputStream 的缓冲区。

4. r 试图再次从 PipedInputStream 读取数据。这时 PipedInputStream 的缓冲区已经为空，而且 w 已经结束，从而导致在读操作执行时出现 IOException 异常。

构造一个程序示范这个问题并不困难，只需从 Listing 3 的 startWriterThread()方法中，删除 while(true)条件。这个改动阻止了执行写操作的方法循环执行，使得执行写操作的方法在一次写入操作之后就结束运行。如前所述，此时主线程试图读取 PipedInputStraem 时，就会

遇到一个 IOException 异常。

这是一种比较少见的情况，而且不存在直接修正它的方法。请不要通过从管道流派生子类的方法修正该问题——在这里使用继承是完全不合适的。而且，如果 Sun 以后改变了管道流的实现方法，现在所作的修改将不再有效。

最后一个问题和第二个问题很相似，不同之处在于，它在读线程（而不是写线程）结束时产生 IOException 异常。

1.3 注意事项三

如果一个写操作在 PipedOutputStream 上执行，同时最近从对应 PipedInputStream 读取的线程已经不再活动（通过 Thread.isAlive()检测），则写操作将抛出一个 IOException 异常。假定有两个线程 w 和 r，w 向 PipedOutputStream 写入数据，而 r 则从对应的 PipedInputStream 读取。下面一系列的事件将导致 w 线程在试图写入 PipedOutputStream 时遇到 IOException 异常：

1. 写操作线程 w 已经创建，但 r 线程还不存在。

2. w 向 PipedOutputStream 写入数据。

3. 读线程 r 被创建，并从 PipedInputStream 读取数据。

4. r 线程结束。

5. w 企图向 PipedOutputStream 写入数据，发现 r 已经结束，抛出 IOException 异常。

实际上，这个问题不象第二个问题那样棘手。和多个读线程/单个写线程的情况相比，也许在应用中有一个读线程（作为响应请求的服务器）和多个写线程（发出请求）的情况更为常见。

1.4 解决问题

要防止管道流前两个局限所带来的问题，方法之一是用一个 ByteArrayOutputStream 作为代理或替代 PipedOutputStream。 Listing 4 显示了一个 LoopedStreams 类，它用一个 ByteArrayOutputStream 提供和 Java 管道流类似的功能，但不会出现死锁和 IOException 异常。这个类的内部仍旧使用管道流，但隔离了本文介绍的前两个问题。我们先来看看这个类的公用方法（参见图 3）。构造函数很简单，它连接管道流，然后调用 startByteArrayReaderThread()方法（稍后再讨论该方法）。getOutputStream()方法返回一个 OutputStream（具体地说，是一个 ByteArrayOutputStream）用以替代 PipedOutputStream。写入该 OutputStream 的数据最终将在 getInputStream()方法返回的流中作为输入出现。和使用 PipedOutputStream 的情形不同，向 ByteArrayOutputStream 写入数据的线程的激活、写数据、结束不会带来负面效果。

【Listing 4：防止管道流应用中出现的常见问题】

```
import java.io.*;
public class LoopedStreams {
    private PipedOutputStream pipedOS =
        new PipedOutputStream();
    private boolean keepRunning = true;
    private ByteArrayOutputStream byteArrayOS =
        new ByteArrayOutputStream() {
        public void close() {
            keepRunning = false;
            try {
                super.close();
                pipedOS.close();
            }
            catch(IOException e) {
                // 记录错误或其他处理
                // 为简单计，此处我们直接结束
                System.exit(1);
```

```
            }
        }
    };
    private PipedInputStream pipedIS = new PipedInputStream() {
        public void close() {
            keepRunning = false;
            try    {
                super.close();
            }
            catch(IOException e) {
                // 记录错误或其他处理
                // 为简单计，此处我们直接结束
                System.exit(1);
            }
        }
    };
    public LoopedStreams() throws IOException {
        pipedOS.connect(pipedIS);
        startByteArrayReaderThread();
    } // LoopedStreams()
    public InputStream getInputStream() {
        return pipedIS;
    } // getInputStream()
    public OutputStream getOutputStream() {
        return byteArrayOS;
    } // getOutputStream()
    private void startByteArrayReaderThread() {
        new Thread(new Runnable() {
            public void run() {
                while(keepRunning) {
                    // 检查流里面的字节数
                    if(byteArrayOS.size() > 0) {
                        byte[] buffer = null;
                        synchronized(byteArrayOS) {
                            buffer = byteArrayOS.toByteArray();
                            byteArrayOS.reset(); // 清除缓冲区
                        }
                        try {
                            // 把提取到的数据发送给 PipedOutputStream
                            pipedOS.write(buffer, 0, buffer.length);
                        }
```

```
                    catch(IOException e) {
                        // 记录错误或其他处理
                        // 为简单计，此处我们直接结束
                        System.exit(1);
                    }
                }
                else // 没有数据可用，线程进入睡眠状态
                    try {
                        // 每隔 1 秒查看 ByteArrayOutputStream 检查新数据
                        Thread.sleep(1000);
                    }
                    catch(InterruptedException e) {}
                }
            }
        }).start();
    } // startByteArrayReaderThread()
} // LoopedStreams
```

startByteArrayReaderThread()方法是整个类真正的关键所在。这个方法的目标很简单，就是创建一个定期地检查 ByteArrayOutputStream 缓冲区的线程。缓冲区中找到的所有数据都被提取到一个 byte 数组，然后写入到 PipedOutputStream。由于 PipedOutputStream 对应的 PipedInputStream 由 getInputStream ()返回，从该输入流读取数据的线程都将读取到原先发送给 ByteArrayOutputStream 的数据。前面提到，LoopedStreams 类解决了管道流存在的前二个问题，我们来看看这是如何实现的。

ByteArrayOutputStream 具有根据需要扩展其内部缓冲区的能力。由于存在“完全缓冲”，线程向 getOutputStream()返回的流写入数据时不会被阻塞。因而，第一个问题不会再给我们带来麻烦。另外还要顺便说一句，ByteArrayOutputStream 的缓冲区永远不会缩减。例如，假设在能够提取数据之前，有一块 500 K 的数据被写入到流，缓冲区将永远保持至少 500 K 的容量。如果这个类有一个方法能够在数据被提取之后修正缓冲区的大小，它就会更完善。

第二个问题得以解决的原因在于，实际上任何时候只有一个线程向 PipedOutputStream 写入数据，这个线程就是由 startByteArrayReaderThread()创建的线程。由于这个线程完全由 LoopedStreams 类控制，我们不必担心它会产生 IOException 异常。

LoopedStreams 类还有一些细节值得提及。首先，我们可以看到 byteArrayOS 和 pipedIS 实际上分别是 ByteArrayOutputStream 和 PipedInputStream 的派生类的实例，也即在它们的 close()方法中加入了特殊的行为。如果一个 LoopedStreams 对象的用户关闭了输入或输出流，在 startByteArrayReaderThread()中创建的线程必须关闭。覆盖后的 close()方法把 keepRunning 标记设置成 false 以关闭线程。另外，请注意 startByteArrayReaderThread ()中的同步块。要确保在 toByteArray()调用和 reset()调用之间 ByteArrayOutputStream 缓冲区不被写入流的线程修改，这是必不可少的。由于 ByteArrayOutputStream 的 write()方法的所有版本都在该流上同步，我们保证了 ByteArrayOutputStream 的内部缓冲区不被意外地修改。

注意 LoopedStreams 类并不涉及管道流的第三个问题。该类的 getInputStream()方法返回 PipedInputStream。如果一个线程从该流读取，一段时间后终止，下次数据从 ByteArrayOutputStream 缓冲区传输到 PipedOutputStream 时就会出现 IOException 异常。

二、捕获 Java 控制台输出

Listing 5 的 ConsoleTextArea 类扩展 Swing JTextArea 捕获控制台输出。不要对这个类有这么多代码感到惊讶，必须指出的是，ConsoleTextArea 类有超过 50%的代码用来进行测试。

【Listing 5：截获 Java 控制台输出】

```
import java.io.*;
import java.util.*;
import javax.swing.*;
import javax.swing.text.*;
public class ConsoleTextArea extends JTextArea {
    public ConsoleTextArea(InputStream[] inStreams) {
        for(int i = 0; i < inStreams.length; ++i)
            startConsoleReaderThread(inStreams[i]);
    } // ConsoleTextArea()
    public ConsoleTextArea() throws IOException {
        final LoopedStreams ls = new LoopedStreams();
        // 重定向 System.out 和 System.err
        PrintStream ps = new PrintStream(ls.getOutputStream());
        System.setOut(ps);
        System.setErr(ps);
        startConsoleReaderThread(ls.getInputStream());
    } // ConsoleTextArea()
    private void startConsoleReaderThread(
        InputStream inStream) {
        final BufferedReader br =
            new BufferedReader(new InputStreamReader(inStream));
        new Thread(new Runnable() {
            public void run() {
                StringBuffer sb = new StringBuffer();
                try {
                    String s;
                    Document doc = getDocument();
                    while((s = br.readLine()) != null) {
                        boolean caretAtEnd = false;
                        caretAtEnd = getCaretPosition() == doc.getLength() ?
                            true : false;
                        sb.setLength(0);
                        append(sb.append(s).append('\n').toString());
                        if(caretAtEnd)
                            setCaretPosition(doc.getLength());
                    }
                }
                catch(IOException e) {
                    JOptionPane.showMessageDialog(null,
                        "从 BufferedReader 读取错误：" + e);
                    System.exit(1);
```

```
                }
            }
        }).start();
    } // startConsoleReaderThread()
    // 该类剩余部分的功能是进行测试
    public static void main(String[] args) {
        JFrame f = new JFrame("ConsoleTextArea 测试");
        ConsoleTextArea consoleTextArea = null;
        try {
            consoleTextArea = new ConsoleTextArea();
        }
        catch(IOException e) {
            System.err.println(
                "不能创建 LoopedStreams: " + e);
            System.exit(1);
        }
        consoleTextArea.setFont(java.awt.Font.decode("monospaced"));
        f.getContentPane().add(new JScrollPane(consoleTextArea),
            java.awt.BorderLayout.CENTER);
        f.setBounds(50, 50, 300, 300);
        f.setVisible(true);
        f.addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(
                java.awt.event.WindowEvent evt) {
                System.exit(0);
            }
        });
        // 启动几个写操作线程向
        // System.out 和 System.err 输出
        startWriterTestThread(
            "写操作线程 #1", System.err, 920, 50);
        startWriterTestThread(
            "写操作线程 #2", System.out, 500, 50);
        startWriterTestThread(
            "写操作线程 #3", System.out, 200, 50);
        startWriterTestThread(
            "写操作线程 #4", System.out, 1000, 50);
        startWriterTestThread(
            "写操作线程 #5", System.err, 850,    50);
    } // main()
    private static void startWriterTestThread(
```

```
        final String name, final PrintStream ps,
        final int delay, final int count) {
        new Thread(new Runnable() {
            public void run() {
                for(int i = 1; i <= count; ++i) {
                    ps.println("***" + name + ", hello !, i=" + i);
                    try {
                        Thread.sleep(delay);
                    }
                    catch(InterruptedException e) {}
                }
            }
        }).start();
    } // startWriterTestThread()
} // ConsoleTextArea
```

main()方法创建了一个 JFrame，JFrame 包含一个 ConsoleTextArea 的实例。这些代码并没有什么特别之处。Frame 显示出来之后，main()方法启动一系列的写操作线程，写操作线程向控制台流输出大量信息。ConsoleTextArea 捕获并显示这些信息，如图一所示。

ConsoleTextArea 提供了两个构造函数。没有参数的构造函数用来捕获和显示所有写入到控制台流的数据，有一个 InputStream[]参数的构造函数转发所有从各个数组元素读取的数据到 JTextArea。稍后将有一个例子显示这个构造函数的用处。首先我们来看看没有参数的 ConsoleTextArea 构造函数。这个函数首先创建一个 LoopedStreams 对象；然后请求 Java 运行时环境把控制台输出转发到 LoopedStreams 提供的 OutputStream；最后，构造函数调用 startConsoleReaderThread()，创建一个不断地把文本行追加到 JTextArea 的线程。注意，把文本追加到 JTextArea 之后，程序小心地保证了插入点的正确位置。

一般来说，Swing 部件的更新不应该在 AWT 事件分派线程(AWT Event Dispatch Thread，AEDT ）之外进行。对于本例来说，这意味着所有把文本追加到 JTextArea 的操作应该在

AEDT 中进行，而不是在 startConsoleReaderThread()方法创建的线程中进行。然而，事实上在 Swing 中向 JTextArea 追加文本是一个线程安全的操作。读取一行文本之后，我们只需调用 JText.append()就可以把文本追加到 JTextArea 的末尾。

三、捕获其他程序的控制台输出

在 JTextArea 中捕获 Java 程序自己的控制台输出是一回事，去捕获其他程序（甚至包括一些非 Java 程序）的控制台数据又是另一回事。ConsoleTextArea 提供了捕获其他应用的输出时需要的基础功能，Listing 6 的 AppOutputCapture 利用 ConsoleTextArea，截取其他应用的输出信息然后显示在 ConsoleTextArea 中。

【Listing 6：截获其他程序的控制台输出】

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;
public class AppOutputCapture {
        private static Process process;
        public static void main(String[] args) {
                if(args.length == 0) {
            System.err.println("用法：java AppOutputCapture " +
                "<程序名字>｛参数 1 参数 2 ...｝");
            System.exit(0);
                }
                try {
                        // 启动命令行指定程序的新进程
                        process = Runtime.getRuntime().exec(args);
                }
                catch(IOException e) {
                        System.err.println("创建进程时出错...\n" + e);
                        System.exit(1);
                }
                // 获得新进程所写入的流
```

```
                InputStream[] inStreams =
                        new InputStream[] {
                    process.getInputStream(),process.getErrorStream()};
                ConsoleTextArea cta = new
    ConsoleTextArea(inStreams);
                cta.setFont(java.awt.Font.decode("monospaced"));
                JFrame frame = new JFrame(args[0] +
                        "控制台输出");
                frame.getContentPane().add(new JScrollPane(cta),
                    BorderLayout.CENTER);
                frame.setBounds(50, 50, 400, 400);
                frame.setVisible(true);
                frame.addWindowListener(new WindowAdapter() {
                        public void windowClosing(WindowEvent evt) {
                                process.destroy();
                                try {
                                        process.waitFor(); // 在 Win98 下可能被挂起
                                }
                                catch(InterruptedException e) {}
                                        System.exit(0);
                                }
                        });
        } // main()
} // AppOutputCapture
```

AppOutputCapture 的工作过程如下：首先利用 Runtime.exec()方法启动指定程序的一个新进程。启动新进程之后，从结果 Process 对象得到它的控制台流。之后，把这些控制台流传入 ConsoleTextArea(InputStream[])构造函数（这就是带参数 ConsoleTextArea 构造函数的用处）。使用 AppOutputCapture 时，在命令行上指定待截取其输出的程序名字。例如，如果在 Windows 2000 下执行 javaw.exe AppOutputCapture ping.exe www.yahoo.com，则结果如图四所示。

使用 AppOutputCapture 时应该注意，被截取输出的应用程序最初输出的一些文本可能无法截取。因为在调用 Runtime.exec()和 ConsoleTextArea 初始化完成之间存在一小段时间差。

在这个时间差内，应用程序输出的文本会丢失。当 AppOutputCapture 窗口被关闭，process.destory()调用试图关闭 Java 程序开始时创建的进程。测试结果显示出，destroy()方法不一定总是有效（至少在 Windows 98 上是这样的）。似乎当待关闭的进程启动了额外的进程时，则那些进程不会被关闭。此外，在这种情况下 AppOutputCapture 程序看起来未能正常结束。但在Windows NT下，一切正常。如果用 JDK v1.1.x运行 AppOutputCapture，关闭窗口时会出现一个NullPointerException。这是一个 JDK 的Bug，JDK 1.2.x和 JDK 1.3.x下就不会出现问题。