【第13话：被虐了，面试官通过几个IO模型给我虐的体无完肤】

Hello 小伙伴们，这节课给大家讲解下，面试官问我们“AIO、BIO、NIO是什么以及他们的区别是什么”时，我们应该如何回答。

我们先来总体看看这三个IO模型。

BIO：英文全称Blocking Input\Output 。是一种同步阻塞IO模型。

NIO：英文全称New Input\Output ，又称Non-Blocking Input\Output ，是一种同步非阻塞IO模型。

AIO：英文全称 Asynchronous Input\Output 。是一种异步非阻塞IO模型。

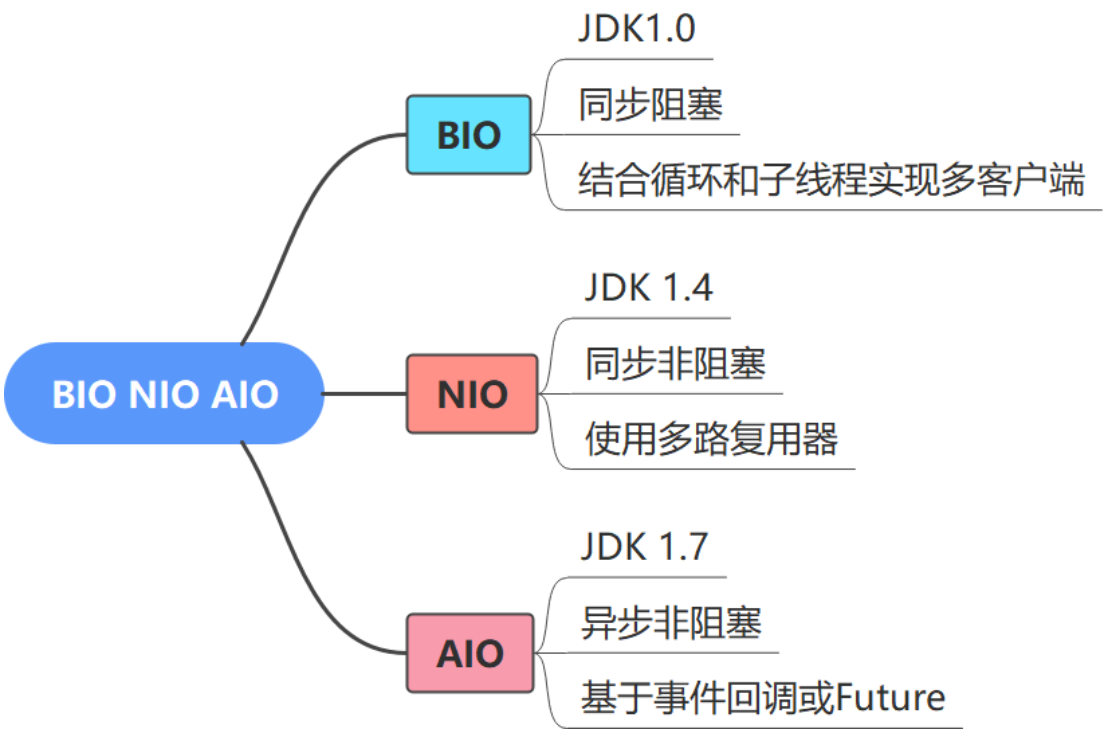
看到这里，小伙伴们可能对里面的两组名词有一些疑惑。同步和异步有什么区别？阻塞和非阻塞有什么区别？

我们分别来说一说。

同步：对于**一个线程内**，下面的代码是否等待上面的代码执行完成，并返回结果之后在执行。如果后面代码等待上面代码执行完成在执行，称为**同步**。如果后面代码不等待上面代码执行完成，就继续执行，称为**异步**。

阻塞：IO模型中阻塞指是否阻塞**多线程**的访问。具体解释：如果一个客户端连接上了服务端，其他客户端就不能连接上，这时就是阻塞IO。反之，一个线程连接上，其他线程也能连接就是非阻塞IO。

BIO、NIO、AIO的区别



下面来详细的说一说这三个IO模型。先来说说BIO。

BIO (Blocking I/O) 同步阻塞IO是 JDK1.4之前唯一选择, 只提供这一种IO模型。

线程发起 IO 请求, 从发起请求起, 线程一直阻塞，直到读写操作完成。

服务端一个线程只能同时处理一个客户端的请求。

因为BIO是同步阻塞IO。所以每个客户端连接后都需要创建一个线程，专门负责与客户端的操作。

```
public class MyServer {
  public static void main(String[] args) {
    try {
      //1.创建ServerSocket, 自动监听指定的ip+port
      ServerSocket serverSocket = new ServerSocket(9999);
      System.out.println("等待客户端连接");
      while (true){
        //2.阻塞等待客户端的连接
        Socket accept = serverSocket.accept();
        System.out.println("接受到了客户端: "+accept);
        //创建线程
        new Thread(()->{
          try {
            //3.等待接收客户端的消息
            InputStream inputStream = accept.getInputStream();
            DataInputStream dataInputStream = new
DataInputStream(inputStream);
            System.out.println("接收到客户端消息: " +
dataInputStream.readUTF());
          } catch (IOException e) {
            e.printStackTrace();
          }
        }).start();
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

客户端1的代码，只是连接了服务端，并向服务端发送了数据。

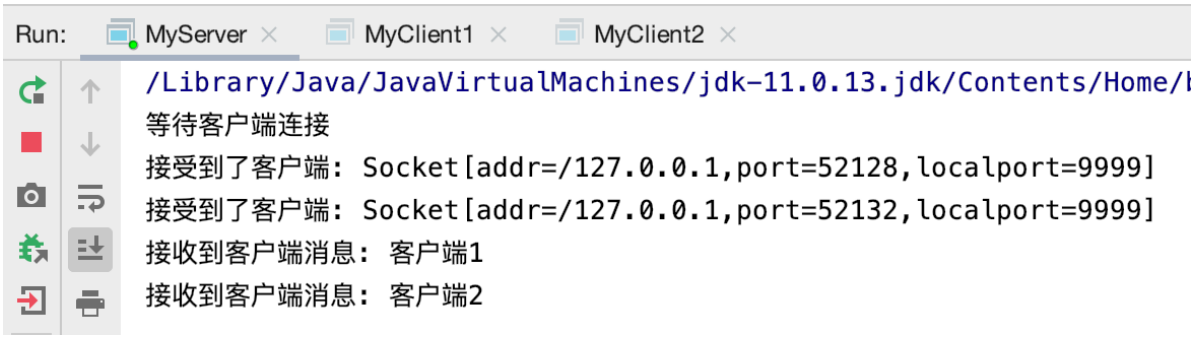
```
public class MyClient1 {
  public static void main(String[] args) {
    try {
      //1.创建客户端Socket, 自动连接指定的ip+port
      Socket socket = new Socket("127.0.0.1", 9999);
      Scanner scanner = new Scanner(System.in);
      //2.向服务端发送消息
      DataOutputStream dataOutputStream = new
DataOutputStream(socket.getOutputStream());
      dataOutputStream.writeUTF(scanner.next());
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
```

```
14  }
```

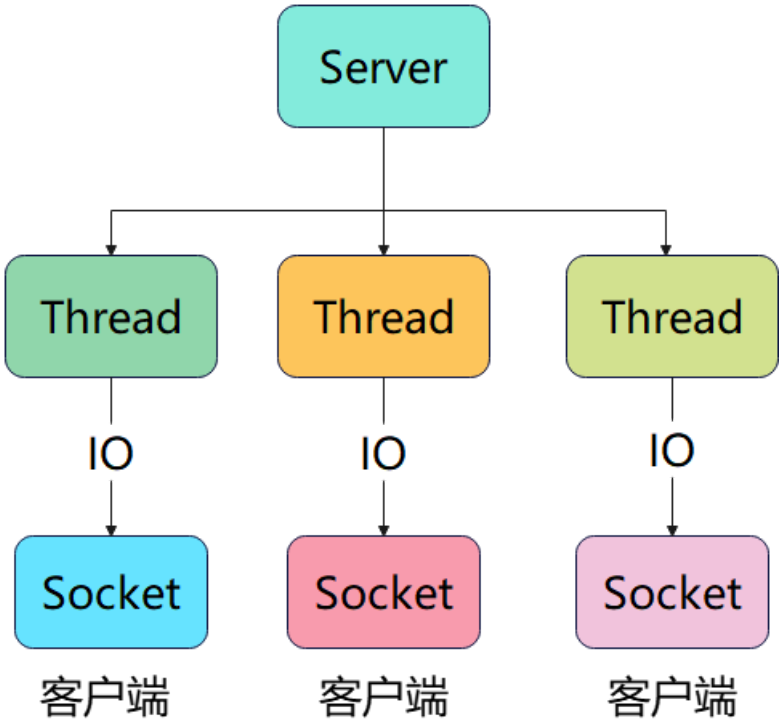
客户端2的代码，也是连接了服务端，并向服务端发送了数据。

```
1  public class MyClient2 {
2    public static void main(String[] args) {
3      try {
4        //1.创建客户端Socket, 自动连接指定的ip+port
5        Socket socket = new Socket("127.0.0.1", 9999);
6        Scanner scanner = new Scanner(System.in);
7        //2.向服务端发送消息
8        DataOutputStream dataOutputStream = new
   DataOutputStream(socket.getOutputStream());
9        dataOutputStream.writeUTF(scanner.next());
10     } catch (Exception e) {
11       e.printStackTrace();
12     }
13   }
14 }
```

最终在IDEA中的效果是



BIO(同步阻塞IO)



每个客户端需要对应服务端的一个子线程。 所以有多少个客户端, 服务端就需要创建多少个子线程。如果客户端特别多，几万甚至几百万，服务器端就需要有几万甚至几百万的子线程。 由于每个子线程都有自己的独立线程区，这么多子线程可能就产生内存不足等问题。而且这么多的子线程也需要进行调度, 切换, 销毁这也是非常消耗性能的

虽然可以解决一个服务端处理多个客户端，但是因为过多的子线程导致系统资源消耗过多，线程切换导致性能下降也是我们不得不需要考虑的问题

这时可以使用线程池，在**一定程度**上提升程序性能。

NIO

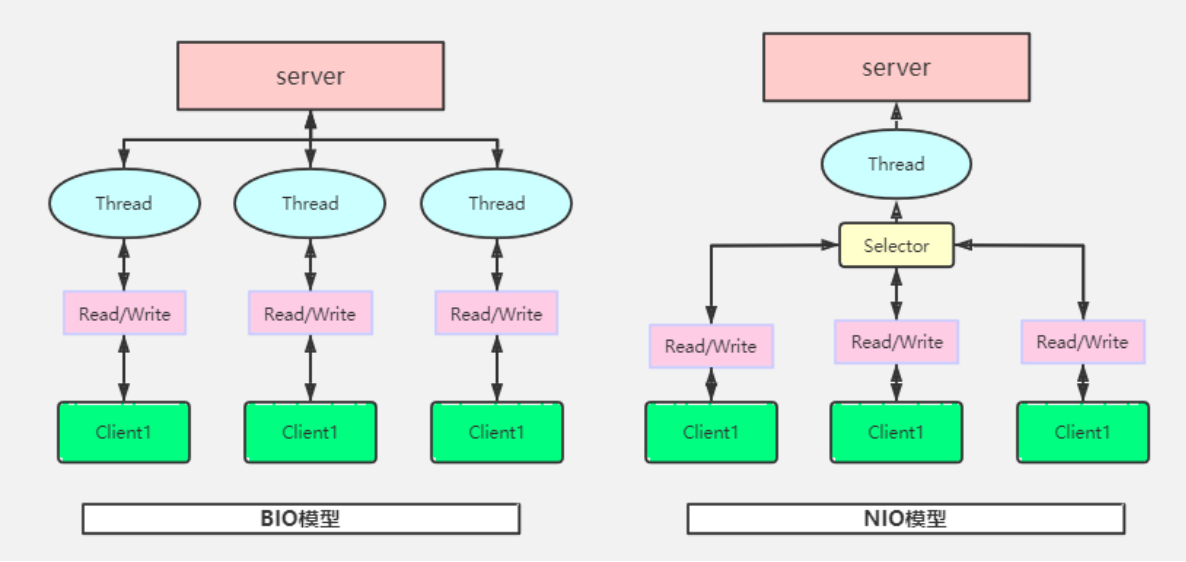
BIO演示完成，下面来说一说NIO

NIO称为New IO(新的IO)又称Non-Blocking IO(非阻塞IO), 同步非阻塞IO模型

Java 从JDK1.4版本开始推出了NIO模型, 在java.nio包中。

NIO的服务端不需要有那么多个线程，只需要一个线程就行。多了一个Selector负责循环查看通道状态。到通道状态为可读状态时从缓存区中读取内容。如果通道为ACCEPT客户端要获取连接。Selector监听所有通道。所以一个线程可以完成之前多线程的事情。

NIO（同步非阻塞IO）



代码实现如下。

1. 服务端

```
public class Server {
  public static void main(String[] args) throws IOException {
    Scanner scanner = new Scanner(System.in);
    //1.获取服务端管道
    ServerSocketChannel ssc = ServerSocketChannel.open();
    ssc.configureBlocking(false); //默认阻塞，设置为非阻塞
    //2.绑定 与 监听指定的 ip和port
    ssc.bind(new InetSocketAddress("127.0.0.1", 6666));
    //3.将服务端管道注册到选择器(多路复用器中)
    //3.1获取多路复用器
    Selector selector = Selector.open();
    //3.2将服务端管道注册到多路复用器中
    /*
        * 监听管道的状态(监听接收到了客户端的状态)
        *   客户端连接成功，服务端就会监听到，接收客户端的状态
        * */
    ssc.register(selector, SelectionKey.OP_ACCEPT);
    while (true){
      //4.选择多路复用器监听到的状态
      /*
       * 没有状态，阻塞
       * 有状态，继续执行
       * */
      selector.select();
      //5.获取所有被监听到的状态
      Set<SelectionKey> selectionKeys = selector.selectedKeys();
      Iterator<SelectionKey> iterator = selectionKeys.iterator(); //获取集合
的迭代器
      while (iterator.hasNext()) {
        SelectionKey next = iterator.next();
```

```
        if (next.isAcceptable()){ //监听到接收到了客户端的状态
          System.out.println("-----------");
          //接收客户端
          SocketChannel sc = ssc.accept();
          sc.configureBlocking(false);
          sc.register(selector, SelectionKey.OP_READ);
        }else if(next.isReadable()){
          //通过多路复用器获取连接的客户端 管道
          SocketChannel sc = (SocketChannel) next.channel();
          ByteBuffer bbf = ByteBuffer.allocate(1024);
          sc.read(bbf);
          byte[] array = bbf.array();
          String s = new String(array, 0, bbf.position());
          System.out.println(s);
          //监听写的状态
          sc.register(selector, SelectionKey.OP_WRITE);
        }else if(next.isWritable()){
          //通过多路复用器获取连接的客户端 管道
          SocketChannel sc = (SocketChannel) next.channel();
          //向客户端发送消息
          //ByteBuffer bb = ByteBuffer.wrap(scanner.next().getBytes());
          ByteBuffer bb = ByteBuffer.wrap("server".getBytes());
          sc.write(bb);
          //监听读的状态
          sc.register(selector, SelectionKey.OP_READ);
        }
        //为了防止重复的执行, 执行后将该被监听到的状态 移除
        iterator.remove();
      }
    }
  }
}
```

2. 客户端

```
public class Client {
  public static void main(String[] args) throws IOException {
    Scanner scanner = new Scanner(System.in);
    //1.获取客户端管道
    SocketChannel sc = SocketChannel.open();
    sc.configureBlocking(false); //设置为非阻塞
    //2.连接服务端
    sc.connect(new InetSocketAddress("127.0.0.1", 6666));
    //3.将客户端管道注册到多路复用器
    Selector selector = Selector.open();
    /*
     * 监听连接了服务端的状态
     * */
    sc.register(selector, SelectionKey.OP_CONNECT);
    while (true){
      //4.选择监听到的状态
      selector.select();
      //5.获取所有被监听到的状态
      Set<SelectionKey> selectionKeys = selector.selectedKeys();
      Iterator<SelectionKey> iterator = selectionKeys.iterator();
```

```
      while (iterator.hasNext()) { //被监听的状态可以获得到，执行循环
        SelectionKey next = iterator.next();
        if (next.isConnectable()) { //连接服务端状态被监听到
          if (sc.isConnectionPending()){//没有完全连接
            sc.finishConnect();//完成连接
          }
          //向服务端发送消息
          ByteBuffer bb = ByteBuffer.wrap("连接成功".getBytes());
          sc.write(bb);
          sc.register(selector, SelectionKey.OP_READ);
        }else if(next.isReadable()){
          //接收服务端的消息
          ByteBuffer bbf = ByteBuffer.allocate(1024);
          sc.read(bbf);
          byte[] array = bbf.array();
          String s = new String(array, 0, bbf.position());
          System.out.println(s);
          sc.register(selector, SelectionKey.OP_WRITE);
        }else if(next.isWritable()){
          //ByteBuffer bb = ByteBuffer.wrap(scanner.next().getBytes());
          ByteBuffer bb = ByteBuffer.wrap("client".getBytes());
          sc.write(bb);
          sc.register(selector, SelectionKey.OP_READ);
        }
        //为了防止重复的执行，执行后将该被监听到的状态 移除
        iterator.remove();
      }
    }
  }
}
```

AIO(异步非阻塞IO)

AIO（asynchronous）：异步非阻塞IO。

在执行时，当前线程内不会阻塞。而且也不需要多线程就可以实现多客户端访问。

AIO是从Java 7 开始出现的。

AIO基于事件模型完成的或Future完成的。

1. 服务端

```
public static void main(String[] args) {
  CountDownLatch countDownLatch = new CountDownLatch(1);
  try {
    AsynchronousServerSocketChannel serverSocketChannel =
AsynchronousServerSocketChannel.open();
    serverSocketChannel.bind(new InetSocketAddress("127.0.0.1",9999));
    serverSocketChannel.accept(null, new
CompletionHandler<AsynchronousSocketChannel, Object>() {
      @Override
      public void completed(AsynchronousSocketChannel
asynchronousSocketChannel, Object attachment) {
        ByteBuffer allocate = ByteBuffer.allocate(1024);
        asynchronousSocketChannel.read(allocate, null, new
CompletionHandler<Integer, Object>() {
          @Override
          public void completed(Integer result, Object attachment) {
            byte[] array = allocate.array();
            String s = new String(array);
            System.out.println(s);
            countDownLatch.countDown();
          }

          @Override
          public void failed(Throwable exc, Object attachment) {
            System.out.println("接收失败, 失败原因"+exc.getMessage());
          }
        });
      }
      @Override
      public void failed(Throwable exc, Object attachment) {
        System.out.println("失败, 失败原因"+exc.getMessage());
      }
    });
    countDownLatch.await();
  } catch (IOException | InterruptedException e) {
    e.printStackTrace();
  }
}
```

2. 客户端

```
public static void main(String[] args) {
  CountDownLatch countDownLatch = new CountDownLatch(1);
  try {
    AsynchronousSocketChannel socketChannel =
AsynchronousSocketChannel.open();
    socketChannel.connect(new InetSocketAddress("127.0.0.1", 9999), null,
new CompletionHandler<Void, Object>() {
      @Override
      public void completed(Void result, Object attachment) {
        socketChannel.write(ByteBuffer.wrap("hello".getBytes()));
```

```
        countDownLatch.countDown();
      }

      @Override
      public void failed(Throwable exc, Object attachment) {
        System.out.println(exc);
      }
    });
    countDownLatch.await();
  } catch (IOException | InterruptedException e) {
    e.printStackTrace();
  }
}
```

BIO、NIO、AIO：

Java BIO： 同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。

Java NIO： 同步非阻塞，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。

Java AIO(NIO.2)： 异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理。