day25【NIO、AIO】

今日内容

- NIO
- AIO

教学目标

- [ ] 能够说出Selector选择器的作用
- [ ] 能够使用Selector选择器
- [ ] 能够说出AIO的特点

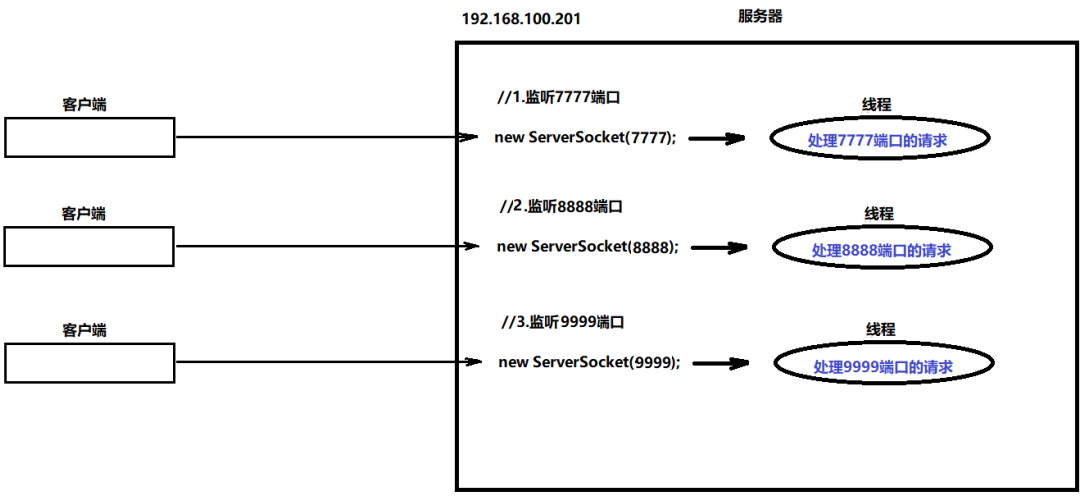
第一章 Selector(选择器)

1.1 多路复用的概念

选择器Selector是NIO中的重要技术之一。它与SelectableChannel联合使用实现了非阻塞的多路复用。使用它可以节省CPU资源，提高程序的运行效率。

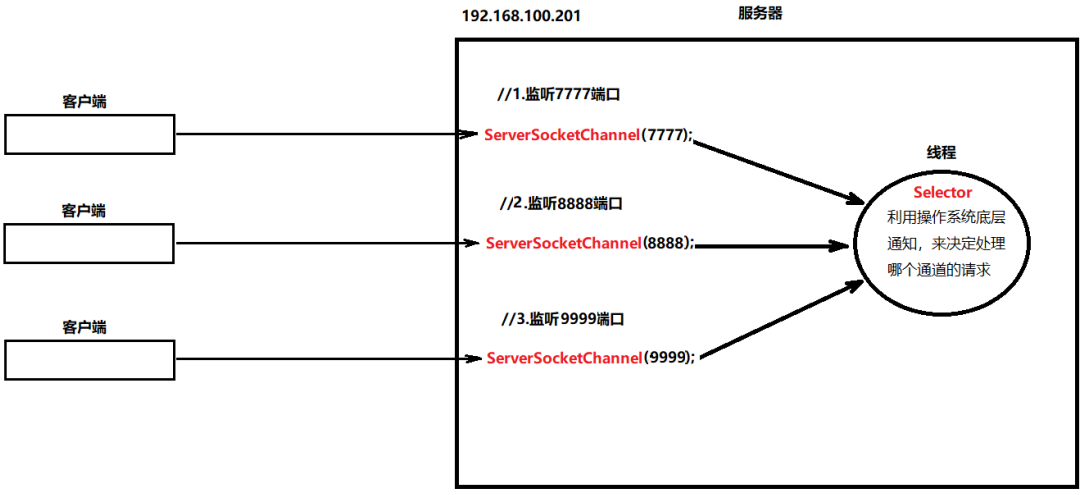
"多路"是指：服务器端同时监听多个“端口”的情况。每个端口都要监听多个客户端的连接。

- 服务器端的非多路复用效果



如果不使用“多路复用”，服务器端需要开很多线程处理每个端口的请求。如果在高并发环境下，造成系统性能下降。

- 服务器端的多路复用效果



使用了多路复用，只需要一个线程就可以处理多个通道，降低内存占用率，减少CPU切换时间，在高并发、高频段业务环境下有非常重要的优势。

1.2 选择器Selector

Selector被称为：选择器，也被称为：多路复用器，它可以注册到很多个Channel上，监听各个Channel上发生的事件，并且能够根据事件情况决定Channel读写。这样，通过一个线程管理多个Channel，就可以处理大量网络连接了。

有了Selector，我们就可以利用一个线程来处理所有的Channel。线程之间的切换对操作系统来说代价是很高的，并且每个线程也会占用一定的系统资源。所以，对系统来说使用的线程越少越好。

- **如何创建一个Selector**

Selector 就是您注册对各种 I/O 事件兴趣的地方，而且当那些事件发生时，就是这个对象告诉您所发生的事件。

```
Selector selector = Selector.open();
```

- **注册Channel到Selector**

为了能让Channel和Selector配合使用，我们需要把Channel注册到Selector上。通过调用channel.register（）方法来实现注册：

```
channel.configureBlocking(false);
SelectionKey key =channel.register(selector,SelectionKey.OP_READ);
```

注意，注册的Channel 必须设置成异步模式才可以,否则异步IO就无法工作，这就意味着我们不能把一个FileChannel注册到Selector，因为FileChannel没有异步模式，但是网络编程中的SocketChannel是可以的。

register()方法的第二个参数：是一个int值，意思是在**通过Selector监听Channel时对什么事件感兴趣。**可以监听四种不同类型的事件，而且可以使用SelectionKey的四个常量表示：

1. 连接就绪--常量：SelectionKey.OP_CONNECT
2. 接收就绪--常量：SelectionKey.OP_ACCEPT (ServerSocketChannel在注册时只能使用此项)
3. 读就绪--常量：SelectionKey.OP_READ
4. 写就绪--常量：SelectionKey.OP_WRITE

注意：对于ServerSocketChannel在注册时，只能使用OP_ACCEPT，否则抛出异常。

- 示例：下面的例子，服务器创建3个通道，同时监听3个端口，并将3个通道注册到一个选择器中。

```
import java.net.InetSocketAddress;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;

public class Server {
    public static void main(String[] args) throws Exception {
        //创建3个通道，同时监听3个端口
        ServerSocketChannel channelA = ServerSocketChannel.open();
        channelA.configureBlocking(false);
        channelA.bind(new InetSocketAddress(7777));

        ServerSocketChannel channelB = ServerSocketChannel.open();
        channelB.configureBlocking(false);
        channelB.bind(new InetSocketAddress(8888));

        ServerSocketChannel channelC = ServerSocketChannel.open();
        channelC.configureBlocking(false);
        channelC.bind(new InetSocketAddress(9999));

        //获取选择器
        Selector selector = Selector.open();

        //注册三个通道
        channelA.register(selector, SelectionKey.OP_ACCEPT);
        channelB.register(selector, SelectionKey.OP_ACCEPT);
        channelC.register(selector, SelectionKey.OP_ACCEPT);
    }
}
```

接下来，就可以通过选择器selector操作三个通道了。

1.3 多路连接

- **Selector的keys()方法**
  - 此方法返回一个Set集合，表示：已注册通道的集合。每个已注册通道封装为一个SelectionKey对象。
- **Selector的selectedKeys()方法**
  - 此方法返回一个Set集合，表示：当前已连接的通道的集合。每个已连接通道统一封装为一个SelectionKey对象。
- **Selector的select()方法**
  - 此方法会阻塞，直到有至少1个客户端连接。
  - 此方法会返回一个int值，表示有几个客户端连接了服务器。
- **示例：** 客户端：启动两个线程，模拟两个客户端，同时连接服务器的7777和8888端口：

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.channels.SocketChannel;
```

```
public class Client {
    public static void main(String[] args) {
        new Thread(()->{
            try (SocketChannel socket = SocketChannel.open()) {
                System.out.println("7777客户端连接服务器......");
                socket.connect(new InetSocketAddress("localhost", 7777));
                System.out.println("7777客户端连接成功....");
                break;
            } catch (IOException e) {
                System.out.println("7777异常重连");
            }

        }).start();

        new Thread(()->{

            try (SocketChannel socket = SocketChannel.open()) {
                System.out.println("8888客户端连接服务器......");
                socket.connect(new InetSocketAddress("localhost", 8888));
                System.out.println("8888客户端连接成功....");
                break;
            } catch (IOException e) {
                System.out.println("8888异常重连");
            }

        }).start();
    }
}
```

服务器端：

```
import java.net.InetSocketAddress;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;


public class Server {
    public static void main(String[] args) throws Exception {
        //创建3个通道，同时监听3个端口
        ServerSocketChannel channelA = ServerSocketChannel.open();
        channelA.configureBlocking(false);
        channelA.bind(new InetSocketAddress(7777));

        ServerSocketChannel channelB = ServerSocketChannel.open();
        channelB.configureBlocking(false);
        channelB.bind(new InetSocketAddress(8888));

        ServerSocketChannel channelC = ServerSocketChannel.open();
        channelC.configureBlocking(false);
        channelC.bind(new InetSocketAddress(9999));

        //获取选择器
        Selector selector = Selector.open();
```

```
            //注册三个通道
            channelA.register(selector, SelectionKey.OP_ACCEPT);
            channelB.register(selector, SelectionKey.OP_ACCEPT);
            channelC.register(selector, SelectionKey.OP_ACCEPT);

            Set<SelectionKey> keys = selector.keys();//获取已注册通道的集合
            System.out.println("注册通道数量：" + keys.size());
            Set<SelectionKey> selectionKeys = selector.selectedKeys();//获取已连接通
道的集合
            System.out.println("已连接的通道数量：" + selectionKeys.size());
            System.out.println("------------------------------------------------");

            System.out.println("【服务器】等待连接......");
            int selectedCount = selector.select();//此方法会"阻塞"
            System.out.println("连接数量：" + selectedCount);

          System.out.println("------------------------------------------------");
            Set<SelectionKey> keys1 = selector.keys();
          System.out.println("注册通道数量：" + keys1.size());
            Set<SelectionKey> selectionKeys1 = selector.selectedKeys();
          System.out.println("已连接的通道数量：" + selectionKeys1.size());
        }
    }
```

1. 先启动服务器，再启动客户端。会看到"服务器端"打印：

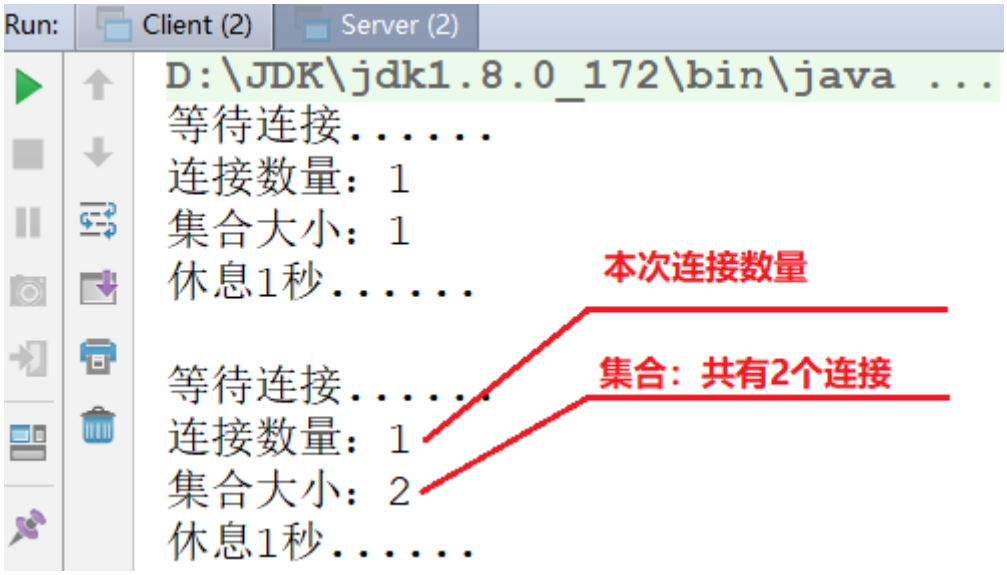
```
注册通道数量：3
已连接的通道数量：0
------------------------------------------------
 【服务器】等待连接......
连接数量：1
------------------------------------------------
 注册通道数量：3
 已连接的通道数量：1
```

在**"服务器端"**加入循环，确保接收到每个通道的连接：(下面的代码去掉了一些测试代码)

```
    public static void main(String[] args) throws Exception {
          //创建3个通道，同时监听3个端口
            ServerSocketChannel channelA = ServerSocketChannel.open();
            channelA.configureBlocking(false);
            channelA.bind(new InetSocketAddress(7777));

            ServerSocketChannel channelB = ServerSocketChannel.open();
            channelB.configureBlocking(false);
            channelB.bind(new InetSocketAddress(8888));

            ServerSocketChannel channelC = ServerSocketChannel.open();
            channelC.configureBlocking(false);
            channelC.bind(new InetSocketAddress(9999));

            //获取选择器
            Selector selector = Selector.open();
```

```
            //注册三个通道
            channelA.register(selector, SelectionKey.OP_ACCEPT);
            channelB.register(selector, SelectionKey.OP_ACCEPT);
            channelC.register(selector, SelectionKey.OP_ACCEPT);

            while(true) {
                System.out.println("等待连接......");
                int selectedCount = selector.select();
                System.out.println("连接数量：" + selectedCount);
                //获取已连接的通道对象
                Set<SelectionKey> selectionKeys = selector.selectedKeys();
                System.out.println("集合大小：" + selectionKeys.size());

                System.out.println("休息1秒......");
                Thread.sleep(1000);
                System.out.println();//打印一个空行

            }

        }
```

2. 先运行"服务器"，再运行"客户端"。服务器打印如下：



注意：此例会有一个问题——服务器端第一次select()会阻塞，获取到一次连接后再次循环时，select()将不会再阻塞，从而造成死循环，所以这里加了一个sleep()，这个我们后边解决！！！

接下来，我们获取"已连接通道"的集合，并遍历：

客户端

使用上例的客户端即可

服务器端

```
import java.net.InetSocketAddress;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
```

```
import java.util.Set;

public class Server {
    public static void main(String[] args) throws Exception {
        //创建3个通道，同时监听3个端口
        ServerSocketChannel channelA = ServerSocketChannel.open();
        channelA.configureBlocking(false);
        channelA.bind(new InetSocketAddress(7777));

        ServerSocketChannel channelB = ServerSocketChannel.open();
        channelB.configureBlocking(false);
        channelB.bind(new InetSocketAddress(8888));

        ServerSocketChannel channelC = ServerSocketChannel.open();
        channelC.configureBlocking(false);
        channelC.bind(new InetSocketAddress(9999));

        //获取选择器
        Selector selector = Selector.open();

        //注册三个通道
        channelA.register(selector, SelectionKey.OP_ACCEPT);
        channelB.register(selector, SelectionKey.OP_ACCEPT);
        channelC.register(selector, SelectionKey.OP_ACCEPT);

        while(true) {
            System.out.println("等待连接......");
            int selectedCount = selector.select();
            System.out.println("连接数量：" + selectedCount);
            //获取已连接的通道对象
            Set<SelectionKey> selectionKeys = selector.selectedKeys();
            System.out.println("集合大小：" + selectionKeys.size());

            //遍历已连接通道的集合
            Iterator<SelectionKey> it = selectionKeys.iterator();
            while (it.hasNext()) {
                //获取当前连接通道的SelectionKey
                SelectionKey key = it.next();
                //从SelectionKey中获取通道对象
                ServerSocketChannel channel = (ServerSocketChannel)
key.channel();
                //看一下此通道是监听哪个端口的
                System.out.println("监听端口：" + channel.getLocalAddress());
            }
            System.out.println("休息1秒......");
            Thread.sleep(1000);
            System.out.println();//打印一个空行

        }

    }
}
```

- 先启动服务器端，再启动客户端。可以看到"服务器端"如下打印：

```
等待连接......
连接数量：1
集合大小：1
监听端口：/0:0:0:0:0:0:0:0:8888
休息1秒......

等待连接......
连接数量：1
集合大小：2
监听端口：/0:0:0:0:0:0:0:0:8888
监听端口：/0:0:0:0:0:0:0:0:7777
休息1秒......
```

- **关于SelectionKey**

```
- 当一个"通道"注册到选择器Selector后，选择器Selector内部就创建一个SelectionKey对象，里面封装了这个通道和这个选择器的映射关系。
- 通过SelectionKey的channel()方法，可以获取它内部的通道对象。
```

- **解决select()不阻塞，导致服务器端死循环的问题**
- 原因：在将"通道"注册到"选择器Selector"时，我们指定了关注的事件SelectionKey.OP_ACCEPT，而我们获取到管道对象后，并没有处理这个事件，所以导致select()方法一直循环。
- 解决：处理SelectionKey.OP_ACCEPT事件

  更改服务器端代码

```
public class Server {
    public static void main(String[] args) throws Exception {
        //创建3个通道，同时监听3个端口
        ...略...

        //获取选择器
        ...略...

        //注册三个通道
        ...略...

        while(true) {
            ......
            ......
            while (it.hasNext()) {
                //获取当前连接通道的SelectionKey
                SelectionKey key = it.next();
                //从SelectionKey中获取通道对象
                ServerSocketChannel channel = (ServerSocketChannel)
key.channel();
                //看一下此通道是监听哪个端口的
                System.out.println("监听端口：" + channel.getLocalAddress());
                SocketChannel accept = channel.accept();//处理accept事件(非阻
塞)
            }
        }
```

```
    }
}
```

现在我们的服务器端可以很好的接收客户端连接了，但还有一个小问题，在接下来的互发信息的例子中我们可以看到这个问题并解决它。

1.4 多路信息接收

- 服务器端代码：

```
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;
import java.util.Set;

public class Server {
    public static void main(String[] args) throws Exception {
        //1.同时监听三个端口：7777,8888,9999
        ServerSocketChannel serverChannel1 = ServerSocketChannel.open();
        serverChannel1.bind(new InetSocketAddress(7777));
        serverChannel1.configureBlocking(false);

        ServerSocketChannel serverChannel2 = ServerSocketChannel.open();
        serverChannel2.bind(new InetSocketAddress(8888));
        serverChannel2.configureBlocking(false);

        ServerSocketChannel serverChannel3 = ServerSocketChannel.open();
        serverChannel3.bind(new InetSocketAddress(9999));
        serverChannel3.configureBlocking(false);

        //2.获取一个选择器
        Selector selector = Selector.open();

        //3.注册三个通道
        SelectionKey key1 = serverChannel1.register(selector,
SelectionKey.OP_ACCEPT);
        SelectionKey key2 = serverChannel2.register(selector,
SelectionKey.OP_ACCEPT);
        SelectionKey key3 = serverChannel3.register(selector,
SelectionKey.OP_ACCEPT);

        //4.循环监听三个通道
        while (true) {
            System.out.println("等待客户端连接...");
            int keyCount = selector.select();
            System.out.println("连接数量: " + keyCount);

            //遍历已连接的每个通道的SelectionKey
            Set<SelectionKey> keys = selector.selectedKeys();
            Iterator<SelectionKey> it = keys.iterator();
```

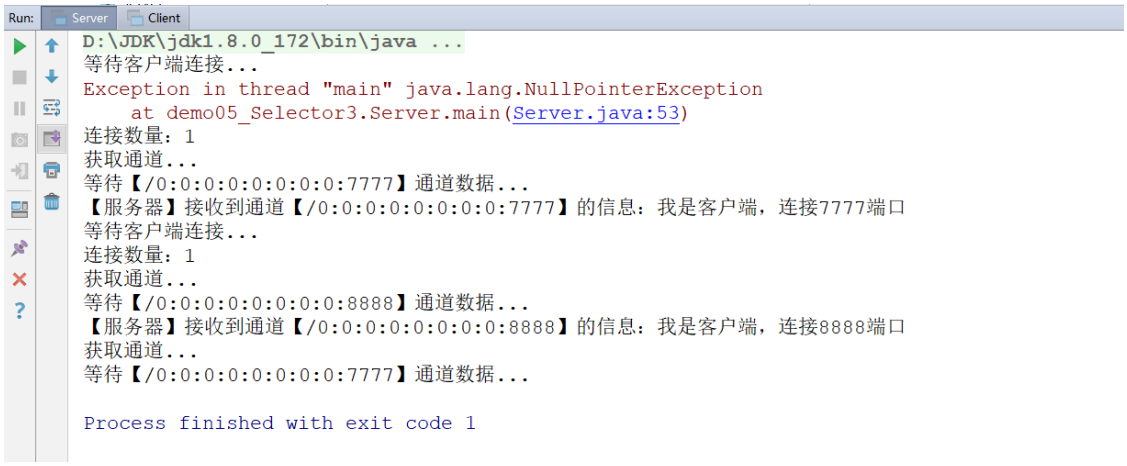
```
            while (it.hasNext()) {
                SelectionKey nextKey = it.next();
                System.out.println("获取通道...");
                ServerSocketChannel channel = (ServerSocketChannel)
nextKey.channel();
                System.out.println("等待【" + channel.getLocalAddress() + "】
通道数据...");
                SocketChannel socketChannel = channel.accept();
                //接收数据
                ByteBuffer inBuf = ByteBuffer.allocate(100);
                socketChannel.read(inBuf);
                inBuf.flip();
                String msg = new String(inBuf.array(), 0, inBuf.limit());
                System.out.println("【服务器】接收到通道【" +
channel.getLocalAddress() + "】的信息：" + msg);

            }
        }
    }
}
```

- 客户端代码：

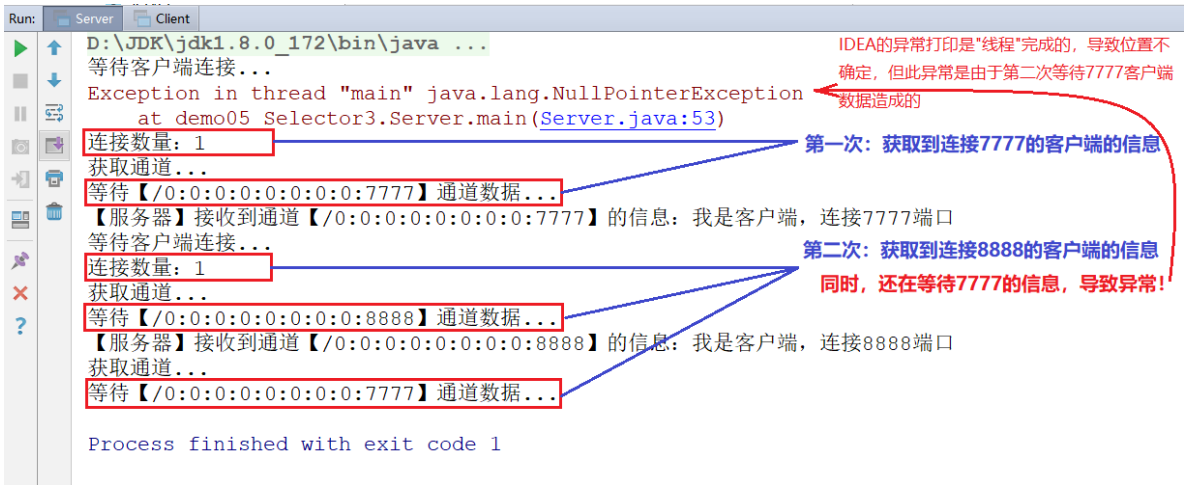
```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;

public class Client {
    public static void main(String[] args) throws InterruptedException {
        //两个线程，模拟两个客户端，分别连接服务器的7777,8888端口
        new Thread(()->{

                try(SocketChannel socket = SocketChannel.open())  {

                    System.out.println("7777客户端连接服务器......");
                    socket.connect(new InetSocketAddress("localhost",
7777));
                    System.out.println("7777客户端连接成功....");
                    //发送信息
                    ByteBuffer outBuf = ByteBuffer.allocate(100);
                    outBuf.put("我是客户端，连接7777端口".getBytes());
                    outBuf.flip();
                    socket.write(outBuf);
                } catch (IOException e) {
                    System.out.println("7777异常重连");
                }

        }).start();
        new Thread(()->{

                try(SocketChannel socket = SocketChannel.open())  {
                    System.out.println("8888客户端连接服务器......");
                    socket.connect(new InetSocketAddress("localhost",
8888));
                    System.out.println("8888客户端连接成功....");
                    //发送信息
```

```
                    ByteBuffer outBuf = ByteBuffer.allocate(100);
                    outBuf.put("我是客户端，连接8888端口".getBytes());
                    outBuf.flip();
                    socket.write(outBuf);

                } catch (IOException e) {
                    System.out.println("8888异常重连");
                }

            }).start();
        }
    }
```

先启动服务器，再启动客户端，打印结果：



可以看到，出现了异常，为什么会这样？



问题就出现在获取selectedKeys()的集合。

- 第一次的7777连接，selectedKeys()获取的集合中只有一个SelectionKey对象。
- 第二次的8888连接，selectedKeys()获取的集合中有2个SelectionKey对象，一个是连接7777客户端的，另一个是连接8888客户端的。而此时应该只处理连接8888客户端的，所以在上一次处理完7777的数据后，应该将其SelectionKey对象移除。

更改服务器端代码：

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

```
        //1.同时监听三个端口：7777,8888,9999
        ServerSocketChannel serverChannel1 = ServerSocketChannel.open();
        serverChannel1.bind(new InetSocketAddress(7777));
        serverChannel1.configureBlocking(false);

        ServerSocketChannel serverChannel2 = ServerSocketChannel.open();
        serverChannel2.bind(new InetSocketAddress(8888));
        serverChannel2.configureBlocking(false);

        ServerSocketChannel serverChannel3 = ServerSocketChannel.open();
        serverChannel3.bind(new InetSocketAddress(9999));
        serverChannel3.configureBlocking(false);

        //2.获取一个选择器
        Selector selector = Selector.open();

        //3.注册三个通道
        SelectionKey key1 = serverChannel1.register(selector,
SelectionKey.OP_ACCEPT);
        SelectionKey key2 = serverChannel2.register(selector,
SelectionKey.OP_ACCEPT);
        SelectionKey key3 = serverChannel3.register(selector,
SelectionKey.OP_ACCEPT);

        //4.循环监听三个通道
        while (true) {
            System.out.println("等待客户端连接...");
            int keyCount = selector.select();
            System.out.println("连接数量：" + keyCount);

            //遍历已连接的每个通道的SelectionKey
            Set<SelectionKey> keys = selector.selectedKeys();
            Iterator<SelectionKey> it = keys.iterator();
            while (it.hasNext()) {
                SelectionKey nextKey = it.next();
                System.out.println("获取通道...");
                ServerSocketChannel channel = (ServerSocketChannel)
nextKey.channel();
                System.out.println("等待【" + channel.getLocalAddress() + "】通道
数据...");
                SocketChannel socketChannel = channel.accept();
                //接收数据
                ByteBuffer inBuf = ByteBuffer.allocate(100);
                socketChannel.read(inBuf);
                inBuf.flip();
                String msg = new String(inBuf.array(), 0, inBuf.limit());
                System.out.println("【服务器】接收到通道【" +
channel.getLocalAddress() + "】的信息：" + msg);
                //移除此SelectionKey
                it.remove();
            }
        }
    }
}
```

测试：先启动服务器，再启动客户端，可以正常接收客户端数据了(客户端可以再添加一个线程连接9999端口)。

第二章 NIO2-AIO(异步、非阻塞)

2.1 AIO 概述

AIO是异步IO的缩写，虽然NIO在网络操作中，提供了非阻塞的方法，但是NIO的IO行为还是同步的。对于NIO来说，我们的业务线程是在IO操作准备好时，得到通知，接着就由这个线程自行进行IO操作，IO操作本身是同步的。

但是对AIO来说，则更加进了一步，它不是在IO准备好时再通知线程，而是在IO操作已经完成后，再给线程发出通知。因此AIO是不会阻塞的，此时我们的业务逻辑将变成一个回调函数，等待IO操作完成后，由系统自动触发。

与NIO不同，当进行读写操作时，只须直接调用API的read或write方法即可。这两种方法均为异步的，对于读操作而言，当有流可读取时，操作系统会将可读的流传入read方法的缓冲区，并通知应用程序；对于写操作而言，当操作系统将write方法传递的流写入完毕时，操作系统主动通知应用程序。 即可以理解为，read/write方法都是异步的，完成后会主动调用回调函数。 在JDK1.7中，这部分内容被称作NIO.2，主要在Java.nio.channels包下增加了下面四个异步通道：

- AsynchronousSocketChannel
- AsynchronousServerSocketChannel
- AsynchronousFileChannel
- AsynchronousDatagramChannel

在AIO socket编程中，服务端通道是AsynchronousServerSocketChannel，这个类提供了一个open()静态工厂，一个bind()方法用于绑定服务端IP地址（还有端口号），另外还提供了accept()用于接收用户连接请求。在客户端使用的通道是AsynchronousSocketChannel,这个通道处理提供open静态工厂方法外，还提供了read和write方法。

在AIO编程中，发出一个事件（accept read write等）之后要指定事件处理类（回调函数），AIO中的事件处理类是CompletionHandler<V,A>，这个接口定义了如下两个方法，分别在异步操作成功和失败时被回调。

void completed(V result, A attachment);

void failed(Throwable exc, A attachment);

2.2 AIO 异步非阻塞连接

- 服务器端：

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousServerSocketChannel;
import java.nio.channels.AsynchronousSocketChannel;
import java.nio.channels.CompletionHandler;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;

public class Server {
    public static void main(String[] args) throws IOException {
        AsynchronousServerSocketChannel serverSocketChannel =
```

```
AsynchronousServerSocketChannel.open()
                                                .bind(new
InetSocketAddress(8888));
        //异步的accept()
        serverSocketChannel.accept(null,
                                new CompletionHandler<AsynchronousSocketChannel,
Void>() {
            //有客户端连接成功的回调函数
            @Override
            public void completed(AsynchronousSocketChannel result, Void
attachment) {
                System.out.println("服务器端接收到连接...");

            }
            //IO操作失败时的回调函数
            @Override
            public void failed(Throwable exc, Void attachment) {
                System.out.println("IO操作失败！");
            }
        });
        System.out.println("服务器端继续....");
        while (true) {

        }
    }
}
```

- 客户端代码

```
import java.net.InetSocketAddress;
import java.nio.channels.AsynchronousSocketChannel;
import java.nio.channels.CompletionHandler;

public class Client {
    public static void main(String[] args) throws Exception {
        AsynchronousSocketChannel socketChannel =
AsynchronousSocketChannel.open();
        //客户端异步、非阻塞connect()方法
        socketChannel.connect(new InetSocketAddress("localhost", 8888), null,
                              new CompletionHandler<Void, Void>() {
            //连接成功时的回调函数
            @Override
            public void completed(Void result, Void attachment) {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("客户端连接成功");
            }

            @Override
            public void failed(Throwable exc, Void attachment) {
                System.out.println("客户端失败！");
            }
        });
```

```
        System.out.println("客户端继续");
        Thread.sleep(30000);
    }
}
```

- 服务器端打印：

```
服务器端继续....//非阻塞
服务器端接收到连接...//异步--回调函数被执行
```

- 客户端打印：

```
客户端继续          //非阻塞
客户端连接成功      //异步--回调函数被执行
```

2.3 AIO 同步非阻塞读写

- 服务器端代码：

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousServerSocketChannel;
import java.nio.channels.AsynchronousSocketChannel;
import java.nio.channels.CompletionHandler;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;

public class Server {
    public static void main(String[] args) throws IOException {
        AsynchronousServerSocketChannel serverSocketChannel =
AsynchronousServerSocketChannel.open()
                .bind(new InetSocketAddress(8888));
        //异步的accept()
        serverSocketChannel.accept(null,
                           new CompletionHandler<AsynchronousSocketChannel,
Void>() {
            //有客户端连接成功的回调函数
            @Override
            public void completed(AsynchronousSocketChannel result, Void
attachment) {
                System.out.println("服务器端接收到连接...");
                ByteBuffer byteBuffer = ByteBuffer.allocate(20);
                Future<Integer> readFuture = result.read(byteBuffer);//同步读
                try {
                    System.out.println("读取信息: " +
                                       new
String(byteBuffer.array(),0,readFuture.get()));
                    result.close();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } catch (ExecutionException e) {
                    e.printStackTrace();
                } catch (IOException e) {
```

```
                    e.printStackTrace();
                }
            }
            //IO操作失败时的回调函数
            @Override
            public void failed(Throwable exc, Void attachment) {
                System.out.println("IO操作失败！");
            }
        });
        System.out.println("服务器端继续....");
        while (true) {

        }
    }
}
```

- 客户端代码：

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousSocketChannel;
import java.nio.channels.CompletionHandler;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Future;

public class Client {
    public static void main(String[] args) throws Exception {
        AsynchronousSocketChannel socketChannel =
AsynchronousSocketChannel.open();
        socketChannel.connect(new InetSocketAddress("localhost", 8888), null,
                                                    new CompletionHandler<Void,
Void>() {
            @Override
            public void completed(Void result, Void attachment) {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("客户端连接成功");
                Future<Integer> writeFuture = socketChannel.write(
                                        ByteBuffer.wrap("我来自客户
端...".getBytes()));//同步写
                try {
                    System.out.println("写入大小：" + writeFuture.get());
                    socketChannel.close();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } catch (ExecutionException e) {
                    e.printStackTrace();
                } catch (IOException e) {
                    e.printStackTrace();
                }

            }
```

```
            @Override
            public void failed(Throwable exc, Void attachment) {
                System.out.println("客户端失败！");
            }
        });
        System.out.println("客户端继续");
        Thread.sleep(30000);
    }
}
```

- 服务器端执行结果：

```
服务器端继续....
服务器端接收到连接...
读取信息：我来自客户端..
```

- 客户端执行结果：

```
客户端继续
客户端连接成功
写入大小：21
```

2.4 AIO 异步非阻塞读写

- 服务器端代码：

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousServerSocketChannel;
import java.nio.channels.AsynchronousSocketChannel;
import java.nio.channels.CompletionHandler;
import java.util.concurrent.TimeUnit;

public class Server {
    public static void main(String[] args) throws IOException {
        AsynchronousServerSocketChannel serverSocketChannel =
                                        AsynchronousServerSocketChannel.open().
                                        bind(new InetSocketAddress(8888));
        serverSocketChannel.accept(null,
                                new CompletionHandler<AsynchronousSocketChannel,
Void>() {
            @Override
            public void completed(AsynchronousSocketChannel ch, Void attachment)
{
                serverSocketChannel.accept(null, this);
                ByteBuffer byteBuffer = ByteBuffer.allocate(Integer.MAX_VALUE /
300);
                System.out.println("【服务器】read开始...");
                ch.read(byteBuffer, 10, TimeUnit.SECONDS, null,
                                        new CompletionHandler<Integer, Void>() {
                    @Override
                    public void completed(Integer result, Void attachment) {
                        if (result == -1) {
```

```
                            System.out.println("客户端没有传输数据就close了...");
                        }
                        System.out.println("服务器读取数据: " +
                                                new
String(byteBuffer.array(),0,result));
                        try {
                            ch.close();
                            System.out.println("服务器关闭! ");
                        } catch (IOException e) {
                            e.printStackTrace();
                        }
                    }

                    @Override
                    public void failed(Throwable exc, Void attachment) {
                        exc.printStackTrace();
                        System.out.println(attachment);
                        System.out.println("【服务器】异常");
                    }
                });
                System.out.println("【服务器】read结束...");

            }

            @Override
            public void failed(Throwable exc, Void attachment) {

            }
        });
        System.out.println("服务器开始循环...");
        while (true) {

        }
    }
}
```

- 客户端代码:

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousSocketChannel;
import java.nio.channels.CompletionHandler;

public class Client {
    public static void main(String[] args) throws IOException,
InterruptedException {
        AsynchronousSocketChannel socketChannel =
AsynchronousSocketChannel.open();
        socketChannel.connect(new InetSocketAddress("localhost", 8888), null,
new CompletionHandler<Void, Void>() {
            @Override
            public void completed(Void result, Void attachment) {

                socketChannel.write(
                        ByteBuffer.wrap("你好服务器".getBytes()), null,
```

```
                        new CompletionHandler<Integer, Void>() {
                            @Override
                            public void completed(Integer result, Void
attachment) {
                                System.out.println("输出完毕！");
                            }

                            @Override
                            public void failed(Throwable exc, Void attachment) {
                                System.out.println("输出失败！");
                            }
                        });

                try {
                    socketChannel.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }

            @Override
            public void failed(Throwable exc, Void attachment) {
                System.out.println("【客户端】异常！");
            }
        });
        Thread.sleep(1000);
    }
}
```

- 服务器端结果：

```
服务器开始循环...
【服务器】read开始...
【服务器】read结束...
服务器读取数据：你好服务器
服务器关闭!
```

- 客户端结果：

```
输出完毕!
```