Java IO - BIO 详解

BIO就是: blocking IO。最容易理解、最容易实现的IO工作方式，应用程序向操作系统请求网络IO操作，这时应用程序会一直等待；另一方面，操作系统收到请求后，也会等待，直到网络上有数据传到监听端口；操作系统在收集数据后，会把数据发送给应用程序；最后应用程序受到数据，并解除等待状态。

几个重要概念

- 阻塞IO 和 非阻塞IO

这两个概念是程序级别的。主要描述的是程序请求操作系统IO操作后，如果IO资源没有准备好，那么程序该如何处理的问题: 前者等待；后者继续执行(并且使用线程一直轮询，直到有IO资源准备好了)

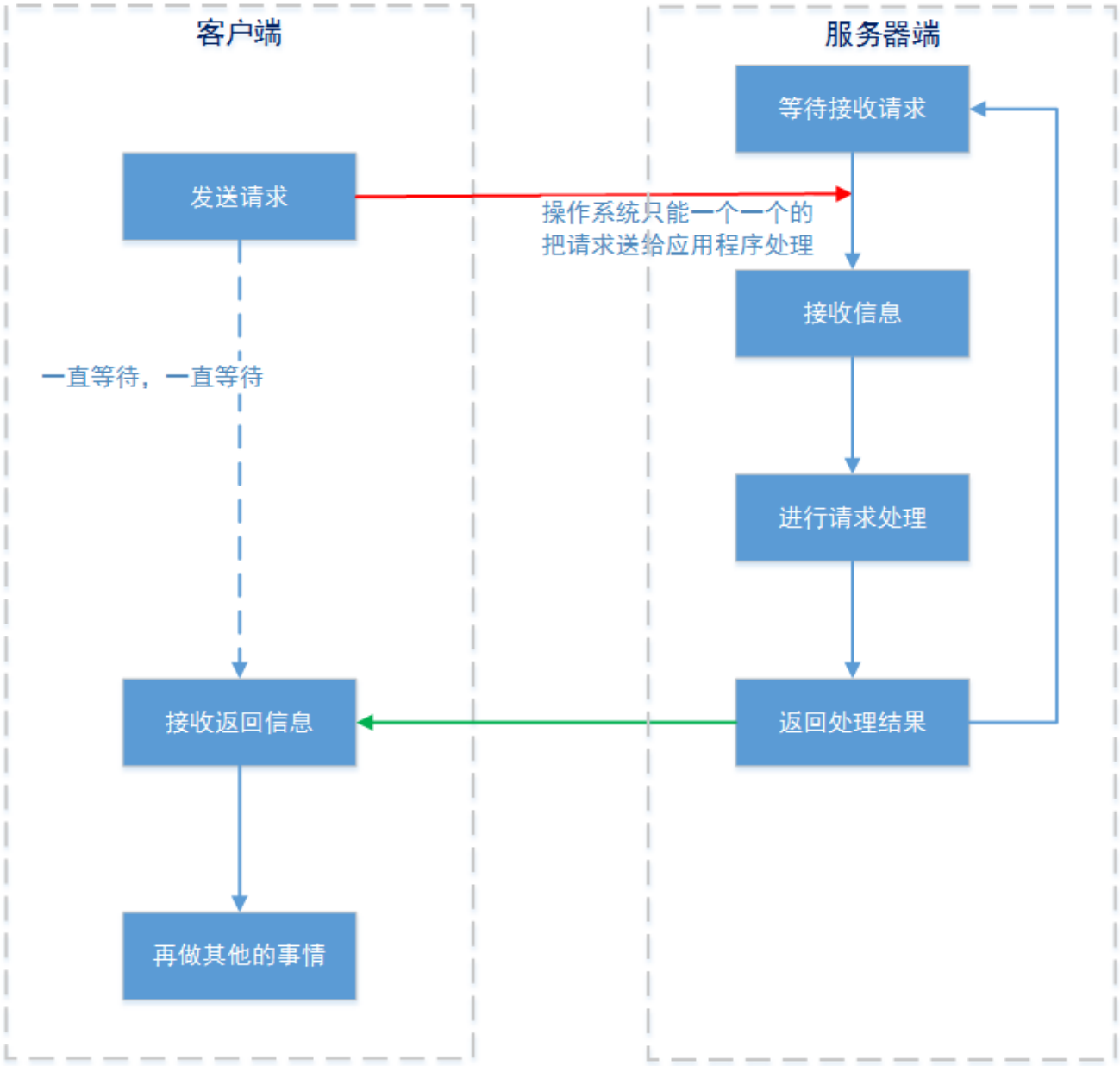
- 同步IO 和 非同步IO

这两个概念是操作系统级别的。主要描述的是操作系统在收到程序请求IO操作后，如果IO资源没有准备好，该如何响应程序的问题: 前者不响应，直到IO资源准备好以后；后者返回一个标记(好让程序和自己知道以后的数据往哪里通知)，当IO资源准备好以后，再用事件机制返回给程序。

传统的BIO通信方式简介

以前大多数网络通信方式都是阻塞模式的，即:

- 客户端向服务器端发出请求后，客户端会一直等待(不会再做其他事情)，直到服务器端返回结果或者网络出现问题。
- 服务器端同样的，当在处理某个客户端A发来的请求时，另一个客户端B发来的请求会等待，直到服务器端的这个处理线程完成上一个处理。



传统的*BIO*的问题

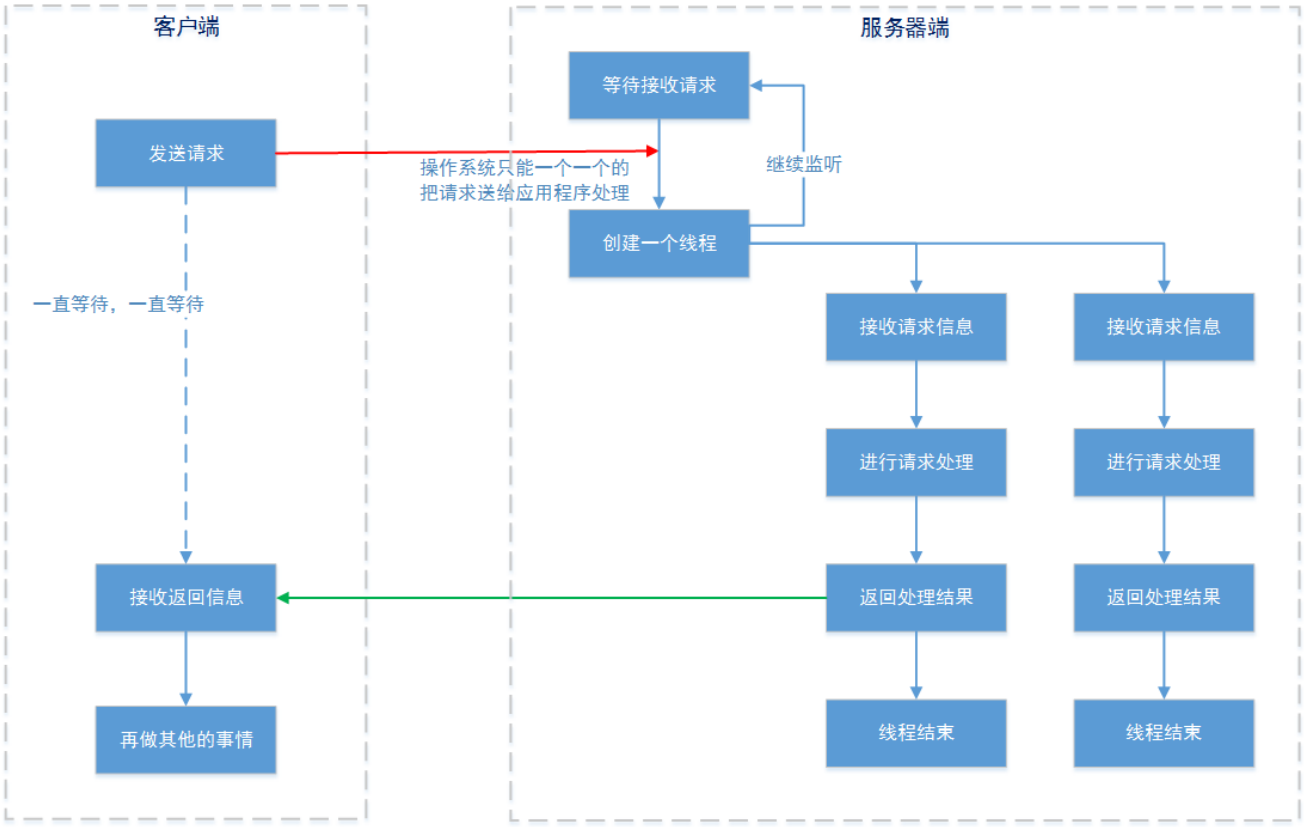
- 同一时间，服务器只能接受来自于客户端A的请求信息；虽然客户端A和客户端B的请求是同时进行的，但客户端B发送的请求信息只能等到服务器接受完A的请求数据后，才能被接受。
- 由于服务器一次只能处理一个客户端请求，当处理完成并返回后(或者异常时)，才能进行第二次请求的处理。很显然，这样的处理方式在高并发的情况下，是不能采用的。

多线程方式 - 伪异步方式

上面说的情况是服务器只有一个线程的情况，那么读者会直接提出我们可以使用多线程技术来解决这个问题:

- 当服务器收到客户端X的请求后，(读取到所有请求数据后)将这个请求送入一个独立线程进行处理，然后主线程继续接受客户端Y的请求。
- 客户端一侧，也可以使用一个子线程和服务器端进行通信。这样客户端主线程的其他工作就不受影响了，当服务器端有响应信息的时候再由这个子线程通过 监听模式/观察模式(等其他设计模式)通知主线程。

如下图所示:



但是使用线程来解决这个问题实际上是有局限性的:

- 虽然在服务器端，请求的处理交给了一个独立线程进行，但是操作系统通知accept()的方式还是单个的。也就是，实际上是服务器接收到数据报文后的“业务处理过程”可以多线程，但是数据报文的接受还是需要一个一个的来(下文的示例代码和debug过程我们可以明确看到这一点)
- 在linux系统中，可以创建的线程是有限的。我们可以通过cat /proc/sys/kernel/threads-max 命令查看可以创建的最大线程数。当然这个值是可以更改的，但是线程越多，CPU切换所需的时间也就越长，用来处理真正业务的需求也就越少。
- 创建一个线程是有较大的资源消耗的。JVM创建一个线程的时候，即使这个线程不做任何的工作，JVM都会分配一个堆栈空间。这个空间的大小默认为128K，您可以通过-Xss参数进行调整。当然您还可以使用ThreadPoolExecutor线程池来缓解线程的创建问题，但是又会造成BlockingQueue积压任务的持续增加，同样消耗了大量资源。
- 另外，如果您的应用程序大量使用长连接的话，线程是不会关闭的。这样系统资源的消耗更容易失控。 那么，如果你真想单纯使用线程解决阻塞的问题，那么您自己都可以算出来您一个服务器节点可以一次接受多大的并发了。看来，单纯使用线程解决这个问题不是最好的办法。

BIO通信方式深入分析

BIO的问题关键不在于是否使用了多线程(包括线程池)处理这次请求，而在于accept()、read()的操作点都是被阻塞。要测试这个问题，也很简单。我们模拟了20个客户端(用20根线程模拟)，利用JAVA的同步计数器CountDownLatch，保证这20个客户都初始化完成后然后同时向服务器发送请求，然后我们来观察一下Server这边接受信息的情况。

模拟20个客户端并发请求，服务器端使用单线程:

客户端代码(SocketClientDaemon)

```
package testBSocket;

import java.util.concurrent.CountDownLatch;

public class SocketClientDaemon {
    public static void main(String[] args) throws Exception {
        Integer clientNumber = 20;
        CountDownLatch countDownLatch = new CountDownLatch(clientNumber);

        //分别开始启动这20个客户端
        for(int index = 0 ; index < clientNumber ; index++ , countDownLatch.countDown()) {
            SocketClientRequestThread client = new SocketClientRequestThread(countDownLatch,
index);
            new Thread(client).start();
        }

        //这个wait不涉及到具体的实验逻辑，只是为了保证守护线程在启动所有线程后，进入等待状态
        synchronized (SocketClientDaemon.class) {
            SocketClientDaemon.class.wait();
        }
    }
}
```

客户端代码(SocketClientRequestThread模拟请求)

```
package testBSocket;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import java.util.concurrent.CountDownLatch;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.log4j.BasicConfigurator;

/**
 * 一个SocketClientRequestThread线程模拟一个客户端请求。
 * @author yinwenjie
 */
public class SocketClientRequestThread implements Runnable {

    static {
        BasicConfigurator.configure();
    }

    /**
     * 日志
     */
    private static final Log LOGGER = LogFactory.getLog(SocketClientRequestThread.class);

    private CountDownLatch countDownLatch;

    /**
```

```
     * 这个线层的编号
     * @param countDownLatch
     */
    private Integer clientIndex;

    /**
     * countDownLatch是java提供的同步计数器。
     * 当计数器数值减为0时，所有受其影响而等待的线程将会被激活。这样保证模拟并发请求的真实性
     * @param countDownLatch
     */
    public SocketClientRequestThread(CountDownLatch countDownLatch , Integer clientIndex) {
        this.countDownLatch = countDownLatch;
        this.clientIndex = clientIndex;
    }

    @Override
    public void run() {
        Socket socket = null;
        OutputStream clientRequest = null;
        InputStream clientResponse = null;

        try {
            socket = new Socket("localhost",83);
            clientRequest = socket.getOutputStream();
            clientResponse = socket.getInputStream();

            //等待，直到SocketClientDaemon完成所有线程的启动，然后所有线程一起发送请求
            this.countDownLatch.await();

            //发送请求信息
            clientRequest.write(("这是第" + this.clientIndex + " 个客户端的请求。").getBytes());
            clientRequest.flush();

            //在这里等待，直到服务器返回信息
            SocketClientRequestThread.LOGGER.info("第" + this.clientIndex + "个客户端的请求发送完
成，等待服务器返回信息");
            int maxLen = 1024;
            byte[] contextBytes = new byte[maxLen];
            int realLen;
            String message = "";
            //程序执行到这里，会一直等待服务器返回信息(注意，前提是in和out都不能close，如果close了就收
不到服务器的反馈了)
            while((realLen = clientResponse.read(contextBytes, 0, maxLen)) != -1) {
                message += new String(contextBytes , 0 , realLen);
            }
            SocketClientRequestThread.LOGGER.info("接收到来自服务器的信息:" + message);
        } catch (Exception e) {
            SocketClientRequestThread.LOGGER.error(e.getMessage(), e);
        } finally {
            try {
                if(clientRequest != null) {
                    clientRequest.close();
                }
                if(clientResponse != null) {
                    clientResponse.close();
                }
            } catch (IOException e) {
                SocketClientRequestThread.LOGGER.error(e.getMessage(), e);
            }
        }
    }
```

```
}
```

服务器端(SocketServer1)单个线程

```
package testBSocket;

import java.io.InputStream;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.log4j.BasicConfigurator;

public class SocketServer1 {

    static {
        BasicConfigurator.configure();
    }

    /**
     * 日志
     */
    private static final Log LOGGER = LogFactory.getLog(SocketServer1.class);

    public static void main(String[] args) throws Exception{
        ServerSocket serverSocket = new ServerSocket(83);

        try {
            while(true) {
                Socket socket = serverSocket.accept();

                //下面我们收取信息
                InputStream in = socket.getInputStream();
                OutputStream out = socket.getOutputStream();
                Integer sourcePort = socket.getPort();
                int maxLen = 2048;
                byte[] contextBytes = new byte[maxLen];
                //这里也会被阻塞，直到有数据准备好
                int realLen = in.read(contextBytes, 0, maxLen);
                //读取信息
                String message = new String(contextBytes , 0 , realLen);

                //下面打印信息
                SocketServer1.LOGGER.info("服务器收到来自于端口: " + sourcePort + "的信息: " +
message);

                //下面开始发送信息
                out.write("回发响应信息！".getBytes());

                //关闭
                out.close();
                in.close();
                socket.close();
            }
        } catch(Exception e) {
            SocketServer1.LOGGER.error(e.getMessage(), e);
        } finally {
            if(serverSocket != null) {
```

```
                serverSocket.close();
            }
        }
    }
}
```

多线程来优化服务器端

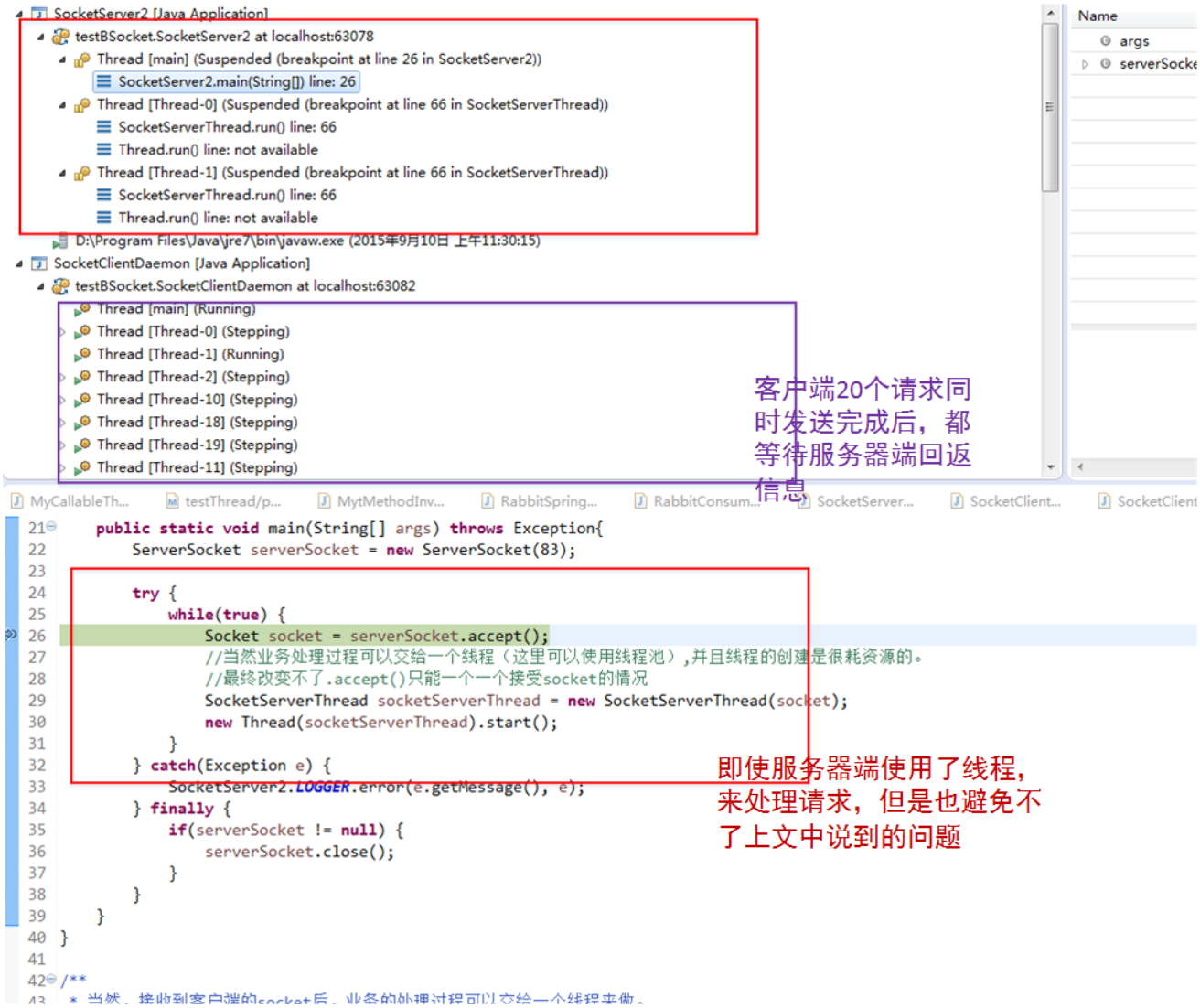
客户端代码和上文一样，最主要是更改服务器端的代码:

```
package testBSocket;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.log4j.BasicConfigurator;

public class SocketServer2 {

    static {
        BasicConfigurator.configure();
    }

    private static final Log LOGGER = LogFactory.getLog(SocketServer2.class);

    public static void main(String[] args) throws Exception{
        ServerSocket serverSocket = new ServerSocket(83);

        try {
            while(true) {
                Socket socket = serverSocket.accept();
                //当然业务处理过程可以交给一个线程(这里可以使用线程池),并且线程的创建是很耗资源的。
                //最终改变不了.accept()只能一个一个接受socket的情况,并且被阻塞的情况
                SocketServerThread socketServerThread = new SocketServerThread(socket);
                new Thread(socketServerThread).start();
            }
        } catch(Exception e) {
            SocketServer2.LOGGER.error(e.getMessage(), e);
        } finally {
            if(serverSocket != null) {
                serverSocket.close();
            }
        }
    }
}

/**
 * 当然，接收到客户端的socket后，业务的处理过程可以交给一个线程来做。
 * 但还是改变不了socket被一个一个的做accept()的情况。
 * @author yinwenjie
 */
class SocketServerThread implements Runnable {
```

```
    /**
     * 日志
     */
    private static final Log LOGGER = LogFactory.getLog(SocketServerThread.class);

    private Socket socket;

    public SocketServerThread (Socket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {
        InputStream in = null;
        OutputStream out = null;
        try {
            //下面我们收取信息
            in = socket.getInputStream();
            out = socket.getOutputStream();
            Integer sourcePort = socket.getPort();
            int maxLen = 1024;
            byte[] contextBytes = new byte[maxLen];
            //使用线程，同样无法解决read方法的阻塞问题，
            //也就是说read方法处同样会被阻塞，直到操作系统有数据准备好
            int realLen = in.read(contextBytes, 0, maxLen);
            //读取信息
            String message = new String(contextBytes , 0 , realLen);

            //下面打印信息
            SocketServerThread.LOGGER.info("服务器收到来自于端口: " + sourcePort + "的信息: " +
message);

            //下面开始发送信息
            out.write("回发响应信息！".getBytes());
        } catch(Exception e) {
            SocketServerThread.LOGGER.error(e.getMessage(), e);
        } finally {
            //试图关闭
            try {
                if(in != null) {
                    in.close();
                }
                if(out != null) {
                    out.close();
                }
                if(this.socket != null) {
                    this.socket.close();
                }
            } catch (IOException e) {
                SocketServerThread.LOGGER.error(e.getMessage(), e);
            }
        }
    }
}
```

看看服务器端的执行效果

看一看服务器使用多线程处理时的情况:



问题根源

那么重点的问题并不是“是否使用了多线程”，而是为什么accept()、read()方法会被阻塞。即: 异步IO模式 就是为了解决这样的并发性存在的。但是为了说清楚异步IO模式，在介绍IO模式的时候，要首先了解清楚，什么是 阻塞式同步、非阻塞式同步、多路复用同步模式。

API文档中对于 serverSocket.accept() 方法的使用描述:

> Listens for a connection to be made to this socket and accepts it. The method blocks until a connection is made.

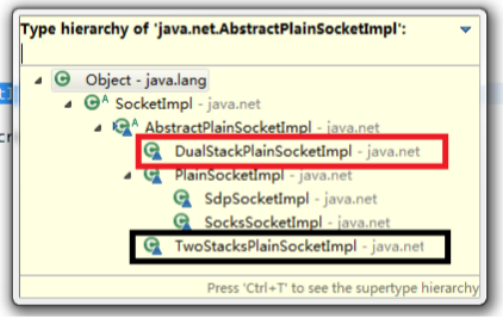
serverSocket.accept()会被阻塞? 这里涉及到阻塞式同步IO的工作原理:

- 服务器线程发起一个accept动作，询问操作系统 是否有新的socket套接字信息从端口X发送过来。

Vsita内核和之后的windows版本使用DualStackPlainSocketImpl这个类

这是大师

```
/**
 * This class defines the plain SocketImpl that is used on Windows platforms
 * greater or equal to Windows Vista. These platforms have a dual
 * layer TCP/IP stack and can handle both IPv4 and IPV6 through a
 * single file descriptor.
 *
 * @author Chris Hegarty
 */

class DualStackPlainSocketImpl extends AbstractPlainSocket
{
    static JavaIOFileDescriptorAccess fdAccess = SharedSecr


    // true if this socket is exclusively bound
    private final boolean exclusiveBind;

    // emulates SO_REUSEADDR when exclusiveBind is true
    private boolean isReuseAddress;
```



我使用的是windows下的JDK，Vsita内核之前的socket实现是使用的TwoStacksPlainSocketImpl这个类

- 注意，是询问操作系统。也就是说socket套接字的IO模式支持是基于操作系统的，那么自然同步IO/异步IO的支持就是需要操作系统级别的了。如下图:

如果java程序没有设置timeout，那么java程序在调用JNI时，会一直等待，知道有数据返回

```
130         if (timeout <= 0) {
131             newfd = accept0(nativefd, isaa);
132         } else {
133             configureBlocking(nativefd, false);
134             try {
135                 waitForNewConnection(nativefd, timeout);
136                 newfd = accept0(nativefd, isaa);
137                 if (newfd != -1) {
138                     configureBlocking(newfd, true);
139                 }
140             } finally {
141                 configureBlocking(nativefd, true);
142             }
143         }
```

```
static native void localAddress(int fd, InetAddressContainer in) throws SocketException;

static native void listen0(int fd, int backlog) throws IOException;

static native int accept0(int fd, InetSocketAddress[] isaa) throws IOException;

static native void waitForNewConnection(int fd, int timeout) throws IOException;

static native int available0(int fd) throws IOException;

static native void close0(int fd) throws IOException;

static native void shutdown0(int fd, int howto) throws IOException;

static native void setIntOption(int fd, int cmd, int optionValue) throws SocketException;

static native int getIntOption(int fd, int cmd) throws SocketException;

static native void sendOOB(int fd, int data) throws IOException;

static native void configureBlocking(int fd, boolean blocking) throws IOException;
```

如果操作系统没有发现有套接字从指定的端口X来，那么操作系统就会等待。这样serverSocket.accept()方法就会一直等待。这就是为什么accept()方法为什么会阻塞: 它内部的实现是使用的操作系统级别的同步IO。