@[toc]

一、介绍

1、传统IO特点

(1)代码执行时会存在两个阻塞点：

```
server.accept();      等待链接
inputStream.read(bytes);      等待输入
```

(2)单线程情况下只能为一个客户端服务；

(3)用线程池可以有多个客户端连接，但是非常消耗性能；

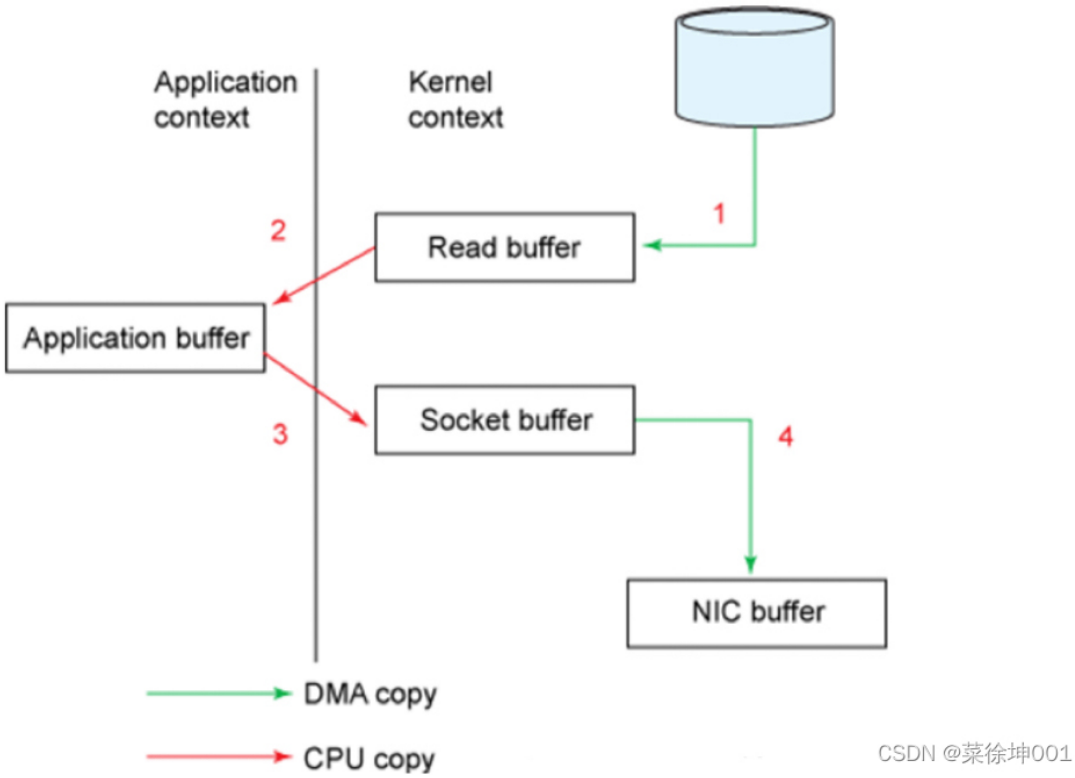
(4)使用传统的I/O程序读取文件内容, 并写入到另一个文件(或Socket), 如下程序:

```
File.read(fileDesc, buf, len);
Socket.send(socket, buf, len);
```

会有较大的性能开销, 主要表现在一下两方面:

```
1. 上下文切换(context switch)，此处有4次用户态和内核态的切换
2. Buffer内存开销，一个是应用程序buffer，另一个是系统读取buffer以及socket buffer
```

其运行示意图如下:



- 1. 先将文件内容从磁盘中拷贝到操作系统buffer

- 2. 再从操作系统buffer拷贝到程序应用buffer
- 3. 从程序buffer拷贝到socket buffer

- 4. 从socket buffer拷贝到协议引擎. 传统IO代码实现:

```
public class OioServer {
@SuppressWarnings("resource")
public static void main(String[] args) throws Exception {

    ExecutorService newCachedThreadPool = Executors.newCachedThreadPool();
    //创建socket服务,监听10101端口
    ServerSocket server=new ServerSocket(10101);
    System.out.println("服务器启动! ");
    while(true){
        //获取一个套接字（阻塞）
        final Socket socket = server.accept();
        System.out.println("来个一个新客户端! ");
        newCachedThreadPool.execute(new Runnable() {

            @Override
            public void run() {
                //业务处理
                handler(socket);
            }
        });

    }
}

/**
 * 读取数据
 * @param socket
 * @throws Exception
 */
public static void handler(Socket socket){
        try {
            byte[] bytes = new byte[1024];
            InputStream inputStream = socket.getInputStream();

            while(true){
                //读取数据（阻塞）
                int read = inputStream.read(bytes);
                if(read != -1){
                    System.out.println(new String(bytes, 0, read));
                }else{
                    break;
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }finally{
            try {
                System.out.println("socket关闭");
                socket.close();
            } catch (IOException e) {
                e.printStackTrace();
```

```
            }
        }
}}
```

2、NIO的特点

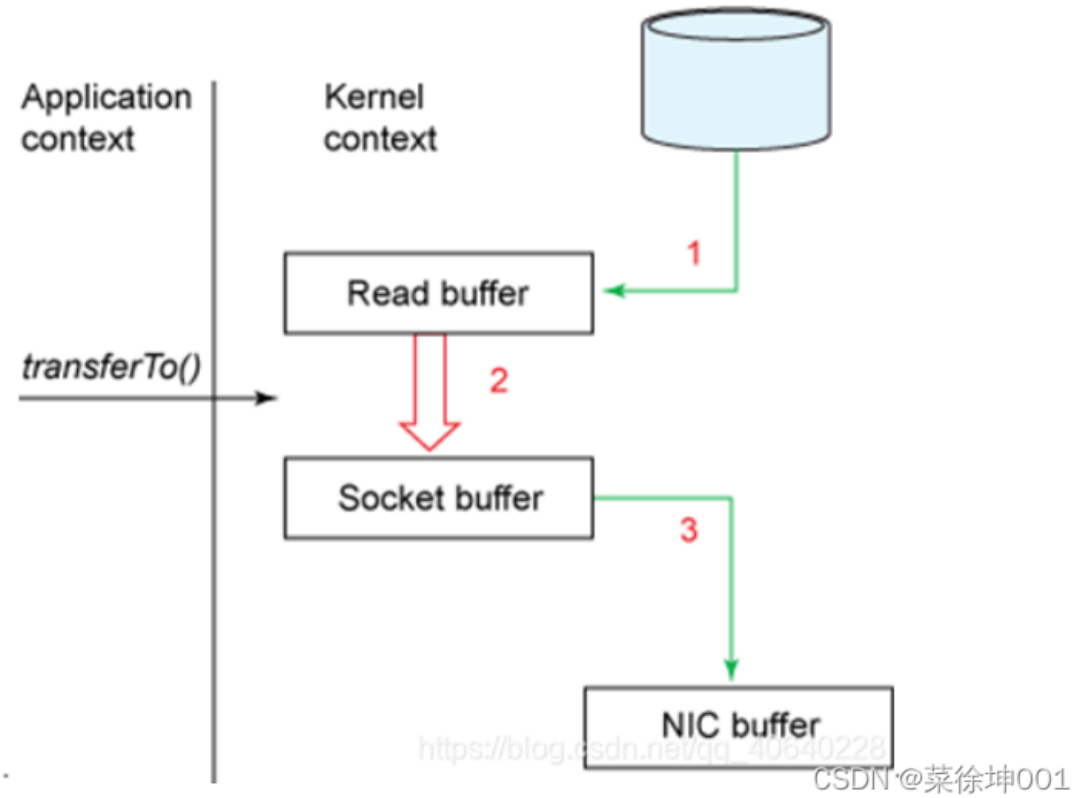
- 1、NIO在单线程下可以同时为多个客户端服务
- 2、NIO技术省去了将操作系统的read buffer拷贝到程序的buffer, 以及从程序buffer拷贝到socket buffer的步骤, 直接将 read buffer 拷贝到 socket buffer. java 的 FileChannel.transferTo() 方法就是这样的实现, 这个实现是依赖于操作系统底层的sendFile()实现的.

```
    public void transferTo(long position, long count, WritableByteChannel
target);
```

他的底层调用的是系统调用sendFile()方法:

```
    sendfile(int out_fd, int in_fd, off_t *offset, size_t count);
```

如下图:



- 3、使用NIO遇到的一些问题

```
(1)客户端关闭的时候会抛出异常，死循环

    解决方案:
    int read = channel.read(buffer);
    if(read > 0){
        byte[] data = buffer.array();
        String msg = new String(data).trim();
        System.out.println("服务端收到信息：" + msg);
```

```
        //回写数据
        ByteBuffer outBuffer = ByteBuffer.wrap("好的".getBytes());
        channel.write(outBuffer);// 将消息回送给客户端
    }else{
        System.out.println("客户端关闭");
        key.cancel();
    }

(2)selector.select();阻塞，那为什么说nio是非阻塞的IO?
   selector.select()
   selector.select(1000);不阻塞
   selector.wakeup();也可以唤醒selector
   selector.selectNow();也可以立马返回数据

(3)SelectionKey.OP_WRITE是代表什么意思
   OP_WRITE表示底层缓冲区是否有空间，是则响应返还true
```

Nio代码实现：

```
public class NIOServer {
// 通道管理器
private Selector selector;
/**
 * 获得一个ServerSocket通道，并对该通道做一些初始化的工作
 *
 * @param port
 *            绑定的端口号
 * @throws IOException
 */
public void initServer(int port) throws IOException {
    // 获得一个ServerSocket通道
    ServerSocketChannel serverChannel = ServerSocketChannel.open();
    // 设置通道为非阻塞
    serverChannel.configureBlocking(false);
    // 将该通道对应的ServerSocket绑定到port端口
    serverChannel.socket().bind(new InetSocketAddress(port));
    // 获得一个通道管理器
    this.selector = Selector.open();
    // 将通道管理器和该通道绑定，并为该通道注册SelectionKey.OP_ACCEPT事件,注册该事件后，
    // 当该事件到达时，selector.select()会返回，如果该事件没到达selector.select()会一直阻
塞。
    serverChannel.register(selector, SelectionKey.OP_ACCEPT);
}

/**
 * 采用轮询的方式监听selector上是否有需要处理的事件，如果有，则进行处理
 *
 * @throws IOException
 */
public void listen() throws IOException {
    System.out.println("服务端启动成功! ");
    // 轮询访问selector
    while (true) {
        // 当注册的事件到达时，方法返回；否则,该方法会一直阻塞
        selector.select();
```

```
            // 获得selector中选中的项的迭代器，选中的项为注册的事件
            Iterator<?> ite = this.selector.selectedKeys().iterator();
            while (ite.hasNext()) {
                SelectionKey key = (SelectionKey) ite.next();
                // 删除已选的key,以防重复处理
                ite.remove();

                handler(key);
            }
        }
    }

    /**
     * 处理请求
     *
     * @param key
     * @throws IOException
     */
    public void handler(SelectionKey key) throws IOException {

        // 客户端请求连接事件
        if (key.isAcceptable()) {
            handlerAccept(key);
            // 获得了可读的事件
        } else if (key.isReadable()) {
            handelerRead(key);
        }
    }

    /**
     * 处理连接请求
     *
     * @param key
     * @throws IOException
     */
    public void handlerAccept(SelectionKey key) throws IOException {
        ServerSocketChannel server = (ServerSocketChannel) key.channel();
        // 获得和客户端连接的通道
        SocketChannel channel = server.accept();
        // 设置成非阻塞
        channel.configureBlocking(false);

        // 在这里可以给客户端发送信息哦
        System.out.println("新的客户端连接");
        // 在和客户端连接成功之后，为了可以接收到客户端的信息，需要给通道设置读的权限。
        channel.register(this.selector, SelectionKey.OP_READ);
    }

    /**
     * 处理读的事件
     *
     * @param key
     * @throws IOException
     */
    public void handelerRead(SelectionKey key) throws IOException {
        // 服务器可读取消息:得到事件发生的Socket通道
        SocketChannel channel = (SocketChannel) key.channel();
        // 创建读取的缓冲区
```

```
    ByteBuffer buffer = ByteBuffer.allocate(1024);
    int read = channel.read(buffer);
    if(read > 0){
        byte[] data = buffer.array();
        String msg = new String(data).trim();
        System.out.println("服务端收到信息：" + msg);

        //回写数据
        ByteBuffer outBuffer = ByteBuffer.wrap("好的".getBytes());
        channel.write(outBuffer);// 将消息回送给客户端
    }else{
        System.out.println("客户端关闭");
        key.cancel();
    }
}

/**
 * 启动服务端测试
 *
 * @throws IOException
 */
public static void main(String[] args) throws IOException {
    NIOServer server = new NIOServer();
    server.initServer(8000);
    server.listen();
}}
```

3、Netty简介

- Netty是基于Java NIO的网络应用框架.
- Netty是一个NIO client-server(客户端服务器)框架，使用Netty可以快速开发网络应用，例如服务器和客户端协议。
- Netty提供了一种新的方式来使开发网络应用程序，这种新的方式使得它很容易使用和有很强的扩展性。
- Netty的内部实现是很复杂的，但是Netty提供了简单易用的api从网络处理代码中解耦业务逻辑。
- Netty是完全基于NIO实现的，所以整个Netty都是异步的。

2、netty可以运用在那些领域?

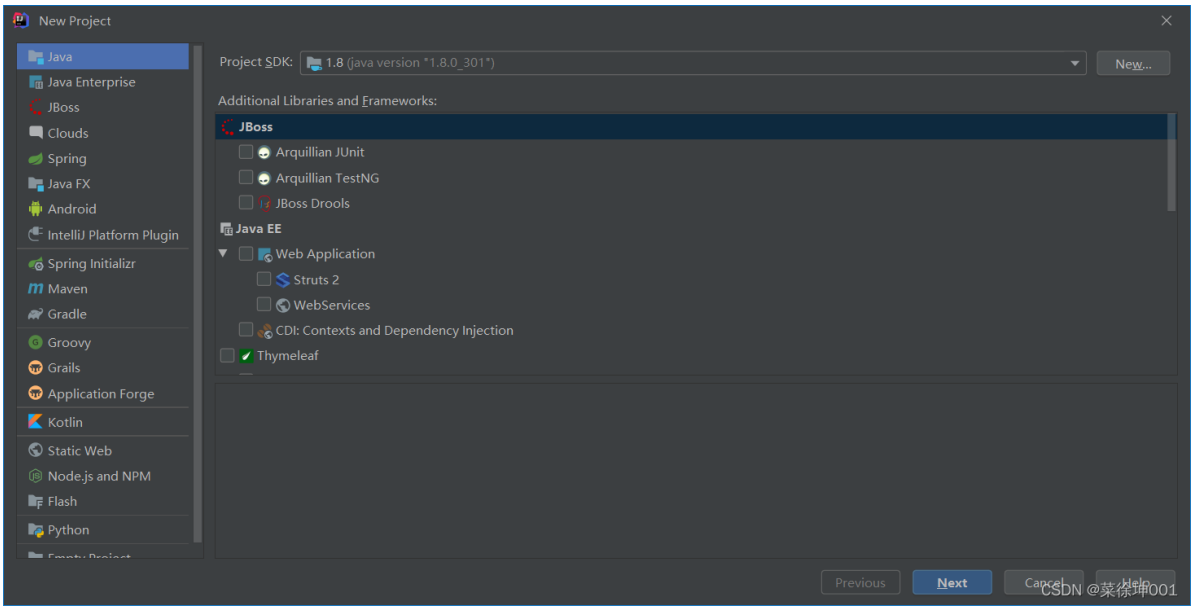
1. 分布式进程通信

   例如: hadoop、dubbo、akka等具有分布式功能的框架，底层RPC通信都是基于netty实现的。

2. 游戏服务器开发
3. netty服务端hello world案例

二、IO示例

打开IDEA，新建两个Java项目。



server端代码：

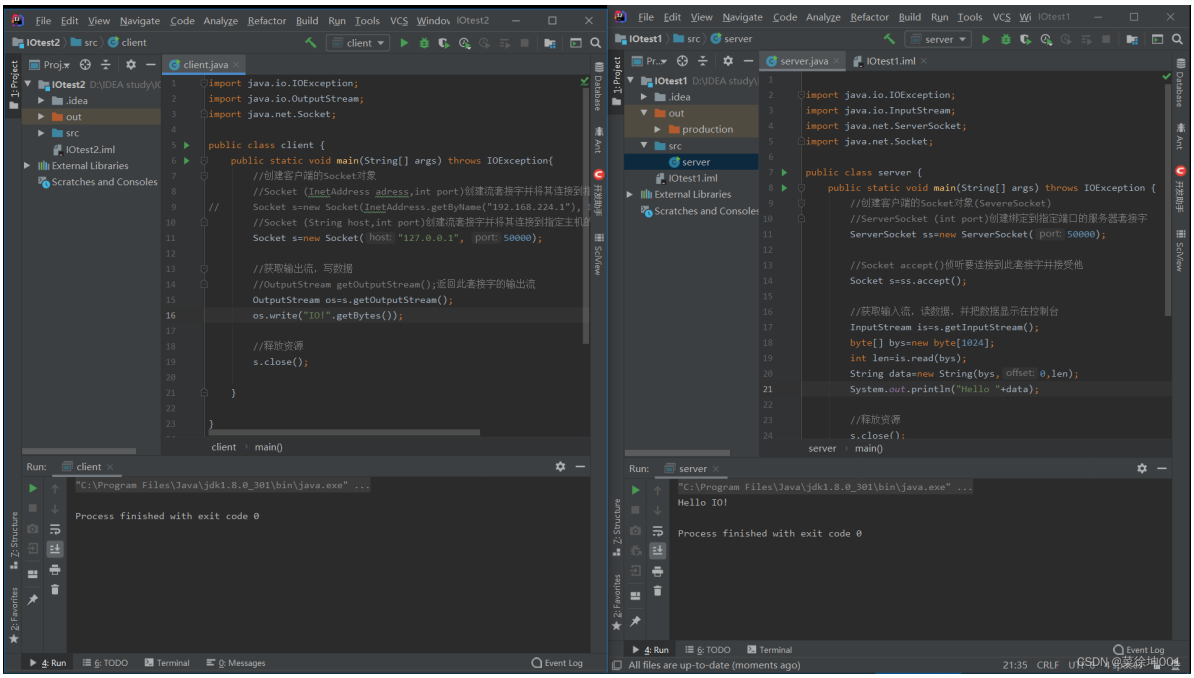
```
import java.io.IOException;
import java.io.InputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class server {
    public static void main(String[] args) throws IOException {
        //创建客户端的Socket对象(SevereSocket)
        //ServerSocket (int port)创建绑定到指定端口的服务器套接字
        ServerSocket ss=new ServerSocket(50000);

        //Socket accept()侦听要连接到此套接字并接受他
        Socket s=ss.accept();

        //获取输入流，读数据，并把数据显示在控制台
        InputStream is=s.getInputStream();
        byte[] bys=new byte[1024];
        int len=is.read(bys);
        String data=new String(bys,0,len);
        System.out.println("Hello "+data);

        //释放资源
        s.close();
        ss.close();
    }

}
```

client端代码：

```
import java.io.IOException;
import java.io.OutputStream;
import java.net.Socket;
```

```
public class client {
    public static void main(String[] args) throws IOException{
        //创建客户端的Socket对象
        //Socket (InetAddress adress,int port)创建流套接字并将其连接到指定IP地址的指定端口号
//      Socket s=new Socket(InetAddress.getByName("192.168.224.1"), 10000);
        //Socket (String host,int port)创建流套接字并将其连接到指定主机的指定端口号
        Socket s=new Socket("127.0.0.1", 50000);

        //获取输出流，写数据
        //OutputStream getOutputStream();返回此套接字的输出流
        OutputStream os=s.getOutputStream();
        os.write("IO!".getBytes());

        //释放资源
        s.close();

    }

}
```

运行结果：



三、NIO示例

还是上面那两个项目 **server端代码**：

```
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.Iterator;
```

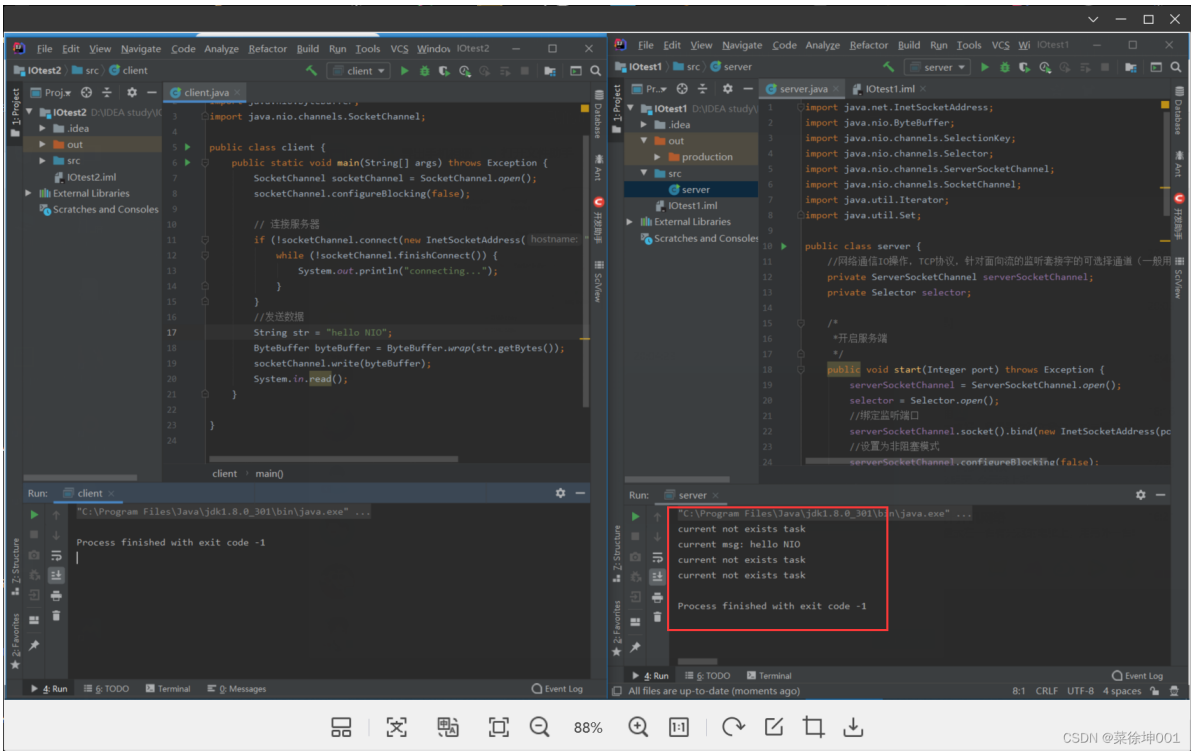
```
import java.util.Set;

public class server {
    //网络通信IO操作，TCP协议，针对面向流的监听套接字的可选择通道（一般用于服务端）
    private ServerSocketChannel serverSocketChannel;
    private Selector selector;

    /*
     *开启服务端
     */
    public void start(Integer port) throws Exception {
        serverSocketChannel = ServerSocketChannel.open();
        selector = Selector.open();
        //绑定监听端口
        serverSocketChannel.socket().bind(new InetSocketAddress(port));
        //设置为非阻塞模式
        serverSocketChannel.configureBlocking(false);
        //注册到Selector上
        serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
        startListener();
    }
        private void startListener() throws Exception {
            while (true) {
                // 如果客户端有请求select的方法返回值将不为零
                if (selector.select(1000) == 0) {
                    System.out.println("current not exists task");
                    continue;
                }
                // 如果有事件集合中就存在对应通道的key
                Set<SelectionKey> selectionKeys = selector.selectedKeys();
                Iterator<SelectionKey> iterator = selectionKeys.iterator();
                // 遍历所有的key找到其中事件类型为Accept的key
                while (iterator.hasNext()) {
                    SelectionKey key = iterator.next();
                    if (key.isAcceptable())
                        handleConnection();
                    if (key.isReadable())
                        handleMsg(key);
                    iterator.remove();
                }
            }
        }
    /**
     * 处理建立连接
     */
    private void handleConnection() throws Exception {
        SocketChannel socketChannel = serverSocketChannel.accept();
        socketChannel.configureBlocking(false);
        socketChannel.register(selector, SelectionKey.OP_READ,
ByteBuffer.allocate(1024));
    }
    /*
     * 接收信息
     */
    private void handleMsg(SelectionKey key) throws Exception {
        SocketChannel channel = (SocketChannel) key.channel();
        ByteBuffer attachment = (ByteBuffer) key.attachment();
        channel.read(attachment);
```

```
            System.out.println("current msg: " + new String(attachment.array()));
        }

        public static void main(String[] args) throws Exception {
           server myServer = new server();
            myServer.start(8888);
        }
    }
```

client端代码：

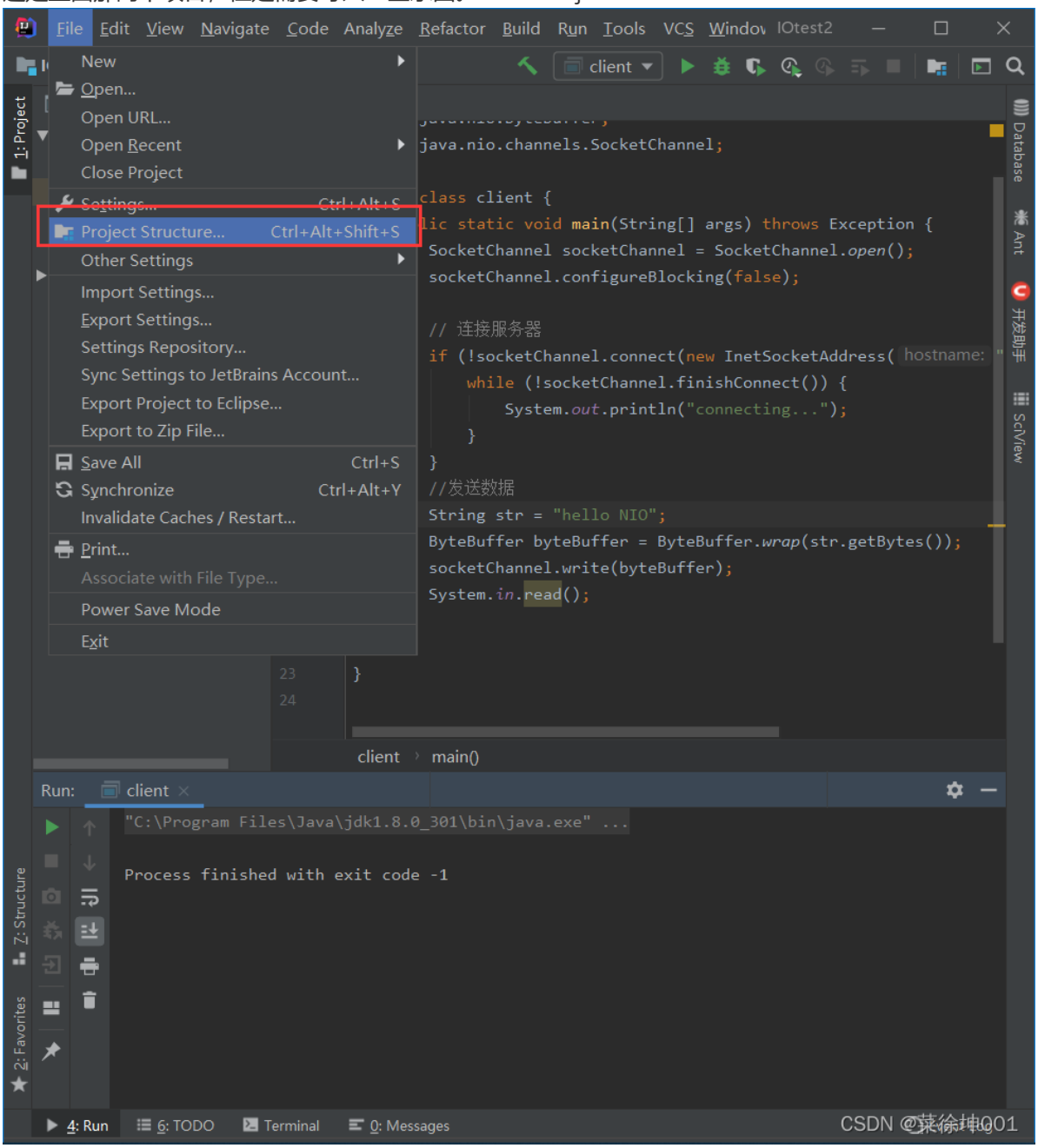
```
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;

public class client {
    public static void main(String[] args) throws Exception {
        SocketChannel socketChannel = SocketChannel.open();
        socketChannel.configureBlocking(false);

        // 连接服务器
        if (!socketChannel.connect(new InetSocketAddress("127.0.0.1", 8888))) {
            while (!socketChannel.finishConnect()) {
                System.out.println("connecting...");
            }
        }
        //发送数据
        String str = "hello netty";
        ByteBuffer byteBuffer = ByteBuffer.wrap(str.getBytes());
        socketChannel.write(byteBuffer);
        System.in.read();
    }

}
```

运行结果：

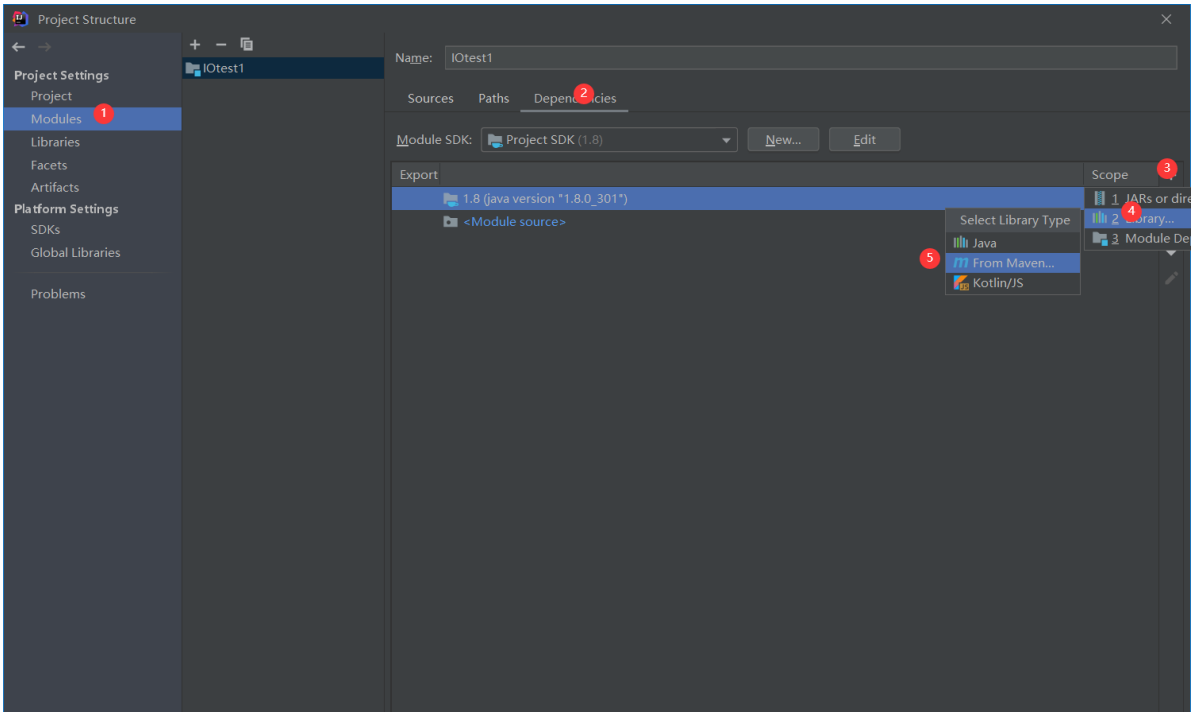


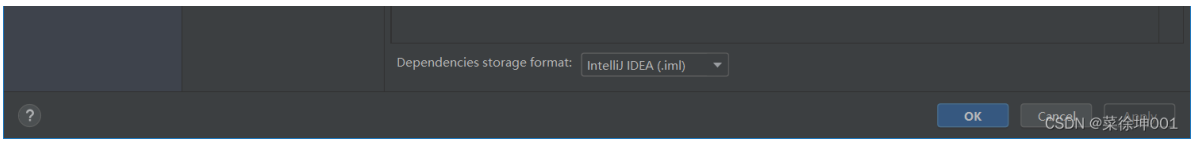
四、Netty示例

还是上面那两个项目，但是需要导入一些东西。 File->Project Structure

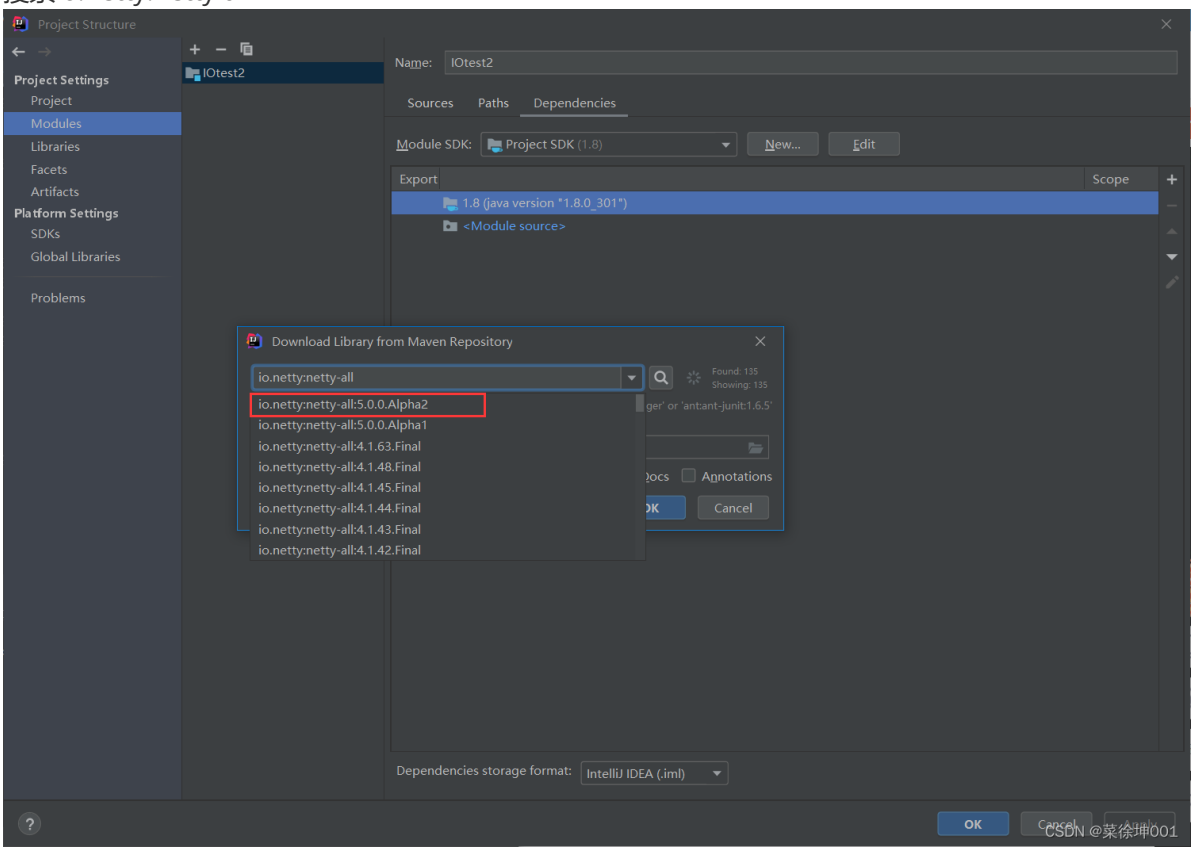


按照序号操作

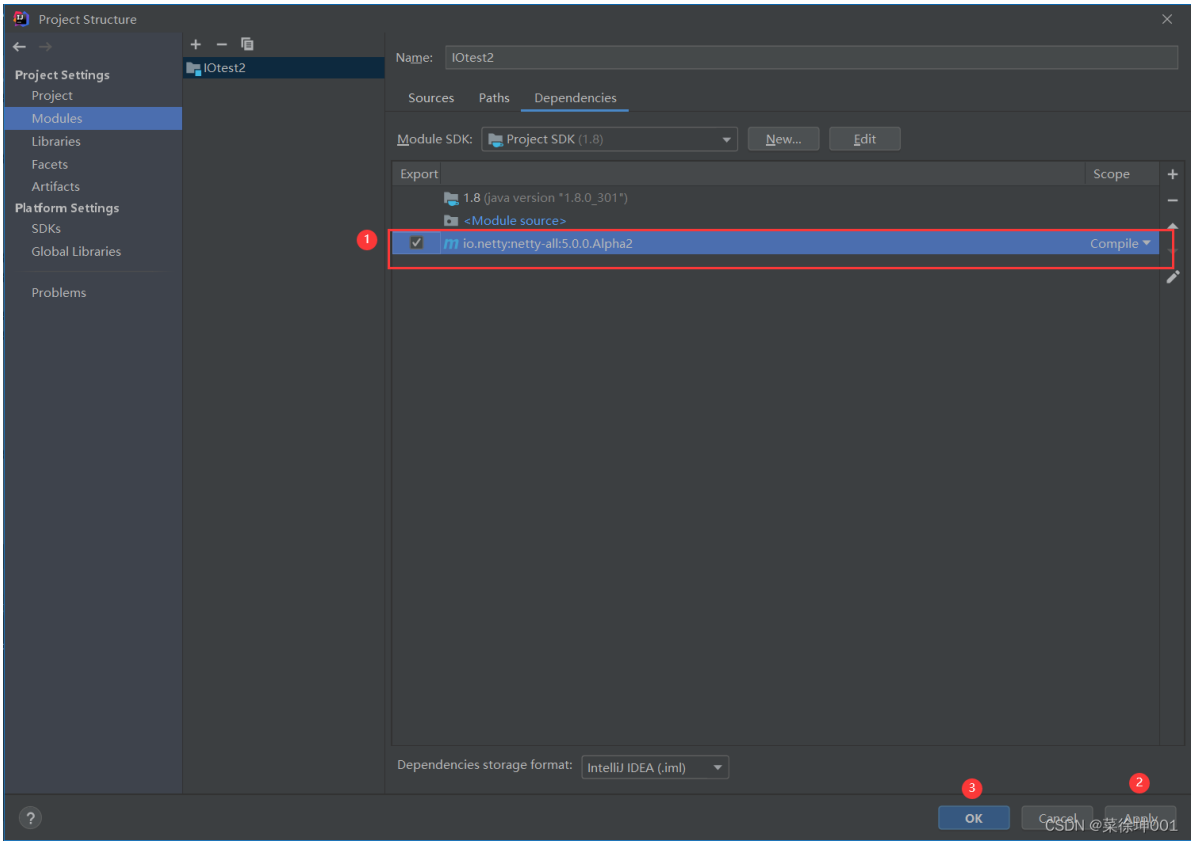




搜索io.netty:netty-all



点击应用



server端代码：

```
import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.*;
import io.netty.channel.nio.NioEventLoopGroup;
```

```
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioServerSocketChannel;
import io.netty.handler.codec.string.StringDecoder;
import io.netty.handler.codec.string.StringEncoder;

import java.net.InetSocketAddress;

/**
 *
 */
public class Server {

    private int port;

    public static void main(String[] args){
        new Server(18080).start();
    }

    public Server(int port) {
        this.port = port;
    }

    public void start() {
        /**
         * 创建两个EventLoopGroup，即两个线程池，boss线程池用于接收客户端的连接，
         * 一个线程监听一个端口，一般只会监听一个端口所以只需一个线程
         * work池用于处理网络连接数据读写或者后续的业务处理（可指定另外的线程处理业务，
         * work完成数据读写）
         */
        EventLoopGroup boss = new NioEventLoopGroup(1);
        EventLoopGroup work = new NioEventLoopGroup();
        try {
            /**
             * 实例化一个服务端启动类，
             * group（）指定线程组
             * channel（）指定用于接收客户端连接的类，对应java.nio.ServerSocketChannel
             * childHandler（）设置编码解码及处理连接的类
             */
            ServerBootstrap server = new ServerBootstrap()
                    .group(boss, work).channel(NioServerSocketChannel.class)
                    .localAddress(new InetSocketAddress(port))
                    .option(ChannelOption.SO_BACKLOG, 128)
                    .childOption(ChannelOption.SO_KEEPALIVE, true)
                    .childHandler(new ChannelInitializer<SocketChannel>() {
                        @Override
                        protected void initChannel(SocketChannel ch) throws
Exception {
                            ch.pipeline()
                                    .addLast("decoder", new StringDecoder())
                                    .addLast("encoder", new StringEncoder())
                                    .addLast(new HelloWorldServerHandler());
                        }
                    });
            //绑定端口
            ChannelFuture future = server.bind().sync();
            System.out.println("server started and listen " + port);
            future.channel().closeFuture().sync();
        } catch (Exception e) {
```

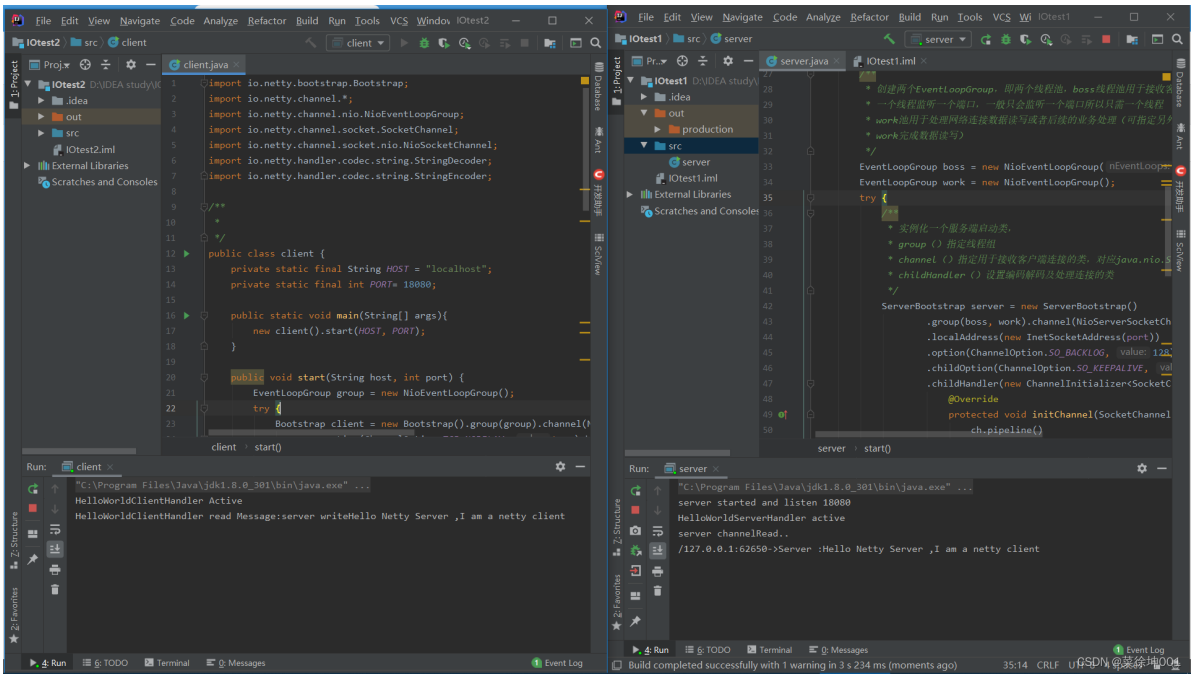
```
            e.printStackTrace();
        }finally {
            boss.shutdownGracefully();
            work.shutdownGracefully();
        }
    }

    public static class HelloWorldServerHandler extends
ChannelInboundHandlerAdapter {

        @Override
        public void channelActive(ChannelHandlerContext ctx) throws Exception {
            System.out.println("HelloWorldServerHandler active");
        }

        @Override
        public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
            System.out.println("server channelRead..");
            System.out.println(ctx.channel().remoteAddress()+"->Server :"+
msg.toString());
            ctx.write("server write"+msg);
            ctx.flush();
        }
    }
}
```

client端代码：

```
import io.netty.bootstrap.Bootstrap;
import io.netty.channel.*;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioSocketChannel;
import io.netty.handler.codec.string.StringDecoder;
import io.netty.handler.codec.string.StringEncoder;

/**
 *
 */
public class Client {
    private static final String HOST = "localhost";
    private static final int PORT= 18080;

    public static void main(String[] args){
        new Client().start(HOST, PORT);
    }

    public void start(String host, int port) {
        EventLoopGroup group = new NioEventLoopGroup();
        try {
            Bootstrap client = new
Bootstrap().group(group).channel(NioSocketChannel.class)
```

```
                    .option(ChannelOption.TCP_NODELAY, true).handler(new
ChannelInitializer<SocketChannel>() {
                        @Override
                        protected void initChannel(SocketChannel ch) throws
Exception {
                            ch.pipeline()
                                    .addLast("decoder", new StringDecoder())
                                    .addLast("encoder", new StringEncoder())
                                    .addLast(new HelloWorldClientHandler());
                        }
                    });
            ChannelFuture future = client.connect(host, port).sync();
            future.channel().writeAndFlush("Hello Netty Server ,I am a netty
client");
            future.channel().closeFuture().sync();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            group.shutdownGracefully();

        }

    }

    public static class HelloWorldClientHandler extends
ChannelInboundHandlerAdapter {
        @Override
        public void channelActive(ChannelHandlerContext ctx) throws Exception {
            System.out.println("HelloWorldClientHandler Active");
        }

        @Override
        public void channelRead(ChannelHandlerContext ctx, Object msg) throws
Exception {
            System.out.println("HelloWorldClientHandler read Message:"+msg);
        }
    }
}
```

运行结果:



五、总结

学习过套接字，再学习IO、NIO就很好理解了，Netty可以看作是NIO的进化版。

六、参考链接

IO与NIO的区别 Netty Helloworld入门