7-6 | 三种 IO 代码

BIO

关于 BIO 的服务端代码如下：

```Java
package bio;

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

/**
 * @Author idea
 * @Date: Created in 20:50 2023/7/1
 * @Description
 */
public class BioServer {

    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = new ServerSocket();
        serverSocket.bind(new InetSocketAddress(9090));
        Socket socket = serverSocket.accept();
        while (true) {
            InputStream inputStream = socket.getInputStream();
            byte[] bytes = new byte[20];
            inputStream.read(bytes);
            System.out.println("读取到的数据是：" + new
String(bytes));
        }
    }
}
```

关于 BIO 的客户端代码如下：

```Java
package bio;
```

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

/**
 * @Author idea
 * @Date: Created in 20:52 2023/7/1
 * @Description
 */
public class BioClient {

    public static void main(String[] args) throws IOException,
InterruptedException {
        Socket socket = new Socket();
        socket.connect(new InetSocketAddress(9090));
        OutputStream outputStream = socket.getOutputStream();
        while (true) {
            outputStream.write("test".getBytes());
            outputStream.flush();
            System.out.println("发送数据");
            Thread.sleep(1000);
        }
    }
}
```

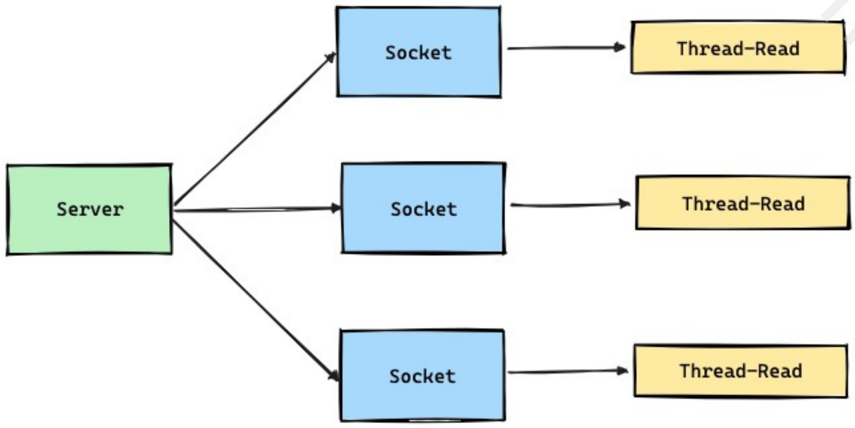
如果希望在 BIO 的服务端使用异步的思路去进行优化，那么可以参考如下版本的代码去进行实践：

```java
package bio;


import java.io.IOException;
import java.io.InputStream;
import java.net.InetSocketAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
```

```
/**
 * @Author idea
 * @Date: Created in 20:50 2023/7/1
 * @Description
 */
public class BioServer2 {

    private static ThreadPoolExecutor threadPoolExecutor = new
ThreadPoolExecutor(10, 10, 3, TimeUnit.MINUTES, new
ArrayBlockingQueue<>(100));

    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = new ServerSocket();
        //绑定端口9090
        serverSocket.bind(new InetSocketAddress(9090));
        while (true) {
            try {
                Socket socket = serverSocket.accept();
                threadPoolExecutor.execute(() -> {
                    try {
                        InputStream inputStream =
socket.getInputStream();
                        byte[] bytes = new byte[10];
                        //阻塞调用
                        inputStream.read(bytes);
                        System.out.println("服务端收到的数据是：" +
new String(bytes));
                    } catch (IOException e) {
                        throw new RuntimeException(e);
                    }
                });
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

但是如果按照上边的这段代码去实现的话，会存在一定的性能问题。

每次来一个请求，就创建一个连接，假设我们极端情况下，一台服务器下维持了 1000 条连接，但是这一千条连接都是没有数据发送的状态，那么我们的服务端就必须要有 1000 条线程去进行维持，并且都是处于 read 的阻塞状态。这不就是白白的资源浪费么？



这种模型的并发度并不会有很好的一个表现，因为它的并发度取决于后台可以创建的线程数。

那么下边，让我们再来看看 NIO 的实现思路是怎样的。

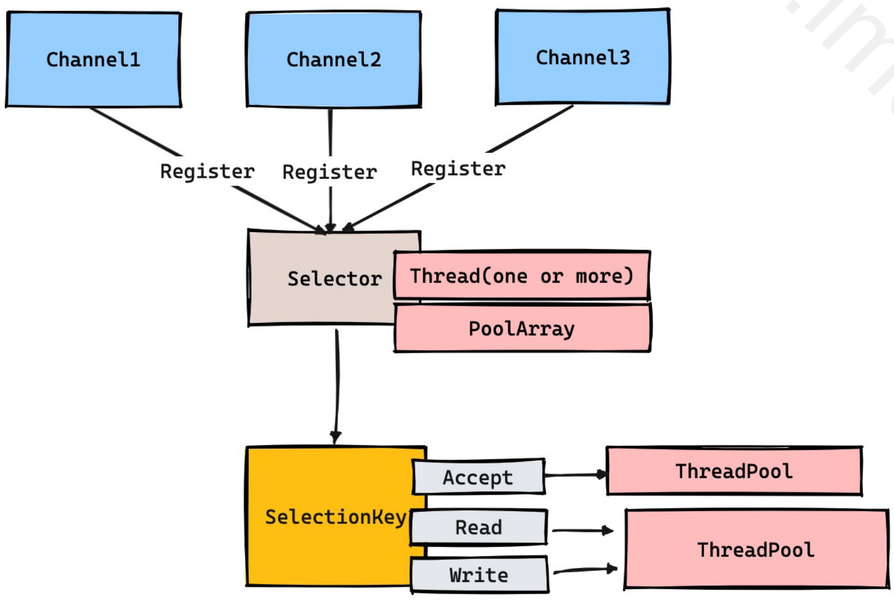
NIO

简单的 NIO 服务端代码案例如下，

```
Java
package nio;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.ArrayList;
import java.util.List;

/**
 * @Author idea
 * @Date: Created in 21:59 2023/7/1
 * @Description
 */
public class NioSimpleServer {

    private static List<SocketChannel> acceptSocketList = new
ArrayList<>();
```

```
    public static void main(String[] args) throws IOException {
        ServerSocketChannel serverSocketChannel =
ServerSocketChannel.open();
        serverSocketChannel.socket().bind(new
InetSocketAddress(9090));
        serverSocketChannel.configureBlocking(false);
        System.out.println("服务启动成功");
        new Thread(() -> {
            while (true) {
                for (SocketChannel socketChannel :
acceptSocketList) {
                    try {
                        ByteBuffer byteBuffer =
ByteBuffer.allocate(10);
                        socketChannel.read(byteBuffer);
                        System.out.println("收到数据:" + new
String(byteBuffer.array()));
                    } catch (IOException e) {
                        throw new RuntimeException(e);
                    }
                }
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
        }).start();
        while (true) {
            SocketChannel socketChannel =
serverSocketChannel.accept();
            if (socketChannel != null) {
                System.out.println("连接成功了");
                socketChannel.configureBlocking(false);
                acceptSocketList.add(socketChannel);
            }
        }
    }
}
```

基于 Selector 去实现的 NIO 代码，底层执行链路如下所示：



基于 Selector 实现 NIO 服务端的代码实现如下：

```Java
package nio;

import java.io.IOException;
 import java.net.InetSocketAddress;
 import java.net.ServerSocket;
 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 NIOSelectorServer {

     /*标识数字*/
     private  int flag = 0;
     /*缓冲区大小*/
     private  int BLOCK = 4096;
     /*接受数据缓冲区*/
     private  ByteBuffer sendbuffer = ByteBuffer.allocate(BLOCK);

     /*发送数据缓冲区*/
```

```
    private  ByteBuffer receivebuffer =
ByteBuffer.allocate(BLOCK);
    private  Selector selector;

    public NIOSelectorServer(int port) throws IOException {
        // 打开服务器套接字通道
        ServerSocketChannel serverSocketChannel =
ServerSocketChannel.open();
        // 服务器配置为非阻塞
        serverSocketChannel.configureBlocking(false);
        // 检索与此通道关联的服务器套接字
        ServerSocket serverSocket = serverSocketChannel.socket();

        // 进行服务的绑定
        serverSocket.bind(new InetSocketAddress(port));
        // 通过 open()方法找到 Selector
        selector = Selector.open();
        System.out.println(selector);
        // 注册到 selector，等待连接
        serverSocketChannel.register(selector,
SelectionKey.OP_ACCEPT);
        System.out.println("Server Start----8888:");
    }


    // 监听
    private void listen() throws IOException {
        while (true) {
            // 这里如果没有 IO 事件抵达 就会进入阻塞状态
            selector.select();
            System.out.println("select");
            // 返回此选择器的已选择键集。
            Set<SelectionKey> selectionKeys =
selector.selectedKeys();
            Iterator<SelectionKey> iterator =
selectionKeys.iterator();
            while (iterator.hasNext()) {
                SelectionKey selectionKey = iterator.next();
                iterator.remove();
                handleKey(selectionKey);
            }
        }
    }
```

```
     // 处理理请求
     private void handleKey(SelectionKey selectionKey) throws
IOException {
         // 接受请求
         ServerSocketChannel server = null;
         SocketChannel client = null;
         String receiveText;
         String sendText;
         int count=0;
         // 测试此键的通道是否已准备好接受新的套接字连接。
         if (selectionKey.isAcceptable()) {
             // 返回为之创建此键的通道。
             server = (ServerSocketChannel)
selectionKey.channel();
             // 接受到此通道套接字的连接。
             // 非阻塞模式这里不会阻塞
             client = server.accept();
             // 配置为非阻塞
             client.configureBlocking(false);
             // 注册到 selector，等待连接
             client.register(selector, SelectionKey.OP_READ);
         } else if (selectionKey.isReadable()) {
             // 返回为之创建此键的通道。
             client = (SocketChannel) selectionKey.channel();
             //将缓冲区清空以备下次读取
             receivebuffer.clear();
             //读取服务器发送来的数据到缓冲区中
             count = client.read(receivebuffer);
             if (count > 0) {
                 receiveText = new
String( receivebuffer.array(),0,count);
                 System.out.println("服务器端接受客户端数
据--:"+receiveText);
                 client.register(selector, SelectionKey.OP_WRITE);

             }
         } else if (selectionKey.isWritable()) {
             //将缓冲区清空以备下次写入
             sendbuffer.clear();
             // 返回为之创建此键的通道。
             client = (SocketChannel) selectionKey.channel();
             sendText="message from server--" + flag++;
             //向缓冲区中输入数据
             sendbuffer.put(sendText.getBytes());
```

```
                //将缓冲区各标志复位,因为向里面 put 了数据标志被改变要想从
中读取数据发向服务器,就要复位
              sendbuffer.flip();
              //输出到通道
              client.write(sendbuffer);
              System.out.println("服务器端向客户端发送数
据--：  "+sendText);
              client.register(selector, SelectionKey.OP_READ);
          }
      }

      /**
       * @param args
       * @throws IOException
       */
      public static void main(String[] args) throws IOException {
          int port = 9090;
          NIOSelectorServer server = new NIOSelectorServer(port);
          server.listen();
      }
 }
```

在 LInux 环境中，java.nio.channels.Selector 的子类叫做 sun.nio.ch.EPollSelectorImpl ，其底层是基于 Epoll 模型去实现的 IO 多路复用器。

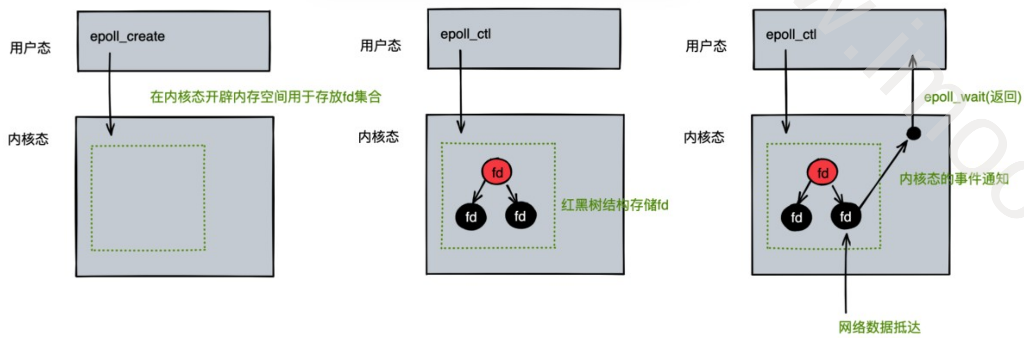
```
[root@VM-12-17-centos tmp]# java nio.NIOSelectorServer
sun.nio.ch.EPollSelectorImpl@33909752
Server Start----8888:
^C[root@VM-12-17-centos tmp]#
```

对于 Epoll 模型 我们需要了解到它底层的三个函数

```
static native int create() throws IOException;

static native int ctl(int epfd, int opcode, int fd, int events);

static native int wait(int epfd, long pollAddress, int numfds, int timeout)
    throws IOException;
```

在 JDK 实现的底层中，EPollSelectorImpl 在初次创建的时候，会调用 create 函数去内存块中开辟一块空间。然后再调用 ctl 方法，往这个内存块中创建一颗红黑树，并且将 socket 对象插入到树上。然后再调用 wait 方法，让出 CPU。

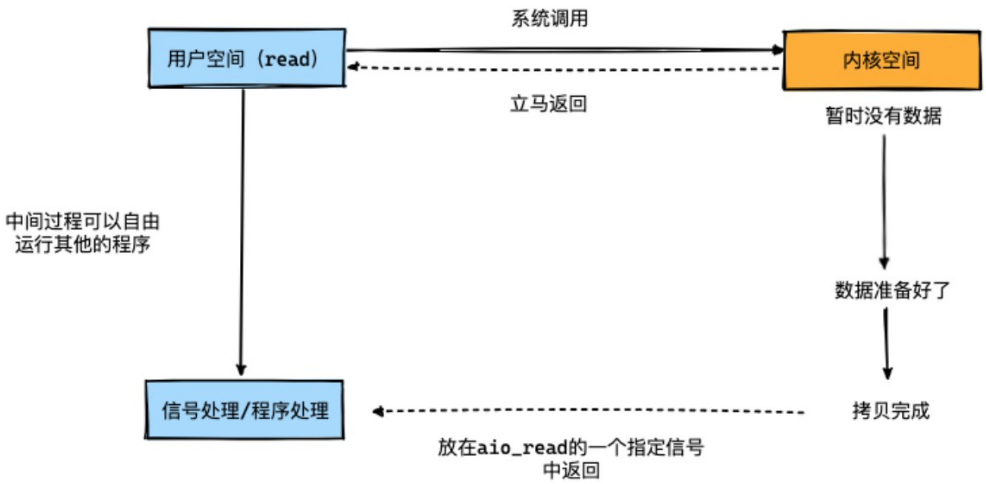
整体的执行过程如下图所示：



等待到有数据抵达的时候，这个 wait 的阻塞方法，才会继续执行下去。

AIO

AIO 代码的流程如下图所示：



在代码实现上，可以基于一个回调通知的形式来进行开发，其服务端代码如下：

```Java
package aio;

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.nio.charset.Charset;
import java.util.concurrent.ExecutionException;
```

```
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;


public class AIOServer {

    public final static int PORT = 9888;
    private AsynchronousServerSocketChannel server;

    public AIOServer() throws IOException {
        server = AsynchronousServerSocketChannel.open().bind(
                new InetSocketAddress(PORT));
    }

    /**
     * 不推荐使用 future 的方式去进行编程，这种方式去实现 AIO 其实本质和
BIO 没有太大的区别
     *
     * @throws InterruptedException
     * @throws ExecutionException
     * @throws TimeoutException
     */
    public void startWithFuture() throws InterruptedException,
            ExecutionException, TimeoutException {
        while (true) {// 循环接收客户端请求
            Future<AsynchronousSocketChannel> future =
server.accept();
            AsynchronousSocketChannel socket = future.get();//
get() 是为了确保 accept 到一个连接
            handleWithFuture(socket);
        }
    }

    public void handleWithFuture(AsynchronousSocketChannel
channel) throws InterruptedException, ExecutionException,
TimeoutException {
        ByteBuffer readBuf = ByteBuffer.allocate(2);
        readBuf.clear();

        while (true) {// 一次可能读不完
            //get 是为了确保 read 完成，超时时间可以有效避免 DOS 攻击，
如果客户端一直不发送数据，则进行超时处理
            Integer integer = channel.read(readBuf).get(10,
```

```
TimeUnit.SECONDS);
            System.out.println("read: " + integer);
            if (integer == -1) {
                break;
            }
            readBuf.flip();
            System.out.println("received: " +
Charset.forName("UTF-8").decode(readBuf));
            readBuf.clear();
        }
    }

    /**
     * 即提交一个 I/O 操作请求，并且指定一个 CompletionHandler。
     * 当异步 I/O 操作完成时，便发送一个通知，此时这个
CompletionHandler 对象的 completed 或者 failed 方法将会被调用。
     *
     * @throws InterruptedException
     * @throws ExecutionException
     * @throws TimeoutException
     */
    public void startWithCompletionHandler() throws
InterruptedException,
            ExecutionException, TimeoutException {
        server.accept(null,
                new CompletionHandler<AsynchronousSocketChannel,
Object>() {
                    public void
completed(AsynchronousSocketChannel result, Object attachment) {
                        server.accept(null, this);// 再此接收客户端
连接
                        handleWithCompletionHandler(result);
                    }

                    @Override
                    public void failed(Throwable exc, Object
attachment) {
                        exc.printStackTrace();
                    }
                });
    }

    public void handleWithCompletionHandler(final
AsynchronousSocketChannel channel) {
```

```
        try {
            final ByteBuffer buffer = ByteBuffer.allocate(4);
            final long timeout = 10L;
            channel.read(buffer, timeout, TimeUnit.SECONDS, null,
new CompletionHandler<Integer, Object>() {
                @Override
                public void completed(Integer result, Object
attachment) {
                    System.out.println("read:" + result);
                    if (result == -1) {
                        try {
                            channel.close();
                        } catch (IOException e) {
                            e.printStackTrace();
                        }
                        return;
                    }
                    buffer.flip();
                    System.out.println("received message:" +
Charset.forName("UTF-8").decode(buffer));
                    buffer.clear();
                    channel.read(buffer, timeout,
TimeUnit.SECONDS, null, this);
                }

                @Override
                public void failed(Throwable exc, Object
attachment) {
                    exc.printStackTrace();
                }
            });
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String args[]) throws Exception {
//        new AIOServer().startWithFuture();
        new AIOServer().startWithCompletionHandler();
        Thread.sleep(100000);
    }
}
```

客户端代码如下：

```Java
package aio;

import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousSocketChannel;

public class AIOClient {

    public static void main(String... args) throws Exception {
        AsynchronousSocketChannel client =
AsynchronousSocketChannel.open();
        client.connect(new InetSocketAddress("localhost",
9888)).get();
        while (true) {
            client.write(ByteBuffer.wrap("123456789".getBytes()));
            Thread.sleep(1000);
        }
    }
}
```

为什么 Netty 没有使用 AIO 而是采用 NIO 的思路去进行设计？

引用了创始人的一段话来解释下这个原因：

> According to the book the main reasons were:
>
> - Not faster than NIO (epoll) on unix systems (which is true)
> - There is no daragram suppport
> - Unnecessary threading model (too much abstraction without usage)
>
> I agree that AIO will not easily replace NIO, but it is useful for windows developers nonetheless.

1. 不比 nio 快在 Unix 系统上
1. 不支持数据报
1. 不必要的线程模型（太多没什么用的抽象化）

总而言之，可以理解为，在 Unix 系统上 AIO 性能综合表现不如 NIO 好，所以 Netty 使用了 NIO 作为底层的核心。