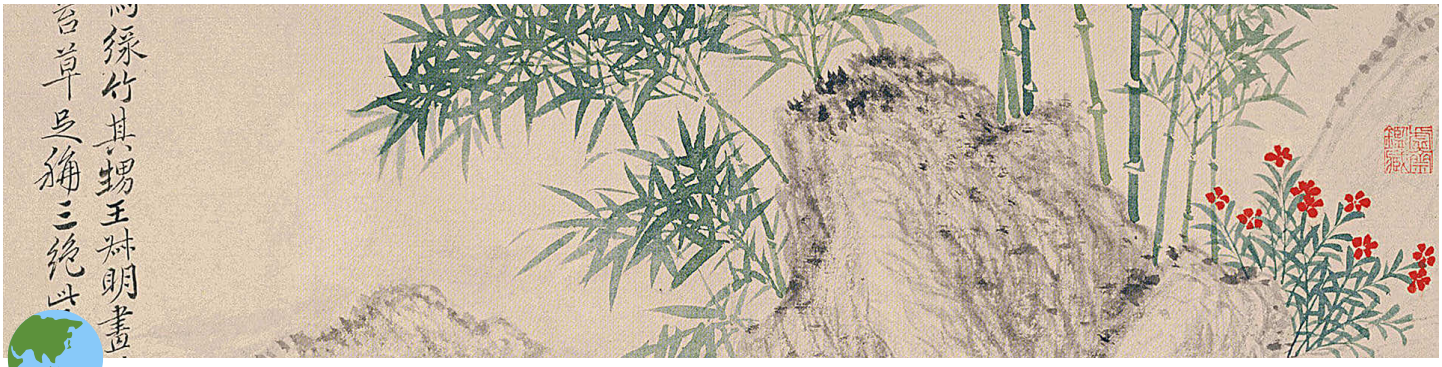


NIO服务器

标签 空

新增属性

源码地址：https://gitee.com/panda_99/my-tomcat.git

源码实现的简单web服务器，涉及了NIO、线程池、HTTP请求等知识点。

基本概念

NIO

Java.nio全称java non-blocking IO，是指jdk1.4及以上版本里提供的新api（New IO），为所有的原始类型（boolean类型除外）提供缓存支持的数据容器，使用它可以提供非阻塞式的高伸缩性网络。

Channel是一个对象，可以通过它读取和写入数据。拿NIO与原来的I/O做个比较，通道就像是流，而且他们面向缓冲区的。

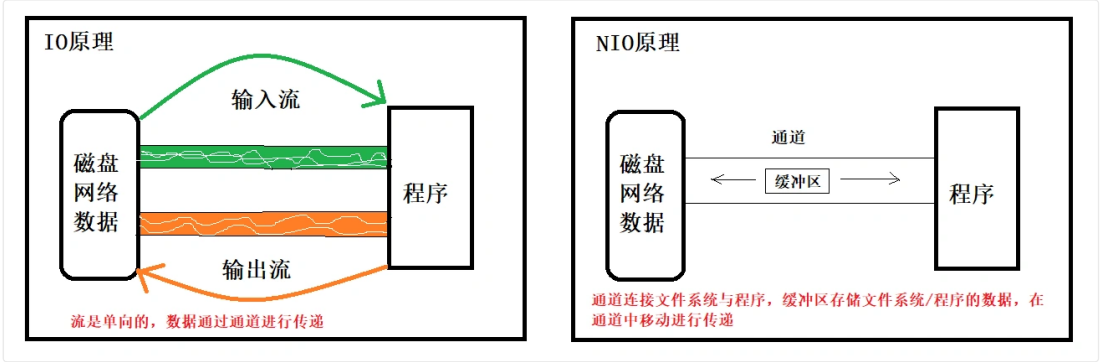
2.什么是缓冲区（Buffer）？

Buffer是一个固定数据量、指定基本类型的数据容器。具有位置（要读写的下一个元素的索引）和界限（第一个应该读写的元素的索引）。每个非布尔基本类型都有一个缓冲区类，可以通过get和put将数据移除或移入。

NIO与传统IO区别

类别	数据传输途径	传输途径的特点
I/O	输入流InputStream、输出流OutputStream	流单向，输入读，输出写
NIO	通道Channel	通道双向，可：读、写、读写。非阻塞

程序与磁盘、网络之间数据的传输如下图。

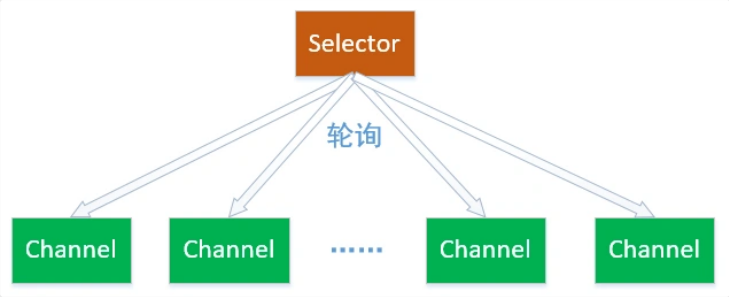


通道 channel

Channel是一个对象，可以通过它读取和写入数据。作用相当于传统I/O中的流。获取通道中的数据时，先将数据读取到buffer（缓冲区），然后从缓冲区获取数据字节。

选择器 Selector

Selector也叫多路复用器，通过轮询的方式去检测一个/多个通道是否可读、可写。从而实现单线程管理多个通道，也就是可以管理多个网络链接。



优点：使用更少的线程来就可以来处理通道了， 相比使用多个线程，避免了线程上下文切换带来的开销。

使用示例：

```java
Selector selector = Selector.open();                    //创建选择器
ServerSocketChannel channel = ServerSocketChannel.open();          //创建通道
channel.configureBlocking(false);                       //设置通道为非阻塞模式
SelectionKey key = channel.register(selector, Selectionkey.OP_READ); //注册通道到选择器
```

> FileChannel无法修改为非阻塞模式，因此无法注册到选择器。
> 注册通道时，会指定选择器要监听的通道的事件类型：

SelectionKey类型标识符	实际值	用途
SelectionKey.OP_CONNECT	1 << 3	通道已准备好连接
SelectionKey.OP_ACCEPT	1 << 4	通道准备接收
SelectionKey.OP_READ	1 << 0	通道已准备好读取
SelectionKey.OP_WRITE	1 << 2	通道已准备好写入

SelectionKey 表示选择器与通道的注册关系，常见的方法如下：

SelectionKey方法	用途
interestOps()	获取监听的事件类型列表
interestOps(SelectionKey.OP_READ)	修改键的事件类型
channel()	获取通道
selector()	获取选择器
readyOps()	通道已经准备就绪的 IO 操作的集合
isAcceptable()	是否可读
isWritable()	是否可写
isConnectable()	是否可连接
isAcceptable()	是否可接收
attach(Object obj)	将一个对象添加到SelectionKey中
attachment()	获取 attach( ) 添加的对象

> 对象或者更多信息附着到SelectionKey上，这样就能方便的识别某个给定的通道。
> 将通道注册到选择器时，也可以将对象添加到 SelectionKey

```java
SelectionKey key = channel.register(selector, SelectionKey.OP_READ, Object);
```

Selector 获取键

Selector 方法	用途
keys()	获取所有与选择器关联的通道所生成的键
select()	获取已经就绪的通道数，在无就绪通道时阻塞，可以设置阻塞时长
selectNow()	获取通道数，非阻塞，只要有通道就绪就立刻返回
selectedKeys()	获取选择器已选择的全部键

选择器执行选择示例：

```java
Set selectedKeys = selector.selectedKeys();
Iterator keyIterator = selectedKeys.iterator();
while(keyIterator.hasNext()) {
    SelectionKey key = keyIterator.next();
    if(key.isAcceptable()) {
        // ServerSocketChannel接受了连接.后的操作
    } else if (key.isConnectable()) {
        // 与远程服务器建立连接.后的操作
    } else if (key.isReadable()) {
        // 通道已准备好读.后的操作
    } else if (key.isWritable()) {
        // 通道已准备好写.后的操作
    }
    keyIterator.remove();  //选择被处理过，直接删除选项
}
```

执行选择后的操作时，可能会发生阻塞，唤醒的方法有 2 种：

1. wakeup()：首个阻塞的线程立刻返回。
2. close()：关闭选择器，并注销所有通道（不关闭），从而唤醒所有阻塞线程。

数据传输

1.server 端

```java
public class WebServer {
    public static void main(String[] args) {
        try {
            ServerSocketChannel ssc = ServerSocketChannel.open();
            ssc.socket().bind(new InetSocketAddress("127.0.0.1", 8000));
            ssc.configureBlocking(false);

            Selector selector = Selector.open();
            // 注册 channel，并且指定感兴趣的事件是 Accept
            ssc.register(selector, SelectionKey.OP_ACCEPT);

            ByteBuffer readBuff = ByteBuffer.allocate(1024);
            ByteBuffer writeBuff = ByteBuffer.allocate(128);
            writeBuff.put("received".getBytes());
            writeBuff.flip();

            while (true) {
                int nReady = selector.select();
                Set<SelectionKey> keys = selector.selectedKeys();
                Iterator<SelectionKey> it = keys.iterator();

                while (it.hasNext()) {
                    SelectionKey key = it.next();
                    it.remove();

                    if (key.isAcceptable()) {
                        // 创建新的连接，并且把连接注册到selector上，而且，
                        // 声明这个channel只对读操作感兴趣。
                        SocketChannel socketChannel = ssc.accept();
                        socketChannel.configureBlocking(false);
```

```
                        socketChannel.register(selector, SelectionKey.OP_READ);
                    }
                    else if (key.isReadable()) {
                        SocketChannel socketChannel = (SocketChannel) key.channel();
                        readBuff.clear();
                        socketChannel.read(readBuff);

                        readBuff.flip();
                        System.out.println("received : " + new String(readBuff.array()));
                        key.interestOps(SelectionKey.OP_WRITE);
                    }
                    else if (key.isWritable()) {
                        writeBuff.rewind();
                        SocketChannel socketChannel = (SocketChannel) key.channel();
                        socketChannel.write(writeBuff);
                        key.interestOps(SelectionKey.OP_READ);
                    }
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

2.client端

Java

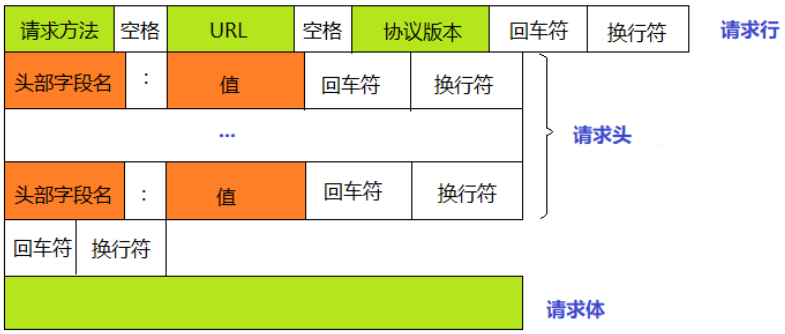
```
public class WebClient {
    public static void main(String[] args) throws IOException {
        try {
            SocketChannel socketChannel = SocketChannel.open();
            socketChannel.connect(new InetSocketAddress("127.0.0.1", 8000));

            ByteBuffer writeBuffer = ByteBuffer.allocate(32);
            ByteBuffer readBuffer = ByteBuffer.allocate(32);

            writeBuffer.put("hello".getBytes());
            writeBuffer.flip();

            while (true) {
                writeBuffer.rewind();
                socketChannel.write(writeBuffer);
                readBuffer.clear();
                socketChannel.read(readBuffer);
            }
        } catch (IOException e) {
        }
    }
}
```

请求与响应

HTTP请求与响应是通过请求报文、响应报文完成的，我们创建HttpRequest对象解析请求报文、使用HttpResponse对象构建响应报文

请求报文

客户端发起一个请求时，实际上是发送了一段请求报文的字节流数据，服务端通过NIO的channel通道来将字节流读到buffer缓冲区，然后从缓冲区读取字节数据并转换为报文文本，最后使用HttpRequest对象来将请求报文封装成请求对象。



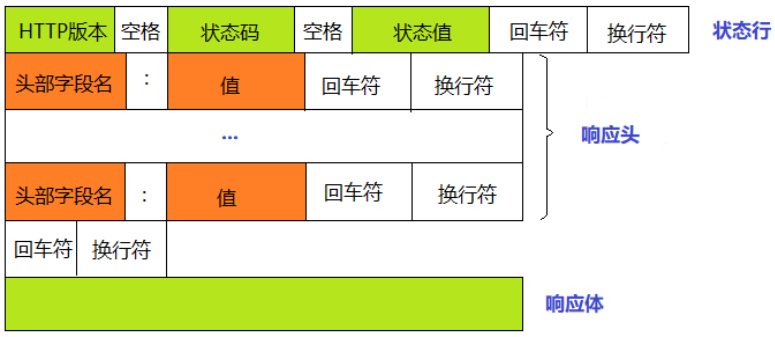
例子：

.properties

```
GET /admin_ui/rdx/core/images/close.png HTTP/1.1
Accept: */*
Referer: http://xxx.xxx.xxx.xxx/menu/neo
Accept-Language: en-US
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.1; WOW64; Trident/7.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3
Accept-Encoding: gzip, deflate
Host: xxx.xxx.xxx.xxx
Connection: Keep-Alive
Cookie: startupapp=neo; is_cisco_platform=0; rdx_pagination_size=250%20Per%20Page; SESSID=deb31b8eb9ca68a514cf55777744e339

77777777777777777777777777
```

响应报文

服务端处理业务后，使用HttpResponse对象作为为响应对象，它将内容封装为响应报文文本，再将报文文本转为字节数据，然后将字节数据放在buffer缓冲区中，最后通过NIO的channel通道将其发送给客户端。



例子：

.properties

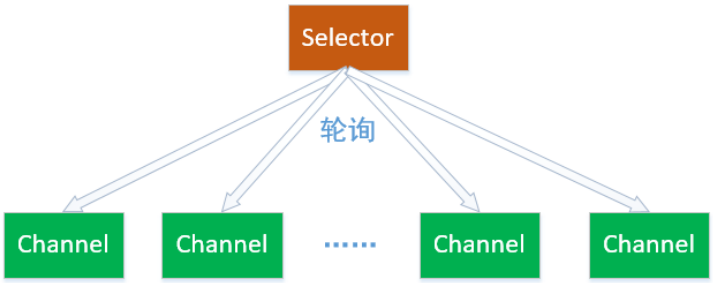
```
HTTP/1.1 200 OK
Bdpagetype: 1
Bdqid: 0xacbbb9d800005133
Cache-Control: private
Connection: Keep-Alive
Content-Encoding: gzip
Content-Type: text/html
Cxy_all: baidu+f8b5e5b521b3644ef7f3455ea441c5d0
Date: Fri, 12 Oct 2018 06:36:28 GMT
Expires: Fri, 12 Oct 2018 06:36:26 GMT
Server: BWS/1.1
Set-Cookie: delPer=0; path=/; domain=.baidu.com
Set-Cookie: BDSVRTM=0; path=/
Set-Cookie: BD_HOME=0; path=/
Set-Cookie: H_PS_PSSID=1433_21112_18560_26350_27245_22158; path=/; domain=.baidu.com
Vary: Accept-Encoding
X-Ua-Compatible: IE=Edge,chrome=1
Transfer-Encoding: chunked
```

```
<!DOCTYPE html>
<html>...</html>
```

NIO 服务器

NIO 主要由 Selector 选择器、Channel 通道、Buffer 缓冲区构成。

- Channel 负责客户端、服务端数据的运输，它有读就绪、写就绪等状态；
- Buffer 用于从 Channel 中读、写数据；
- Selector 用于轮询多个 Channel，对已就绪的 Channel 执行对应的操作。



- SelectionKey 用于维护 Selector 与 Channel 之间的关系，它可以将通道注册到选择器。

Java
```
SelectionKey key = channel.register(selector, Selectionkey.OP_READ);
```

上面这一段的作用是，修改通道的就绪状态成为一个新通道，并且将通道与选择器绑定，然后生成一个新的 key 可以被选择器 select( )方法获得。

于是我们可以通过下面的方式进行请求的几个处理阶段：

Java
```
//通道初始状态
channel.register(selector, SelectionKey.OP_ACCEPT);
...
// 开始处理
while (selector.select() > 0){
  Iterator keyIterator = selector.selectedKeys().iterator();
  while(keyIterator.hasNext()) {
      SelectionKey key = keyIterator.next();
      if(key.isAcceptable()) { // 接受了连接
          ...
          channel.register(selector,SelectionKey.OP_READ);
      } else if (key.isConnectable()) { // 建立了连接
          ...
      } else if (key.isReadable()) {
          ...
          channel.register(selector,SelectionKey.OP_WRITE);
      } else if (key.isWritable()) {
          channel.close();
      }
      keyIterator.remove();  //删除处理后的key
  }
}
```

上面这段代码主要说明：key 的作用是改变通道状态，并使选择器查找的状态为该状态。注意外层循环表示轮询。

1. 通道初始状态为 OP_ACCEPT（连接就绪），选择器对此状态感兴趣，此时状态 key 为连接就绪状态。
2. 第一轮循环：连接处理，状态 key 改为读就绪（生成新 key 删除旧 key）。
3. 第二轮循环：读处理，状态 key 改为写就绪（生成新 key 删除旧 key）。
4. 第三轮循环：写处理，关闭通道，删除 key，轮询结束。

1.创建 Selector

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

2.创建 Channel

1.建立通道

- SeverSocketChannel ： 只有一个，服务器启动后一直开启。
- SocketChannel：可以有多个，每个请求创建一个，请求结束时关闭。

```java
// 创建serverSocket连接通道
ServerSocketChannel channel = ServerSocketChannel.open();
// 设置非阻塞通道
channel.configureBlocking(false);
```

2.通道建立 Socket

```java
// 使用serverSocket连接通道打开一个serverSocket
ServerSocket serverSocket = channel.socket();
// 绑定服务端地址到serverSocket
serverSocket.bind(new InetSocketAddress("127.0.0.1",8080));
```

3.通道注册到选择器

```java
channel.register(selector, SelectionKey.OP_ACCEPT);
```

3.Selector 轮询

服务器中使用一个独立的 Selector 来轮询 ServerSocketChannel 通道的连接就绪状态，如果通道连接就绪，则创建一个线程进行处理。

```java
// 使用线程池来开启线程处理通道上的请求
ExecutorService executorService = Executors.newFixedThreadPool(100);
Selector selector = initSelector();
// 开始轮询选择器
while (selector.select() > 0){
    Iterator<SelectionKey> keys = selector.selectedKeys().iterator();
    // 监听通道状态
    while (keys.hasNext()){
        SelectionKey key = keys.next();
        keys.remove();
        if(key.isAcceptable()){
            // 如果想在线程内部获取SocketChannel通道，需要在启动线程后添加延迟
            SocketChannel channel = ((ServerSocketChannel)key.channel()).accept();
            // 启动一个线程来处理不同状态的通道
            executorService.execute(new ServerRunnable(channel));

        }
    }
}
```

4.处理线程

处理线程中，创建新的 Selector 来轮询通道的读写就绪状态

```java
public class ServerRunnable implements Runnable{

    private final SocketChannel channel;

    public ServerRunnable(SocketChannel channel) {
        this.channel = channel;
    }

    @Override
    public void run() {
        HttpRequest request = new HttpRequest();
        HttpResponse response = new HttpResponse();
        try {
            SelectorHandler handler = new SelectorHandler(Selector.open());
            Selector selector = handler.getSelector();
            channel.configureBlocking(false);
            channel.register(selector,SelectionKey.OP_READ);
```

```
18
19              // 查询所有就绪key，并处理
20              while (selector.select() > 0){
21                  Set<SelectionKey> keySet = selector.selectedKeys();
22                  Iterator<SelectionKey> iterator = keySet.iterator();
23                  // 选择器轮询选择器所有key，(key绑定的通道修改注册方式来变换进度)
24                  while (iterator.hasNext()){
25                      SelectionKey key = iterator.next();
26                      iterator.remove();
27                      if(key.isReadable()){ //开始读、并处理业务
28                          Log.info("开始接收请求数据...");
29                          request.reception(handler.read(key));
30                          Log.info("请求数据接收完成!!! 开始处理业务...");
31                          handler.requestHandler(request,response);
32                          Log.info("业务处理完成!!!");
33                      }else if (key.isWritable()){ //开始写
34                          Log.info("准备发送响应数据...");
35                          handler.writ(key,response.sendBody());
36                          Log.info("响应数据发送完成，已关闭SocketChannel通道!!!");
37                      }
38                  }
39
40              }
41          } catch (IOException e) {
42              e.printStackTrace();
43          }
44      }
45  }
```

不同状态的通道处理逻辑

Java

```
1   public class SelectorHandler implements ChannelAction, Business {
2
3       private final Selector selector;
4
5       public SelectorHandler(Selector selector) {
6           this.selector = selector;
7       }
8
9       @Override
10      public void accept(SelectionKey key) throws IOException {
11          SocketChannel channel = ((ServerSocketChannel)key.channel()).accept();
12          channel.configureBlocking(false);
13          channel.register(selector,SelectionKey.OP_READ);
14      }
15
16      @Override
17      public byte[] read(SelectionKey key) throws IOException {
18          SocketChannel channel = (SocketChannel)key.channel();
19          channel.register(selector,SelectionKey.OP_WRITE);
20          return ByteUtils.getByteByChannel(channel,1024);
21      }
22
23      @Override
24      public void writ(SelectionKey key,byte[] data) throws IOException {
25          SocketChannel channel = (SocketChannel)key.channel();
26          channel.write(ByteBuffer.wrap(data));
27          channel.close();
28      }
29
30      @Override
31      public void connect(SelectionKey key) throws IOException {
32          SocketChannel channel = (SocketChannel)key.channel();
33          channel.close();
34      }
35
36      public Selector getSelector() {
37          return selector;
38      }
39
40      @Override
41      public void requestHandler(HttpRequest request, HttpResponse response) {
42          try {
43              List<Class<?>> listClass = ClassManager.getClassByExtends("com.tom.wen.tomcat.servlet.HttpServlet");
44              for (Class<?> aClass : listClass) {
45                  ((HttpServlet)aClass.newInstance()).service(request,response);
46              }
47          } catch (Exception e) {
48              response.toServerError();
```

```
            e.printStackTrace();
        }
    }
}
```

WEB 连接处理

NIO 构建 web 服务器时，需要将连接与读写分开处理，即使用 Selector1 处理连接、Selector2 处理读写。

服务器中使用 Selector1 轮询连接状态，当监测到该状态后，之间创建一个线程处理读写，并不影响 Selector1 继续轮询就绪的连接。

Java

```
ExecutorService executorService = Executors.newFixedThreadPool(100);
NIOServer server = new NIOServer("127.0.0.1",8088);
Selector selector = server.getSelector();
while (selector.select() > 0){
    Iterator<SelectionKey> iterator = selector.selectedKeys().iterator();
    while (iterator.hasNext()){
        SelectionKey key = iterator.next();
        iterator.remove();
        if(key.isAcceptable()){
            Log.info("启动子线程处理请求...");
            // 如果想在线程内部获取SocketChannel通道，需要在启动线程后添加延迟
            SocketChannel channel = ((ServerSocketChannel)key.channel()).accept();
            executorService.execute(new ServerRunnable(channel));
            Log.info("主通道请求处理完成!!!允许下一个请求进入...");
        }
    }
}
```

上面使用 ServerSocketChannel 构建一个主通道，selector 只轮询 OP_ACCEPT（连接就绪）状态，并将结果交给线程处理，自身则立刻进行下一轮轮询。

线程内部

线程参数为 SocketChannel 通道，它通过主通道的 accept( )方法获得，它不会影响主通道的进行。

每个线程内部单独创建了一个 Selector2、并且使用了通过主通道获得的 SocketChannel 通道，他们之间通过 key 绑定，进行实际处理的进行。一个线程表示一个请求的实际处理。

Java

```
    SelectorHandler handler = new SelectorHandler(Selector.open());
    Selector selector = handler.getSelector();
    channel.configureBlocking(false);
    channel.register(selector,SelectionKey.OP_READ);

    // 查询所有就绪key，并处理
    while (selector.select() > 0){
        Set<SelectionKey> keySet = selector.selectedKeys();
        Iterator<SelectionKey> iterator = keySet.iterator();
        // 选择器轮询选择器所有key，(key绑定的通道修改注册方式来变换进度)
        while (iterator.hasNext()){
            SelectionKey key = iterator.next();
            iterator.remove();
            if(key.isReadable()){//开始读、并处理业务
                Log.info("开始接收请求数据...");
                request.reception(handler.read(key));
                Log.info("请求数据接收完成!!! 开始处理业务...");
                handler.requestHandler(request,response);
                Log.info("业务处理完成!!!");
            }else if (key.isWritable()){//开始写
                Log.info("准备发送响应数据...");
                handler.writ(key,response.sendBody());
                Log.info("响应数据发送完成，已关闭SocketChannel通道!!!");
            }
        }

    }
```

总结

主线程处理连接，主线程中 Selector 只处理连接就绪的请求，将 SeverSocketChannel 的所有就绪请求获得后，之直交给线程池，自身则继续轮询。

子线程处理请求，子线程中 Selector 处理读写状态，使用 SocketChannel 并根据请求处理进度改变状态，在处理完成后关闭 SocketChannel 通道。

- SeverSocketChannel ： 只有一个，服务器启动后一直开启。
- SocketChannel：可以有多个，每个请求创建一个，请求结束时关闭。

NIO 拓展

Java NIO

Server

Java

```
/**
 * @author 闪电侠
 */
public class NIOServer {
    public static void main(String[] args) throws IOException {
        Selector serverSelector = Selector.open();
        Selector clientSelector = Selector.open();

        new Thread(() -> {
            try {
                // 对应IO编程中的服务端启动
                ServerSocketChannel listenerChannel = ServerSocketChannel.open();
                listenerChannel.socket().bind(new InetSocketAddress(8000));
                listenerChannel.configureBlocking(false);
                listenerChannel.register(serverSelector, SelectionKey.OP_ACCEPT);

                while (true) {
                    // 监测是否有新连接，这里的1指阻塞的时间为 1ms
                    if (serverSelector.select(1) > 0) {
                        Set<SelectionKey> set = serverSelector.selectedKeys();
                        Iterator<SelectionKey> keyIterator = set.iterator();

                        while (keyIterator.hasNext()) {
                            SelectionKey key = keyIterator.next();

                            if (key.isAcceptable()) {
                                try {
                                    // (1) 每来一个新连接，不需要创建一个线程，而是直接注册到clientSelector
                                    SocketChannel clientChannel = ((ServerSocketChannel) key.channel()).accept();
                                    clientChannel.configureBlocking(false);
                                    clientChannel.register(clientSelector, SelectionKey.OP_READ);
                                } finally {
                                    keyIterator.remove();
                                }
                            }

                        }
                    }
                }
            } catch (IOException ignored) {
            }

        }).start();


        new Thread(() -> {
            try {
                while (true) {
                    // (2) 批量轮询哪些连接有数据可读，这里的1指阻塞的时间为 1ms
                    if (clientSelector.select(1) > 0) {
                        Set<SelectionKey> set = clientSelector.selectedKeys();
                        Iterator<SelectionKey> keyIterator = set.iterator();

                        while (keyIterator.hasNext()) {
                            SelectionKey key = keyIterator.next();

                            if (key.isReadable()) {
                                try {
                                    SocketChannel clientChannel = (SocketChannel) key.channel();
                                    ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
                                    // (3) 面向Buffer
                                    clientChannel.read(byteBuffer);
                                    byteBuffer.flip();
                                    System.out.println(Charset.defaultCharset().newDecoder(). decode(byteBuffer)
                                            .toString());
                                } finally {
```

```
                                keyIterator.remove();
                                key.interestOps(SelectionKey.OP_READ);
                            }
                        }

                    }
                }
            }
        } catch (IOException ignored) {
        }
    }).start();


    }
}
```

Netty NIO

maven

XML

```
  <dependency>
        <groupId>io.netty</groupId>
        <artifactId>netty-all</artifactId>
        <version>4.1.6.Final</version>
    </dependency>
```

Server

Java

```
public class NettyServer {
    public static void main(String[] args) {
        ServerBootstrap serverBootstrap = new ServerBootstrap();
        // 负责创建新连接
        NioEventLoopGroup boss = new NioEventLoopGroup();
        // 负责读取数据线程
        NioEventLoopGroup worker = new NioEventLoopGroup();
        serverBootstrap
                .group(boss, worker)
                .channel(NioServerSocketChannel.class)
                .childHandler(new ChannelInitializer<NioSocketChannel>() {
                    protected void initChannel(NioSocketChannel ch) {
                        ch.pipeline().addLast(new StringDecoder());
                        ch.pipeline().addLast(new SimpleChannelInboundHandler<String>() {
                            @Override
                            protected void channelRead0(ChannelHandlerContext ctx, String msg) {
                                System.out.println(msg);
                            }
                        });
                    }
                })
                .bind(8000);
    }
}
```

client

Java

```
 * @author 闪电侠
 */
public class NettyClient {
    public static void main(String[] args) throws InterruptedException {
        Bootstrap bootstrap = new Bootstrap();
        NioEventLoopGroup group = new NioEventLoopGroup();

        bootstrap.group(group)
                .channel(NioSocketChannel.class)
                .handler(new ChannelInitializer<Channel>() {
                    @Override
                    protected void initChannel(Channel ch) {
                        ch.pipeline().addLast(new StringEncoder());
                    }
                });

        Channel channel = bootstrap.connect("127.0.0.1", 8000).channel();
```

```
        while (true) {
            channel.writeAndFlush(new Date() + ": hello world!");
            Thread.sleep(2000);
        }
    }
}
```