Java 经历过 bio、nio 以及最新的 aio 阶段；

bio：是 java1.0 版本推出的，面向 socket 变成，**是同步阻塞的**，每一个 socket 客户端向服务端 serverSocket 客户端发送请求，这个请求中包含包含客户端发送请求的 socket 信息
服务端都需要创建一个单独的线程去处理请求，服务端从发送请求携带的 socket 里去读、写数据，并且在读、写过程中，该线程一直阻塞着，等请求写完数据再释放线程，这样的话，导致的结果是若有大量的客户端同时去与服务端建立连接，服务端就一下子要创建大量的线程。
为啥叫同步阻塞呢？
Bio 的这个同步阻塞不仅仅针对网络的模型去说的，主要是针对磁盘的 io 读写操作，FileInputStream 文件操作，一直卡在那，直到读写结束。

Nio：java 1.4 版本推出的，它引入 channel 的概念，网络编程是 socketChannel 和 serverSocketChannel，并且是同步非阻塞的。解决 bio 的两大问题：
1、 大量客户端去创建服务端的连接时，服务端不需要去创建很多的线程去处理它，只需要创建一个多了复用器 selector 线程+线程池即可。

客户端 socketChannel 与服务端 ServerSocketChannel 建立连接时，会在服务端创建一个 Channel，服务端监听服务时，就把它注册到多路复用器 selector 上，这样服务端创建的 channel 也注册到 selector 上，大量的客户端建立连接，服务端就创建大量的 channel 对象，然后一个多路复用器线程就去轮训这些 channel，一旦有 channel 可读写，就去线程池里拿出工作线程去处理这个 channel；处理完后再把工作线程放回线程池。

为啥 bio 里不引入线程池呢？若 bio 引入线程池就限制了服务端的可连接数。而在 nio 里，一个线程就可搞定。

2、 nio 是同步非阻塞的，就是说你可以通过 FileChannel 去发起 io 操作，发起之后就返回了，工作线程可以去干其他的事情，这就是非阻塞的；但是接下来工作线程还是要定时的去轮训，检查 io 操作是否完成。

3、nio 中从客户端读写数据，不是直接从 socketchannel 中获取的，需要中间的 buffer 进行，读的时候需要先读到 buffer 中，然后再从 buffer 中进行 io 操作，写的时候也是需要先从磁盘 io 中写入 buffer 中去，在写入 channel 中返回数据；

Aio：java 1.7 推出的，和 nio 类似，但他是异步非阻塞的，工作线程通过 AsynchronousFileChannel 发起 io 操作后，工作线程可以去干其他的事情，并且不需要去轮训 io 操作的结果，由操作系统去完成 io 操作的检查，一旦完成 io 的操作，在回调工作线程去继续处理。

下面对给 nio 的两个 demo：

Demo1、多个客户端线程，一个服务端的 selector 线程，多个写处理线程：

```
package com.study.kafka.nio.demo1;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.SocketChannel;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

/**
* @author:zhangfd
* @version:1.0
* @date:2017 年 12 月 7 日  下午 10:09:13
* @description:
*/
public class MySocketClient {

    public static final String    CHAR_SET = "UTF-8";

    private String ip;

    private int port;

    private SocketChannel socketChannel;

    private Selector sel;

    public MySocketClient() throws IOException {

    }

     public MySocketClient(String ip, int port) throws IOException {
         this();
        this.ip = ip;
        this.port = port;

    }

     private void init() throws IOException{
         // 创建 socketChannel 对象
         socketChannel = SocketChannel.open();
         sel = Selector.open();
```

```
        //绑定网络
        socketChannel.connect(new InetSocketAddress(ip,port));
        socketChannel.configureBlocking(false);
        //注册读事件,把 socketChannel 注册到选择器上
        SelectionKey selectionKey = socketChannel.register(sel, SelectionKey.OP_READ, ByteBuffer.allocate(1024));
        selectionKey.attach("test....."+Thread.currentThread().getName());
        //另外启动一个线程等待服务端返回的结果,主线程可以其做其他事件
        new SocketClientReadThread(sel);


    }

    public synchronized void sendMessage(String message) {
        try {
            init();//放在这里每次都创建一个网络连接
            ByteBuffer bf = ByteBuffer.allocate(message.getBytes().length);
            bf.clear();
            bf.put(message.getBytes(CHAR_SET));
            bf.flip();
            socketChannel.write(bf);//写数据给服务端
            bf=null;

        }catch (Exception e){
                System.out.println(e.getMessage());
        }

    }

    public static void main(String[] args) throws Exception {

        MySocketClient client = new MySocketClient("localhost",8888);
        Executor executors = Executors.newFixedThreadPool(10);
        for (int i=0; i <10; i++){
            executors.execute(() -> client.sendMessage(Thread.currentThread().getName()+": client:-->>>>  我是好人！ "));
        }

    }

}
```

```
package com.study.kafka.nio.demo1;
```

```
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.SocketChannel;
import java.nio.charset.Charset;
import java.util.Iterator;

/**
 * @author:zhangfd
 * @version:1.0
 * @date:2017 年 12 月 7 日  下午 11:22:31
 * @description:
 */
public class SocketClientReadThread implements Runnable {

    private Selector selector;

    public SocketClientReadThread() {

    }

    public SocketClientReadThread(Selector sel) {
        this.selector = sel;
        new Thread(this).start();

    }

    public void run() {

        try {

            //主要选择器中有可读事件存在
            while (selector.select() > 0) {

                //获取所有可读事件集合
                Iterator<SelectionKey> selectionKeys = selector.selectedKeys().iterator();
                while (selectionKeys.hasNext()) {
                    SelectionKey key = selectionKeys.next();
                    System.out.println("selectionKey attachment..."+key.attachment());
                    selectionKeys.remove();//把事件从集合中清楚，避免下次重复处理
                    handleKeys(key);
                }
```

```
                }
            } catch (IOException ex) {
                ex.printStackTrace();
            }

        }

        private void handleKeys(SelectionKey sk) throws IOException {
            // 如果该 SelectionKey 对应的 Channel 中有可读的数据
            if (sk.isReadable()) {
                // 使用 NIO 读取 Channel 中的数据
                SocketChannel sc = (SocketChannel) sk.channel();
                ByteBuffer buffer = ByteBuffer.allocate(1024);
                        //(ByteBuffer) sk.attachment();
                sc.read(buffer);
                buffer.flip();

                // 将字节转化为为 UTF-8 的字符串
                String receivedString =
Charset.forName(MySocketClient.CHAR_SET).newDecoder().decode(buffer).toString();

                // 控制台打印出来
                System.out.println("接收到来自服务器" + sc.socket().getRemoteSocketAddress() + "的信息:" +
receivedString);

                // 为下一次读取作准备
                //sk.interestOps(SelectionKey.OP_READ);
            }else{
                System.out.println("异常事件不做处理.....");
            }
        }


}
```

服务端代码:

```
package com.study.kafka.nio.demo1;

import java.io.IOException;
import java.net.InetAddress;
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.ArrayList;
import java.util.Iterator;
import java.util.Random;
import java.util.Set;
import java.util.concurrent.ConcurrentHashMap;

public class Server {
    private static int PORT = 8888;
    private static ByteBuffer echoBuffer = ByteBuffer.allocate(1024);
    private static ByteBuffer sendBuffer = ByteBuffer.allocate(256);



    public static void main(String args[]) throws Exception {
        Selector selector = Selector.open();
        // Open a listener on each port, and register each one
        ServerSocketChannel ssc = ServerSocketChannel.open();
        ssc.configureBlocking(false);
        InetSocketAddress address = new InetSocketAddress("localhost", PORT);
        ssc.bind(address);
        ssc.register(selector, SelectionKey.OP_ACCEPT);
        System.out.println("Going to listen on " + PORT);
        while (selector.select() > 0) {
            Iterator<SelectionKey> selectionKeys = selector.selectedKeys().iterator();
            while (selectionKeys.hasNext()) {
                SelectionKey key = selectionKeys.next();
                selectionKeys.remove();
                handleKeys(selector, key);
            }
        }



    }

    private static void handleKeys(Selector selector, SelectionKey key) throws IOException {
```

```
            if (key.isAcceptable()) {
                ServerSocketChannel sscNew = (ServerSocketChannel) key.channel();
                SocketChannel sc = sscNew.accept();
                sc.configureBlocking(false);
                System.out.println(Thread.currentThread().getName()+"----isAcceptable");
                // Add the new connection to the selector
                sc.register(selector, SelectionKey.OP_READ);
                //key.interestOps(SelectionKey.OP_READ);
            } else if (key.isReadable()) {
                System.out.println(Thread.currentThread().getName()+"----isReadable");
                String msg = new String();
                SocketChannel sc = (SocketChannel) key.channel();
                int code = 0;
                while ((code = sc.read(echoBuffer)) > 0) {
                    byte b[] = new byte[echoBuffer.position()];
                    echoBuffer.flip();
                    echoBuffer.get(b);
                    msg += new String(b, "UTF-8");
                }
                //client 关闭时，收到可读事件，code = -1
                if (code == -1 || msg.toUpperCase().indexOf("BYE") > -1) {
                    sc.close();
                } else {
                    //code=0，消息读完或者 echoBuffer 空间不够时，部分消息内容下一次 select 后收到
                    echoBuffer.clear();
                }
                System.out.println("msg: " + msg + " from: " + sc + "code:    " + code);
                //注册可写通知
                sc.register(selector, SelectionKey.OP_WRITE);

                //key.interestOps(key.interestOps() & (~SelectionKey.OP_WRITE));

            } else if (key.isWritable()) {
                SocketChannel client = (SocketChannel) key.channel();
                //工作线程做的事，不影响主线程，等业务线程处理完后把结果返回到主线程，主线程再把结果推给服务端
                new Thread(()->Server.write(client)).start();
                //写就绪相对有一点特殊，一般来说，你不应该注册写事件。写操作的就绪条件为底层缓冲区有空闲空间，而写缓冲区绝大部分时间都是有空闲空间的，所以当你注册写事件后，写操作一直是就绪的，选择处理线程全占用整个 CPU 资源。所以，只有当你确实有数据要写时再注册写操作，并在写完以后马上取消注册。
                key.interestOps(key.interestOps() & (~SelectionKey.OP_WRITE));
            }
        }

        private synchronized     static void write(SocketChannel client)    {
```

```
            try {


                String sendTxt = Thread.currentThread().getName()+"Message from Server";
                sendBuffer.clear();
                sendBuffer.put(sendTxt.getBytes());
                sendBuffer.flip();
                int code = 0;


                while ((code=client.write(sendBuffer)) != 0) {
                }
                if (code == -1) { //不能直接关闭client，否则客户端会一直读取数据
                    client.close();
                } else {
                    sendBuffer.clear();
                }
                System.out.println(Thread.currentThread().getName()+"Send message to client ");
            }catch (Exception e){

                System.out.println(e);
            }

        }
}
```

在 demo1 中，我们只把写操作使用其他线程去操作，demo2 中，把读写业务操作都放入其他线程去操作，客户端代码不变：

```
package com.study.kafka.nio.demo1.demo2;


import java.io.IOException;
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;


public class Server {
```

```
    private static int PORT = 8888;
    private static ByteBuffer echoBuffer = ByteBuffer.allocate(1024);
    private static ByteBuffer sendBuffer = ByteBuffer.allocate(256);



    public static void main(String args[]) throws Exception {
        Selector selector = Selector.open();
        // Open a listener on each port, and register each one
        ServerSocketChannel ssc = ServerSocketChannel.open();
        ssc.configureBlocking(false);
        InetSocketAddress address = new InetSocketAddress("localhost", PORT);
        ssc.bind(address);
        ssc.register(selector, SelectionKey.OP_ACCEPT);
        System.out.println("Going to listen on " + PORT);
        while (selector.select() > 0) {
            Iterator<SelectionKey> selectionKeys = selector.selectedKeys().iterator();
            while (selectionKeys.hasNext()) {
                SelectionKey key = selectionKeys.next();
                selectionKeys.remove();
                //业务处理线程....
                handleKeys(selector,key);
            }
        }


    }


    private static void handleKeys(Selector selector, SelectionKey key) throws IOException {
        if (key.isAcceptable()) {
            ServerSocketChannel sscNew = (ServerSocketChannel) key.channel();
            SocketChannel sc = sscNew.accept();
            sc.configureBlocking(false);
            System.out.println(Thread.currentThread().getName()+"----isAcceptable");
            // Add the new connection to the selector
            sc.register(selector, SelectionKey.OP_WRITE);


        } /*else if (key.isReadable()) {
            SocketChannel sc = (SocketChannel) key.channel();
            sc.register(selector, SelectionKey.OP_WRITE);
        }*/ else if (key.isWritable()) {
            SocketChannel client = (SocketChannel) key.channel();
            //工作线程做的事，不影响主线程，等业务线程处理完后把结果返回到主线程，主线程再把结果推给服务端
            new Thread(()->Server.write(client)).start();
            //写就绪相对有一点特殊，一般来说，你不应该注册写事件。写操作的就绪条件为底层缓冲区有空闲空间，
```

```
而写缓冲区绝大部分时间都是有空闲空间的，所以当你注册写事件后，写操作一直是就绪的，选择处理线程全占用整个
CPU 资源。所以，只有当你确实有数据要写时再注册写操作，并在写完以后马上取消注册。
                key.interestOps(key.interestOps() & (~SelectionKey.OP_WRITE));
            }
        }

        private synchronized     static void write(SocketChannel client)    {
            try {

                //1、读取消息
                String msg = new String();
                int code = 0;
                while ((code = client.read(echoBuffer)) > 0) {
                    byte b[] = new byte[echoBuffer.position()];
                    echoBuffer.flip();
                    echoBuffer.get(b);
                    msg += new String(b, "UTF-8");
                }
                //client 关闭时，收到可读事件，code = -1
                if (code == -1 || msg.toUpperCase().indexOf("BYE") > -1) {
                    client.close();
                } else {
                    //code=0，消息读完或者 echoBuffer 空间不够时，部分消息内容下一次 select 后收到
                    echoBuffer.clear();
                }
                System.out.println("msg: " + msg + " from: " + client + "code:    " + code);

                //2、业务处理

                //3、写消息
                String sendTxt = Thread.currentThread().getName()+"Message from Server";
                sendBuffer.clear();
                sendBuffer.put(sendTxt.getBytes());
                sendBuffer.flip();
                int code1 = 0;



                while ((code1=client.write(sendBuffer)) != 0) {
                }
                if (code1 == -1) { //不能直接关闭 client，否则客户端会一直读取数据
                    client.close();
                } else {
                    sendBuffer.clear();
                }
```

```
                System.out.println(Thread.currentThread().getName()+"Send message to client ");
            }catch (Exception e){


                System.out.println(e);
            }


    }
}
```

如果在进一步的把业务处理(读写操作)放入线程池就去操作，这样就可以保证在 nio 模型中，虽然有大量的客户端线程，服务端只需要 1+线程池个线程即可满足业务处理。