Java 网络编程入门

目录

BIO 与 NIO

Java 网络编程入门的目标是为了解如何使用 Java 语言来实现网络交互。

在了解 Java 网络模型前，我们需要了解以下两种概念：BIO（同步阻塞 IO）和 NIO（同步非阻塞 IO）

✧ **同步阻塞 IO**

即线程处理网络 IO 事件时采用的是阻塞处理的方式，当前事件处理完成之前是不允许处理其它网络事件的，如果没有事件发生，也要阻塞等待事件发生。

✧ **同步非阻塞 IO**

即非阻塞的处理网络 IO 事件，这个也很好理解。就是当前如果没有要处理的事件，就继续执行线程去处理其它网络事件。

很明显，阻塞 IO 的效率是十分低下的，在网络编程中会存在大量的网络事件需要处理，如果因为事件未发生而产生线程阻塞是我们不希望看见的，阻塞 IO 浪费了线程资源。

通常我们讲的网络编程都是非阻塞 IO 模型。这里我们就讲讲同步非阻塞 IO 下的网络模型编程，当然更进阶的还有异步非阻塞 IO（AIO），由于 AIO 编程实现比较复杂，如果感兴趣可以自行学习。

入门案例

在学习计算机网络课程中，我们学习过**服务端/客户端**模式，即客户端主动向服务端建立 tcp 连接，向服务端发送消息，服务端接收消息并处理消息，返回结果给客户端。

这里我们用 java 来编程实现一个服务端类，以及对应的客户端类，来实现二者的网络通信。

概念须知

套接字

即 IP 和端口，在网络模型中，我们通常用套接字来表示网络连接目标的地址，套接字是网络连接的端点，在下面的案例中，我们讲使用 127.0.0.1 （本机）来模拟网络连接。当然如果有公网服务器也能把 IP 号改成公网 ip 或域名。

`new InetSocketAddress(8080)` 这里缺省代表的监听本机的 `8080` 端口
`new InetSocketAddress("47.92.145.165", 8080)` 这里代表的就是 `ip` 为 `47.92.145.165` 机器的 `8080` 端口

Channel & Buffer

✧ **Channel：**数据传输通道
Channel 是一个**对象**，作用是用于**源节点**和**目标节点**的连接，在 javaNIO 中负责缓冲区数据的传递，Channel 本身不存储数据，因此需要配合缓冲区进行传输。

✧ **Buffer：** 缓冲区，暂存数据的区域
NIO 的 Buffer（缓冲区）本质上是一个**内存块**，既可以**写入**数据，也可以从中**读取**数据
Java NIO 中代表缓冲区的 Buffer 类是一个抽象类，位于 java.nio 包中
ByteBuffer 其实就像 byte[]的包装类，提供了一系列方便的字节数组操作方法，这是我的理解。

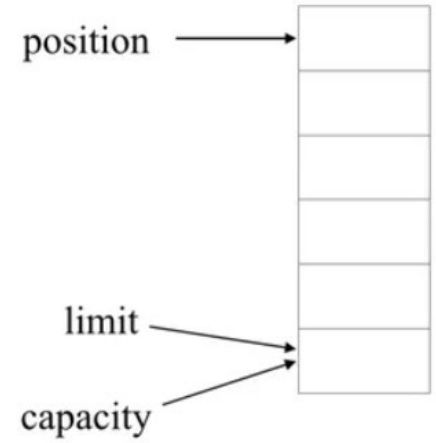
ByteBuffer

对于 ByteBuffer，其主要有五个属性：mark，position，limit，capacity 和 array。这五个属性的作用如下：

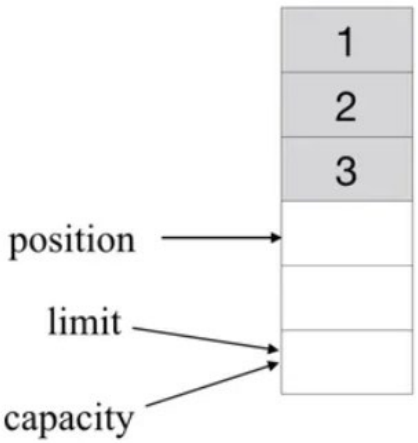
- mark：记录了当前所标记的索引下标；
- position：对于写入模式，表示当前可写入数据的下标，对于读取模式，表示接下来可以读取的数据的下标；
- limit：对于写入模式，表示当前可以写入的数组大小，默认为数组的最大长度，对于读取模式，表示当前最多可以读取的数据的位置下标；
- capacity：表示当前数组的容量大小；
- array：保存了当前写入的数据。

写模式

下面表示 byteBuffer 在写模式下写入三个数据后的状态



初始状态：
position=0，limit=6，capacity=6

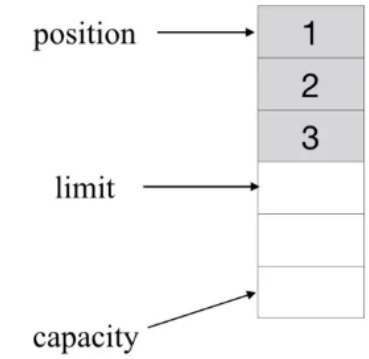


写入三条数据后：
position=3，limit=6，capacity=6

从图中可以看出，在写入模式下，limit 指向的始终是当前可最多写入的数组索引下标，position 指向的则是下一个可以写入的数据的索引位置，而 capacity 则始终不会变化，即为数组大小。

读取模式

假设我们按照上述方式在初始长度为 6 的 ByteBuffer 中写入了三个字节的数据，此时我们将模式切换为读取模式，那么这里的 position，limit 和 capacity 则变为如下形式：



切换为读取模式后：
position=0，limit=3，capacity=6

可以看到，当切换为读取模式之后，limit 则指向了最后一个可读取数据的下一个位置，表示最多可读取的数据；position 则指向了数组的初始位置，表示下一个可读取的数据的位置；capacity 还是表示数组的最大容量。这里当我们一个一个读取数据的时候，position 就会依次往下切换，当期与 limit 重合时，就表示当前 ByteBuffer 中已没有可读取的数据了。

服务端代码

```
public class Server {

    //全局ByteBuffer
    public static ByteBuffer buffer = ByteBuffer.allocate(100);

    public static void main(String[] args) throws IOException {
        //打开一个服务端套接字通道
        ServerSocketChannel ssc = ServerSocketChannel.open();
        //这一步很关键，开启非阻塞模式
        ssc.configureBlocking(false);
        //绑定服务器监听端口
        ssc.bind(new InetSocketAddress(8080));
        //建立连接集合，之后我们要处理的与客户端通信的连接都放在这里
        List<SocketChannel> channels = new ArrayList<>();
        while (true){
            //建立与客户端的连接,该方法为阻塞方法
            //如果我们之前没开启非阻塞模式，线程就会阻塞在这里
            SocketChannel socketChannel = ssc.accept();
            if (socketChannel != null){
                //运行到这里表明与客户端建立起连接了
                System.out.println("connect success!");
                //添加连接
                channels.add(socketChannel);
            }
            //线程处理接收客户端发送的数据
            for (SocketChannel channel : channels) {
                //接收数据调用read 方法
                int read = channel.read(buffer);
                if (read > 0){
                    //切换到读模式，读取buffer 信息
                    buffer.flip();
                    System.out.println("reading...");
System.out.println(Charset.defaultCharset().decode(buffer));
                    //切换回写模式，以便于下次读取
                    buffer.clear();
                }
            }
        }

    }

}
```

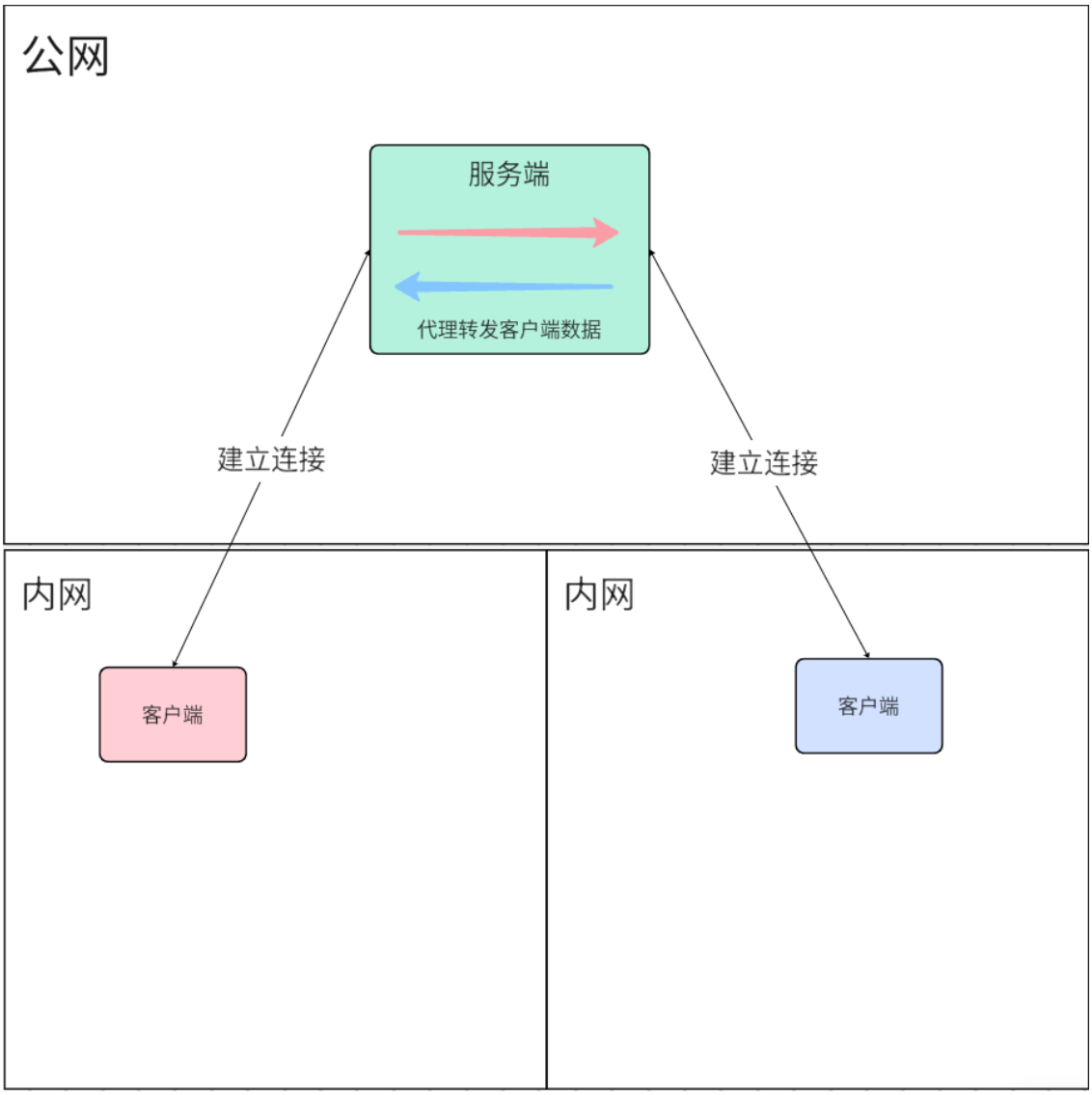
客户端代码

```
public class Client {
    public static void main(String[] args) throws IOException, InterruptedException {
        int i = 0;
        SocketChannel socketChannel = null;
        while (true) {
            try {
                if (socketChannel == null || !socketChannel.isConnected()) {
                    System.out.println("connecting...");
                    //建立连接,这里表明与本机 8080 端口建立连接
                    socketChannel = SocketChannel.open(new InetSocketAddress("127.0.0.1",
8080));
                }
                //发送的消息
                String message = "第"+(i++)+"条消息, Hello Server!";
                ByteBuffer buffer = Charset.defaultCharset().encode(message);
                System.out.println("sending...");
                //写入通道
                socketChannel.write(buffer);
                //保持建立通讯，每隔 4 秒发送一次消息
                Thread.sleep(4000);
            } catch (IOException e) {
                System.out.println("connection failed, retrying...");
                socketChannel = null;
            }
        }
    }
}
```

进阶案例

经过上面入门案例代码的学习，我们掌握了，服务端和客户端基本通信的 Java 代码实现方法。下面我们尝试写一个服务端代理下的聊天系统。

原理图



由于我们的客户端是在各个局域网内的（不具有公网 ip），因此我们通过一个具有（公网 ip）的服务端进行客户端数据之间的代理转发，以实现不同客户端之间的聊天通信。这也是内网穿透的基本原理。

如果没有公网 ip 服务器的同学也可以使用本机 ip 127.0.0.1 代表服务器来模拟代理转发

服务端代码

```
public class ServerListener {
    public static void main(String[] args) {
        try {
            //打开服务端套接字通道
            ServerSocketChannel ssc = ServerSocketChannel.open();
            //监听本地端口
            ssc.bind(new InetSocketAddress(8080));
            //设置非阻塞模式，accept 不阻塞
            ssc.configureBlocking(false);
            //存储的与客户端套接字处理列表，套接字哈希表
            List<SocketChannel> socketChannelList = new ArrayList<>();
            ConcurrentHashMap<String,SocketChannel> socketChannelMap = new
ConcurrentHashMap();
            while (true){
                //处理连接事件
                SocketChannel socketChannel = ssc.accept();
                if (socketChannel != null){
                    System.out.println("connect success!");
                    //与客户端的套接字通道设置非阻塞
                    socketChannel.configureBlocking(false);
                    //建立连接成功，将套接字通道加入处理列表
                    socketChannelList.add(socketChannel);
                }
                //处理连接通道消息
                C: for (SocketChannel channel : socketChannelList) {
                    //创建服务端缓冲区
                    ByteBuffer buffer = ByteBuffer.allocate(100);
                    int read;
                    try {
                        //读取客户端套接字通道信息
                        read = channel.read(buffer);
                    }catch (IOException e){
                        //与客户端发生连接异常，一般为断开连接
                        read = 0;
                        //遍历套接字表，将其信息移除
                        Iterator<Map.Entry<String, SocketChannel>> iterator =
socketChannelMap.entrySet().iterator();
                        while (iterator.hasNext()) {
                            Map.Entry<String, SocketChannel> next = iterator.next();
                            if (next.getValue() == channel){
                                //移除套接字表该套接字通道项
                                socketChannelMap.remove(next.getKey());
                                break;
                            }
                        }
                        //将套接字通道移出处理列表
                        Iterator<SocketChannel> iterator1 = socketChannelList.iterator();
                        while (iterator1.hasNext()) {
                            SocketChannel next = iterator1.next();
```

```
                        if (next == channel){
                            iterator1.remove();
                            System.out.println("remote client closed...");
                            //打破本次处理循环，避免因为移除元素造成的遍历异常
                            break C;
                        }
                    }
                }
                if (read > 0){
                    //缓冲区切换为读模式，读取套接字通道信息
                    buffer.flip();
                    CharBuffer decode = Charset.forName("utf-8").decode(buffer);
                    //解析消息，消息规定以;符号分割
                    String[] split = decode.toString().split(";");
                    if ("connect".equals(split[0])){
                        //该消息为初次连接消息，将用户标识和对应通道放入哈希表，以便服务端识别
                        System.out.println("register-user:"+split[1]);
                        socketChannelMap.put(split[1],channel);
                    }else if ("msg".equals(split[0])){
                        //读取客户端发生的消息消息，服务端为其进行转发，获取到消息中存储的目标用
户消息
                        //获取到目标用户套接字通道
                        SocketChannel sc = socketChannelMap.get(split[2]);
                        System.out.println("proxy...");
                        ByteBuffer encode = Charset.forName("utf-8").encode("来自用户
"+split[1] + "的消息" + ":" + split[3] + "\n");
                        if (sc == null){
                            //未找到对应用户 id 的套接字通道
                            System.out.println("target user is offline");
                        }else {
                            //写入目标用户套接字通道
                            sc.write(encode);
                        }
                    }

                    //切换回写模式，以便于下次读取
                    buffer.clear();
                }
            }

        }

    } catch (IOException ioException) {
        ioException.printStackTrace();
    }


}

}
```

客户端代码

```
public class Client {

    public static void main(String[] args) {

        SocketChannel socketChannel = null;

        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String id = null;
        //创建缓冲区
        ByteBuffer bf = ByteBuffer.allocate(100);
        while (true){
            try {
                if (socketChannel == null || !socketChannel.isConnected()){
                    //与对应主机端口建立 tcp 连接
                    socketChannel = SocketChannel.open(new
InetSocketAddress("localhost",8080));
                    //设置该套接字通道非阻塞 write，read 方法不阻塞
                    socketChannel.configureBlocking(false);

                    //客户端用户输入本次连接识别 id
                    System.out.println("连接 id:");
                    id = br.readLine();
                    //要向服务端发送连接信息
                    ByteBuffer buffer = Charset.forName("utf-8").encode("connect;"+id);
                    //写入通道
                    socketChannel.write(buffer);
                }

                System.out.println("消息模式:读/写(0/1):");
                String mode = br.readLine();
                if (mode.equals("0")){
                    //读取通道内数据
                    int read = socketChannel.read(bf);
                    if (read > 0){
                        //切换到读模式
                        bf.flip();
                        //读取消息
                        System.out.println("接收到消息:");
                        String s = Charset.forName("utf-8").decode(bf).toString();
                        String[] split = s.split("\n");
                        for (String s1 : split) {
                            System.out.println(s1);
                        }
                        //清除缓存区，切换写模式
                        bf.clear();
                    }
                }else {
```

```
                    //用户输入要发送的信息
                    System.out.println("传输目标:");
                    String target = br.readLine();
                    System.out.println("传输消息:");
                    String msg = br.readLine();

                    //封装消息对象
                    Message message = new Message(id,target,msg);

                    System.out.println("Client:sending msg...");

                    //要发送的消息信息
                    ByteBuffer buffer = Charset.forName("utf-8").encode(message.toString());
                    //写入通道
                    socketChannel.write(buffer);
                }
            } catch (IOException ioException) {
                //发送连接异常尝试重新连接
                System.out.println("connection failed, retrying...");
                socketChannel = null;
            }
        }

    }

}

class Message {

    private String id;

    private String target;

    private String msg;

    public Message(String id,String target, String msg) {
        this.id = id;
        this.target = target;
        this.msg = msg;
    }

    @Override
    public String toString() {
        return "msg;" + id + ";" + target + ";" + msg;
    }
}
```

简单消息格式设计
真实网络中的消息传递协议十分复杂，且规定严格，这里我们为了方便教学，使用一种较为简单的消息格式设计。
将消息所携带的参数信息用分号隔开即可。第一个参数标识提供服务端识别要对消息进行的具体操作。之后的参数为本条消息携带的各种信息。

可以看到，在服务端客户端之间的消息通信主要有两种:

- 一种是 connect 消息，客户端在与服务端初次连接时需要携带 connect;id 的消息格式以提供服务端识别客户端的标识。
  头部标识 connect 代表是初次连接消息
  第二个参数 id 代表初次连接本套接字代表的客户端用户 id

- 另外一种是 msg 消息。
  普通的通信消息由以下几个部分构成 msg;id;target;context
  每个部分都用分号隔开，头部标识 msg，告诉服务端本条消息是要发送给客户端的消息
  第二个参数 id 代表发送用户 id
  第三个参数 target 代表目标用户 id
  第四个参数是用户发送的消息具体内容

优化服务端

在上面的简单聊天系统中，我们的服务端设计其实使用的是一个线程去循环处理连接事件，遍历所有客户端套接字通道去读取通道消息。但实际上大部分时间并没有发生连接或消息传输事件，网络是处于 IO 空闲状态的。由于线程并不知道是否发生网络 IO 事件，只能去循环遍历，这种现象是十分浪费 CPU 资源的。因此我们引入多路复用的概念。

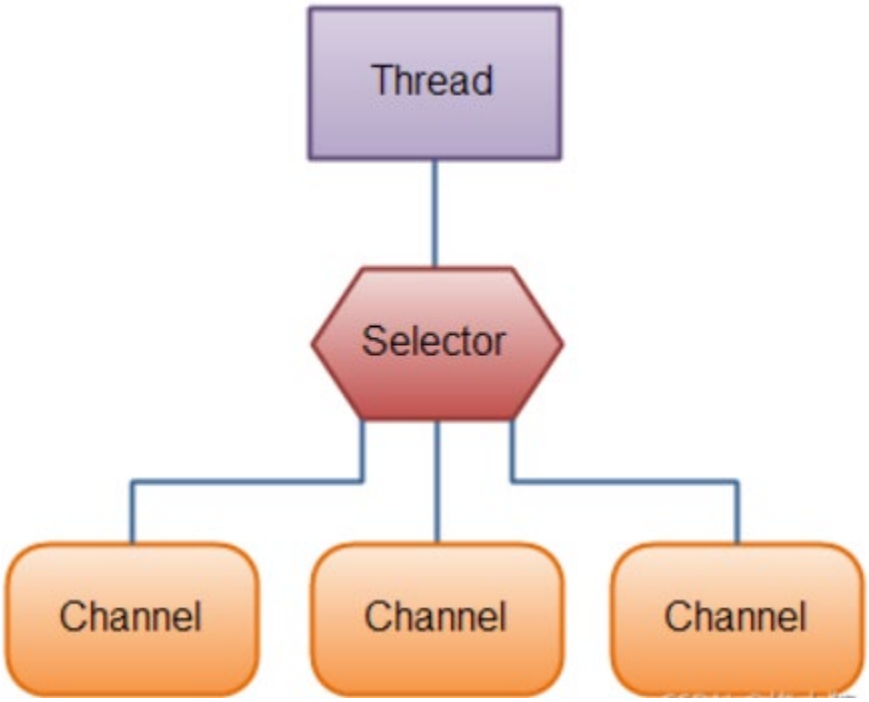
Selector 实现多路复用

为什么使用选择器?

Selector 中能够检测到一到多个 NIO 通道，并能够知道通道是否为读写事件做好准备
这样，一个单独的线程可以管理多个 Channel，从而管理多个网络连接

原来的设计一个线程只管理一个 socket，这样子会造成线程开销的内存浪费，线程上下文切换成本高，只适合连接数少的场景

因此我们要让一个线程管理多个连接就需要使用 Selector，适合连接数多，但是流量低的场景
由于阻塞模式下，线程只能处理一个 socket 连接，导致了线程利用率不高
引入 selector 可以实现单线程的非阻塞式多信道管理，实现了**多路复用**

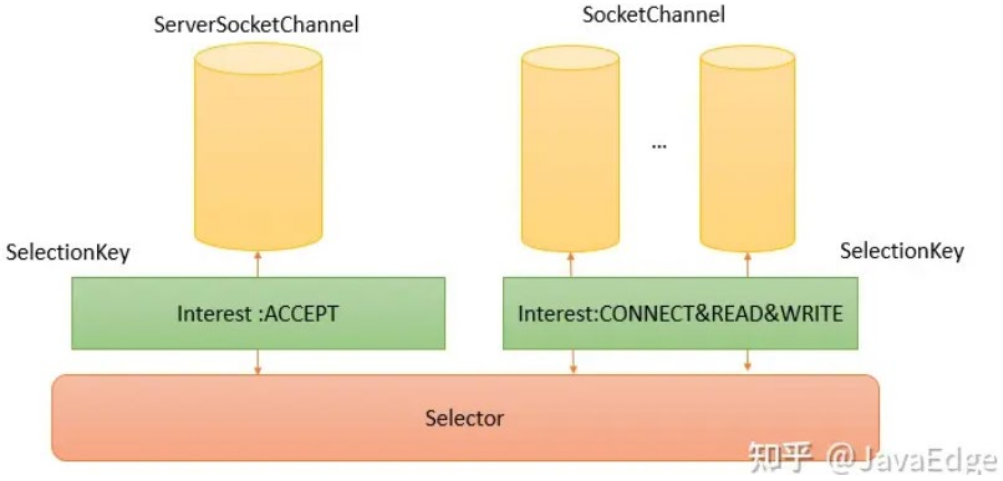


Selector 模型讲解

Selector 模型
引入 Selector 后，需要将之前创建的一或多个可选择的 Channel 注册到 Selector 对象，一个键(SelectionKey)将会被返回。 SelectionKey 会记住你关心的 Channel，也会追踪对应的 Channel 是否已就绪。

简单来说 Selector 模型会管理所有注册在 Selector 中的套接字通道
而在注册的时候，你需要为 Selector 模型注册该套接字通道被关注的事件
例如一般我们会为服务器套接字通道注册 Accept 事件，因为我们关心服务器和客户端连接时发生的事件。
我们还会为客户端套接字通道注册 Read 事件，因为我们的服务端关系来自客户端的消息，也就是可读事件。

我们使用 select 方法来阻塞线程，直到 Selector 模型发现事件时，Selector 模型会将发生的事件放入 SelectedKeys 集合中，此时线程将不再阻塞，我们将遍历事件集合，处理每个事件。



Selector 优化后的服务端代码

```
public class ServerListener {

    private static ConcurrentHashMap<String,SocketChannel> socketChannelMap;

    public static void main(String[] args) {
        try {
            //打开服务端套接字通道
            ServerSocketChannel ssc = ServerSocketChannel.open();
            //监听本地端口
            ssc.bind(new InetSocketAddress(8080));
            //设置非阻塞模式，accept 不阻塞
            ssc.configureBlocking(false);
            Selector selector = Selector.open();
            SelectionKey selectionKey = ssc.register(selector, 0, null);
            selectionKey.interestOps(SelectionKey.OP_ACCEPT);
            socketChannelMap = new ConcurrentHashMap();
            while (true){
                //阻塞直到有事件发生
                selector.select();
                //遍获取事件集合
                Set<SelectionKey> selectionKeys = selector.selectedKeys();
                Iterator<SelectionKey> iterator = selectionKeys.iterator();
                while (iterator.hasNext()) {
                    SelectionKey key = iterator.next();
                    //处理完成该事件，移除事件
                    iterator.remove();
                    if (key.isAcceptable()){
                        //处理连接事件
                        //获取发生事件的通道
                        ServerSocketChannel serverSocketChannel = (ServerSocketChannel)
key.channel();
                        //创建连接
                        SocketChannel socketChannel = serverSocketChannel.accept();
                        //通道必须为非阻塞才能注册
                        socketChannel.configureBlocking(false);
                        SelectionKey register = socketChannel.register(selector, 0,
ByteBuffer.allocate(200));
                        //关注读事件
                        register.interestOps(SelectionKey.OP_READ);
                        System.out.println("connect success");
                    }else if (key.isReadable()){
                        //处理读事件
                        SelectableChannel channel = key.channel();
                        if (channel instanceof SocketChannel){
                            SocketChannel sc = (SocketChannel) channel;
                            //拿到附带缓冲区
                            ByteBuffer buffer = (ByteBuffer) key.attachment();
```

```
                        int read = 0;
                        try {
                             read = sc.read(buffer);
                        }catch (IOException e){
                            disconnect(key);
                        }
                        if (read == -1){
                            disconnect(key);
                        }
                        if (read > 0){
                            //处理正常消息
                            buffer.flip();
                            CharBuffer decode = Charset.forName("utf-8").decode(buffer);
                            //解析消息，消息规定以;符号分割
                            String[] split = decode.toString().split(";");
                            if ("connect".equals(split[0])){
                                //该消息为初次连接消息，将用户标识和对应通道放入哈希表，以便服务端识别
                                System.out.println("register-user:"+split[1]);
                                socketChannelMap.put(split[1],sc);
                            }else if ("msg".equals(split[0])){
                                //读取客户端发生的消息消息，服务端为其进行转发，获取到消息中存储的目标用户消息
                                //获取到目标用户套接字通道
                                SocketChannel targetSc = socketChannelMap.get(split[2]);
                                System.out.println("proxy...");
                                ByteBuffer encode = Charset.forName("utf-8").encode("来自用户"+split[1] + "的消息" + ":" + split[3] + "\n");
                                if (targetSc == null){
                                    //未找到对应用户 id 的套接字通道
                                    System.out.println("target user is offline");
                                }else {
                                    //写入目标用户套接字通道
                                    targetSc.write(encode);
                                }
                            }
                            buffer.clear();
                        }
                    }
                }
            }
        } catch (IOException ioException) {
            ioException.printStackTrace();
        }
    }

    private static void disconnect(SelectionKey key) {
        //处理连接断开事件
        SocketChannel channel = (SocketChannel) key.channel();
        for (String s : socketChannelMap.keySet()) {
```

```
            SocketChannel socketChannel = socketChannelMap.get(s);
            if (socketChannel == channel){
                socketChannelMap.remove(s);
                break;
            }
        }
        key.cancel();
        System.out.println("client connect closed");
    }

}
```