链滴

从 Java NIO 到 Reactor 模式

作者：ZhangVincent

原文链接：https://ld246.com/article/1569163003025

来源网站：链滴

Java NIO

Java NIO（New IO）是jdk1.4引入的全新IO方式，与传统的Java IO相比，Java NIO具有以下特点：

- Java IO是面向字节流和字符流的IO，而Java NIO是面向通道（Channel）和缓存（Buffer）的IO数据总是从通道流向缓存，或者从缓存写入通道
- Java NIO 引入了选择器（Selector）组件，一个选择器可以监听多个通道的事件（如链接打开、数已到达等），这使得一个线程可以监听多个通道

三大核心组件

虽然Java NIO提供了很多类和组件，但是最核心的API有三个：Channel、Buffer、Selector。正如面讲的，Selector监听Channel是否已打开、可读、可写等，当Channel可读或者可写时，通过Buffe来从Channel读取数据或者向Channel写数据。

Channel

几个重要的Channel实现：

- FileChannel 负责从文件中读取数据、写入数据
- DatagramChannel 负责从UDP中读取数据、写入数据
- ServerSocketChannel 负责监听新到来的TCP连接，对于每个新的TCP连接都会创建一个SocketCannel
- SocketChannel 负责从TCP连接中读取数据、写入数据

Buffer

Buffer本质上一块可读可写的内存，向外提供一组方法使得访问更加便捷。Buffer的类型有如下几种：

- ByteBuffer
- LongBuffer
- CharBuffer
- DoubleBuffer
- ShortBuffer
- IntBuffer
- FloatBuffer
- MappedByteBuffer

可以看到，java的基础类型除boolean外，都有对应的Buffer实现。而MappedByteBuffer则比较特，是用来将文件的某个部分与内存的某个部分隐射起来，提升文件访问的读写速度。

使用allocate方法分配一个Buffer

```
ByteBuffer buf = ByteBuffer.allocate(48);
CharBuffer buf = CharBuffer.allocate(1024);
```

向Buffer写入数据

```
int bytesRead = inChannel.read(buf);//从channel读取数据，写入buffer
buffer.put(123);//将整数123写入一个IntBuffer
```

从Buffer读取数据

```
int bytesWritten = inChannel.write(buf);//从buffer中读取数据并写入channel
int number = buffer.get(); // 从IntBuffer中读取一个数据
```

Buffer的内置的四个变量和几个重要方法

每个Buffer内部都是包含有一个数组，同时维护了四个变量（mark、position、limit、capacity）来记读写位置和范围。

- capacity 用来表明数组的长度，在Buffer创建时作为allocate方法的入参。Buffer一经创建，则不改变。
- limit 用来标记数组中最后一个可写入的位置或最后一个能读取数据的位置。
- position 用来标记数组中下一个将会被读取数据的位置，或者下一个将会被写入数据的位置
- mark 可以用mark来标记数组中某个位置，以便于之后将position值设定到该位置，重新从该位置行读写

Buffer提供了 flip()/rewind()/mark()/reset()/clear()/compact() 方法来设置这四个变量：

```
//此时mark=-1,position=0,limit=1024,capacity=1024
IntBuffer buffer = IntBuffer.allocate(1024);

//200个整数放入Buffer后，mark=-1,position=200,limit=1024,capacity=1024
for(int i=0;i<200;i++){
    buffer.put(i);
}

//mark方法标记当前position所处位置，此时mark=200,position=200,limit=1024,capacity=1024
buffer.mark();

//用flip方法将Buffer从写模式切换到读模式（limit切到position位置，而position置0）
//flip方法调用后，mark=200,position=0,limit=200,capacity=1024
buffer.flip();

//读取一百整数。读取完毕后，mark=200,position=100,limit=200,capacity=1024
for(int i=0;i<100;i++){
    System.out.println(buffer.get());
}

//reset方法将postion值设置为mark值。mark=200,position=200,limit=200,capacity=1024
buffer.reset();

//rewind方法会重置position和mark。mark=-1,position=0,limit=200,capacity=1024
buffer.rewind();

//rewind之后可以从头开始重新读取数据
```

```
//读取完毕后，mark=-1,position=50,limit=200,capacity=1024
for(int i=0;i<50;i++){
    System.out.println(buffer.get());
}

//compact方法将剩余未读取的数据移动到列表头部，并将position设置到数据的尾部，limit设置为
组长度，将mark设置为-1
//compact执行完毕后，mark=-1,position=150,limit=1024,capacity=1024
buffer.compact();

//clear方法将buffer置为初始状态。执行完毕后，mark=-1,position=0,limit=1024,capacity=1024
buffer.clear();
```

Selector

可将多个设置为非阻塞模式的Channel注册到同一个Selector上，在注册时指定该通道上感兴趣的事（如连接是否建立、数据是否可读等）。因此，一个线程可通过一个selector监控、处理多个Channe
。

创建一个Selector

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

向Selector注册Channel

```
//需要将Channel设置为非阻塞模式
channel.configureBlocking(false);
//该通道上感兴趣的事件可以是SelectionKey.OP_READ，SelectionKey.OP_WRITE，
//SelectionKey.OP_CONNECT，SelectionKey.OP_ACCEPT
SelectionKey key = channel.register(selector, SelectionKey.OP_READ|SelectionKey.OP_WRITE);
//或者在注册时附加一个对象
SelectionKey key = channel.register(selector, SelectionKey.OP_READ|SelectionKey.OP_WRITE,
ew Object());
```

SelectionKey

通过注册时返回的SelectionKey，可以获取

- 注册时指定的感兴趣的事件

```
int interestSet = selectionKey.interestOps();
boolean isInterestedInAccept  = interestSet & SelectionKey.OP_ACCEPT;
boolean isInterestedInConnect = interestSet & SelectionKey.OP_CONNECT;
boolean isInterestedInRead    = interestSet & SelectionKey.OP_READ;
boolean isInterestedInWrite   = interestSet & SelectionKey.OP_WRITE;
```

- 已经就绪的感兴趣事件

```
int readySet = selectionKey.readyOps();
boolean isAcceptable  = readySet & SelectionKey.OP_ACCEPT;
boolean isConnectable = readySet & SelectionKey.OP_CONNECT;
boolean isReadable    = readySet & SelectionKey.OP_READ;
```

```
boolean isWritable   = readySet & SelectionKey.OP_WRITE;
//或者直接如下调用
selectionKey.isAcceptable();
selectionKey.isConnectable();
selectionKey.isReadable();
selectionKey.isWritable();
```

- 对应的Channel和Selector

```
Channel  channel  = selectionKey.channel();
Selector selector = selectionKey.selector();
```

- 附加在SelectionKey上的Object

```
Object attachedObj = selectionKey.attachment();
//或者通过attach方法更新附加的对象
selectionKey.attach(new Object())
```

Selector的select方法

select方法会返回上一次调用select之后，有多少通道再次变为就绪状态

```
int select()   会一直阻塞，直到某个或某几个通道注册时指定的感兴趣事件已经就绪
int select(long timeout)  与select方法类似，只不过最长会阻塞timeout毫秒
int selectNow() 不管有没有感兴趣的事件就绪，都会立即返回。如果没有感兴趣的事件就绪，返回为
```

Selector的selectedKeys方法

一旦调用了select()方法，并且返回值表明有一个或更多个通道就绪了，然后可以通过调用selector的selectedKeys()方法，获取已就绪通道的SelectKey

```
Set selectedKeys = selector.selectedKeys();
```

Selector的wakeUp方法

某个线程调用select()方法后阻塞了，即使没有通道已经就绪，只要在另外一个线程中调用同一个Selector对象的wakeUp方法，也可以唤醒该阻塞线程。

如果有其它线程调用了wakeup()方法一次或者多次，但当前没有线程阻塞在select()方法上，下个调用elect()方法的线程会立即“醒来（wake up）”。

Selector的close方法

用完Selector后调用其close()方法会关闭该Selector，且使注册到该Selector上的所有SelectionKey例无效。通道本身并不会关闭。

一个完整的网络IO示例

```
public class JavaNioServer {
```

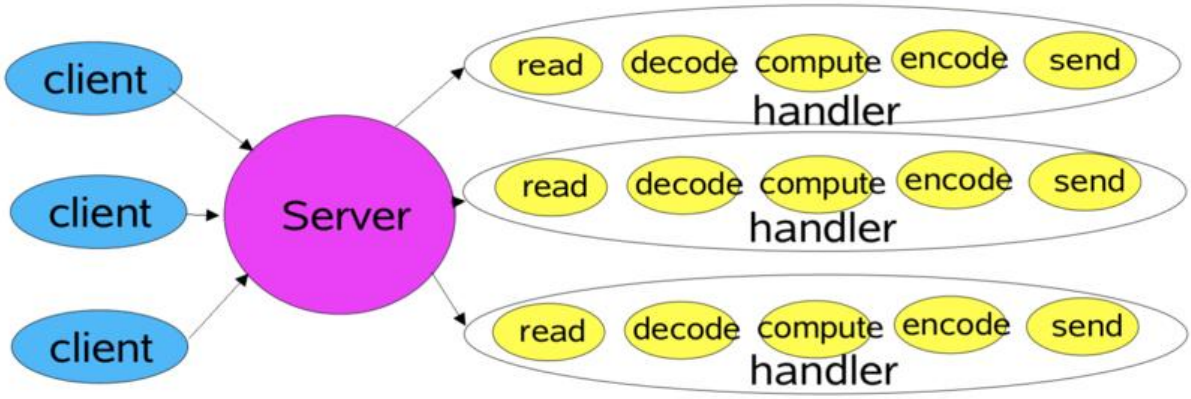
```
    public static void main(String[] args) throws IOException {
        ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
        serverSocketChannel.socket().bind(new InetSocketAddress(8080));
        serverSocketChannel.configureBlocking(false);
        Selector selector = Selector.open();
        serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
        while(true) {
            if(selector.select() <= 0) {
                continue;
            }
            Set<SelectionKey> selectedKeys = selector.selectedKeys();
            Iterator<SelectionKey> keyIterator = selectedKeys.iterator();

            while(keyIterator.hasNext()) {

                SelectionKey key = keyIterator.next();
                if(key.isAcceptable()) {
                    SocketChannel socketChannel = ((ServerSocketChannel)key.channel()).accept();
                    socketChannel.configureBlocking(false);
                    socketChannel.register(selector,SelectionKey.OP_READ);
                    System.out.println("receive a  connection from client");
                }  else if (key.isReadable()) {
                    SocketChannel socketChannel = (SocketChannel)key.channel();
                    System.out.println(NioCommonUtil.readFromChannel(socketChannel));
                    key.interestOps(SelectionKey.OP_WRITE);
                } else if (key.isWritable()) {
                    SocketChannel socketChannel = (SocketChannel)key.channel();
                    NioCommonUtil.writeToChannel("this is from server : hello    client",socketChannel
;
                    key.interestOps(SelectionKey.OP_READ);
                }
            }
            selectedKeys.clear();
        }
    }
}
```

Reactor模式

利用Java NIO，我们可以构建Reactor模式的高并发、高吞吐的网络程序。而当前大多数IO相关组件如netty、redis等都是使用的Reactor模式。这节我们将从经典多线程IO模式讲起，引入Reactor模式并介绍Reactor模式的演化。同时，我们将利用Java IO和Java NIO相关的API演示相关代码。

经典的多线程模式

在传统的网络编程中，在服务器端我们往往启动一个线程不断监听端口的连接请求。每当接收到一个接时，都会为新连接创建一个线程，每个线程单独处理业务。




```
public class BIOServer {
    public static void main(String[] args) throws IOException {
        ServerSocket serverSocket = new ServerSocket(8080);
        while (true){
            Socket socket = serverSocket.accept();
            new Handler(socket).start();
        }
    }
}

public class Handler extends Thread {
    private Socket socket;

    public Handler(Socket socket) {
        this.socket = socket;
    }

    public void readData() {
        //从socket中读取数据，有可能会阻塞
    }

    public void decode(){
        //解码
    }

    public void process() {
        //业务处理逻辑
    }

    public void encode(){
        //对结果进行编码
    }

    public void sendResult() {
        //通过socket向client端发送编码后的结果
    }

    @Override
    public void run() {
        readData();
```

原文链接：从 Java NIO 到 Reactor 模式

```
            decode();
            process();
            encode();
            sendResult();
        }
    }
```

这种一个连接对应一个线程的线程模型，具有编码简单、连接间相互不影响（某个线程阻塞了不影响他线程）的优点。但是这种模型具有以下缺点：

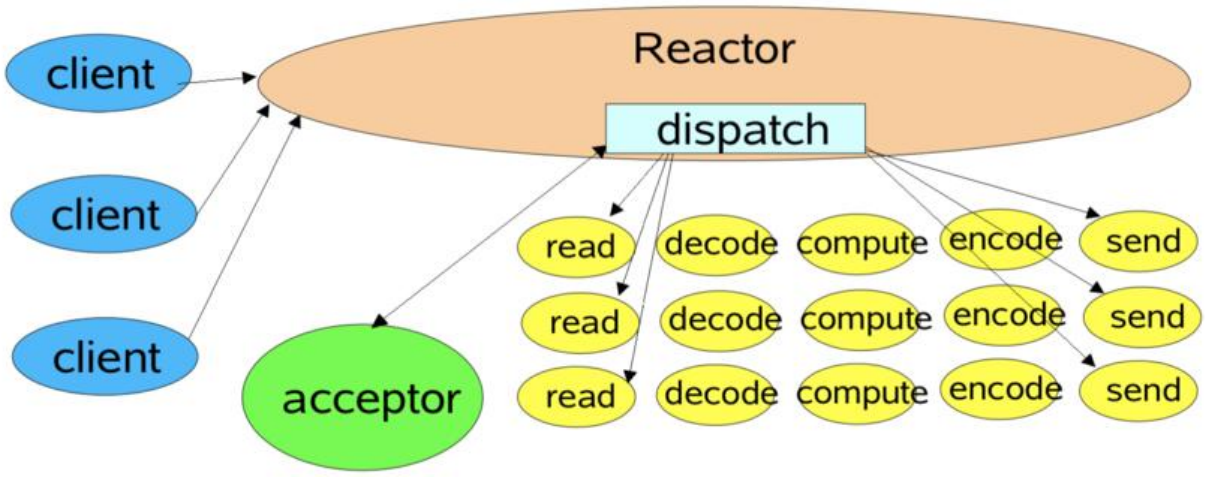
- 系统在创建、回收、切换线程时，都存在开销
- 每个线程都有可能阻塞在读数据或者写数据的环节，一旦连接数增大，系统性能会急剧下降

Reactor模式完美的克服了传统多线程模式的缺点，同时能很好的应对高并发的场景。Reactor模式的程模型，会监听连接可创建、数据可读、数据可写等事件，并将其分发给不同的逻辑进行处理。在Reator模式中，存在2种角色：

- Dispatcher 负责调用不同的Handler，分发处理逻辑
- Handler 负责创建连接、读取数据、解码、处理、编码、发送结果等逻辑

单线程Reactor模式

单线程Reactor模式，即整个过程只有一个线程，该线程会负责全部的Dispatcher和Handler的工作。




```
public class SingleThreadReactorServer {
    public static void main(String[] args) throws IOException {
        ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
        serverSocketChannel.socket().bind(new InetSocketAddress(8080));
        serverSocketChannel.configureBlocking(false);
        Selector selector = Selector.open();
        serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
        while (true) {
            int num = selector.select();
            if (num <= 0) {
                continue;
            }
```

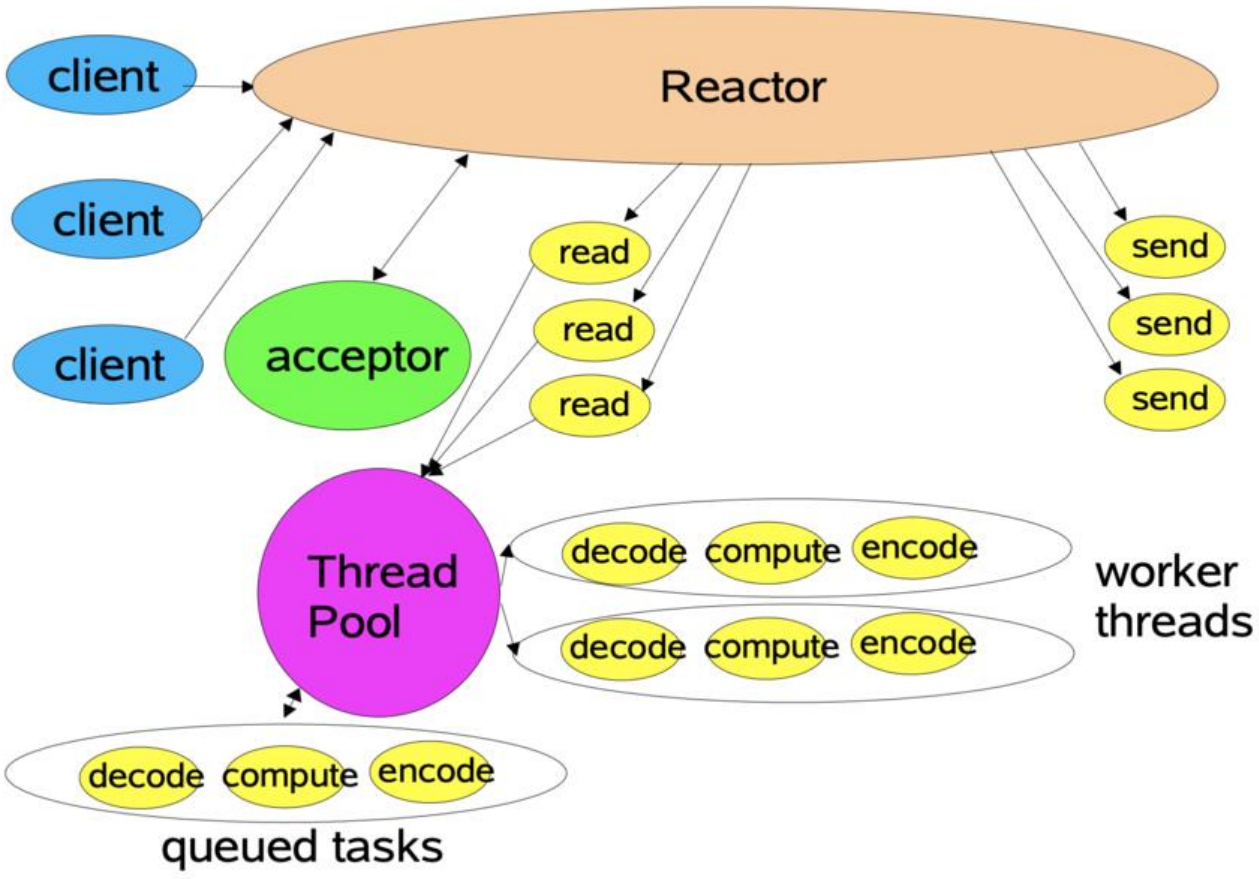
```
            Set<SelectionKey> selectionKeySet = selector.selectedKeys();
            for (SelectionKey selectionKey : selectionKeySet) {
                dispatch(selectionKey);
            }
            selectionKeySet.clear();
        }
    }

    public static void dispatch(SelectionKey selectionKey) throws IOException {
        if (selectionKey.isValid() && selectionKey.isAcceptable()) {
            new ReactorAcceptor(selectionKey).run();
        } else if (selectionKey.isValid() && selectionKey.isReadable()) {
            ((Thread) selectionKey.attachment()).run();
        } else if (selectionKey.isValid() && selectionKey.isWritable()) {
            ((Thread) selectionKey.attachment()).run();
        }else {
            selectionKey.channel().close();
        }
    }
}

public class ReactorAcceptor extends Thread {

    SelectionKey selectionKey;

    public ReactorAcceptor(SelectionKey selectionKey) {
        this.selectionKey = selectionKey;
    }

    @Override
    public void run() {
        try {
            ServerSocketChannel serverSocketChannel = (ServerSocketChannel) this.selectionKey.
hannel();
            SocketChannel socketChannel = serverSocketChannel.accept();
            socketChannel.configureBlocking(false);
            SelectionKey key = socketChannel.register(selectionKey.selector(),SelectionKey.OP_RE
D);
            key.attach(new ReactorHandler(key));
        } catch (IOException e) {

        }
    }
}


public class ReactorHandler extends Thread {

    private SelectionKey selectionKey;

    public ReactorHandler(SelectionKey selectionKey) {
        this.selectionKey = selectionKey;
    }
```

```
public void readData() {
    // 读取数据
}


public void decode() {
    //解码
}

public void process() {
    //业务处理逻辑
}

public boolean hasReceivedCompleteMsg(){
    //确认是否已经收到一个完整的数据
    return true;
}


@Override
public void run() {
    readData();
    if (hasReceivedCompleteMsg()){
        decode();
        process();
        this.selectionKey.interestOps(SelectionKey.OP_READ);
        this.selectionKey.attach(new Sender(selectionKey));
    }
}

public static class Sender extends Thread{

    private SelectionKey selectionKey;


    public Sender(SelectionKey selectionKey) {
        this.selectionKey = selectionKey;
    }

    public void encode(){

    }

    public void send(){

    }

    public boolean hasFinishResultSending(){
        //确定结果是否都已经发送完毕
        return true;
    }

    @Override
    public void run() {
```

```
            encode();
            send();
            if (hasFinishResultSending()){
                //数据发送完毕后，切换到到监听可读事件，开始下一轮读数据、解码、处理、编码、写
据的过程
                this.selectionKey.interestOps(SelectionKey.OP_READ);
                this.selectionKey.attach(new ReactorHandler(selectionKey));
            }
        }
    }
}
```

多线程的Reactor模式

在单线程的Reactor模式，由于整个过程只有一个线程在处理，所以要求Handler的业务处理逻辑能快速完成，否则会导致后续的请求处理不及时。因此这里可以改进Handler，内部使用线程池来处理务逻辑。




```
public class ReactorHandlerV2 extends Thread {

    //创建一个线程池，用于处理解码和业务逻辑
    private static final ExecutorService EXECUTOR_SERVICE = new ThreadPoolExecutor(5,10,30,
TimeUnit.SECONDS,new LinkedBlockingDeque<>());

    //省略代码
    ......
```

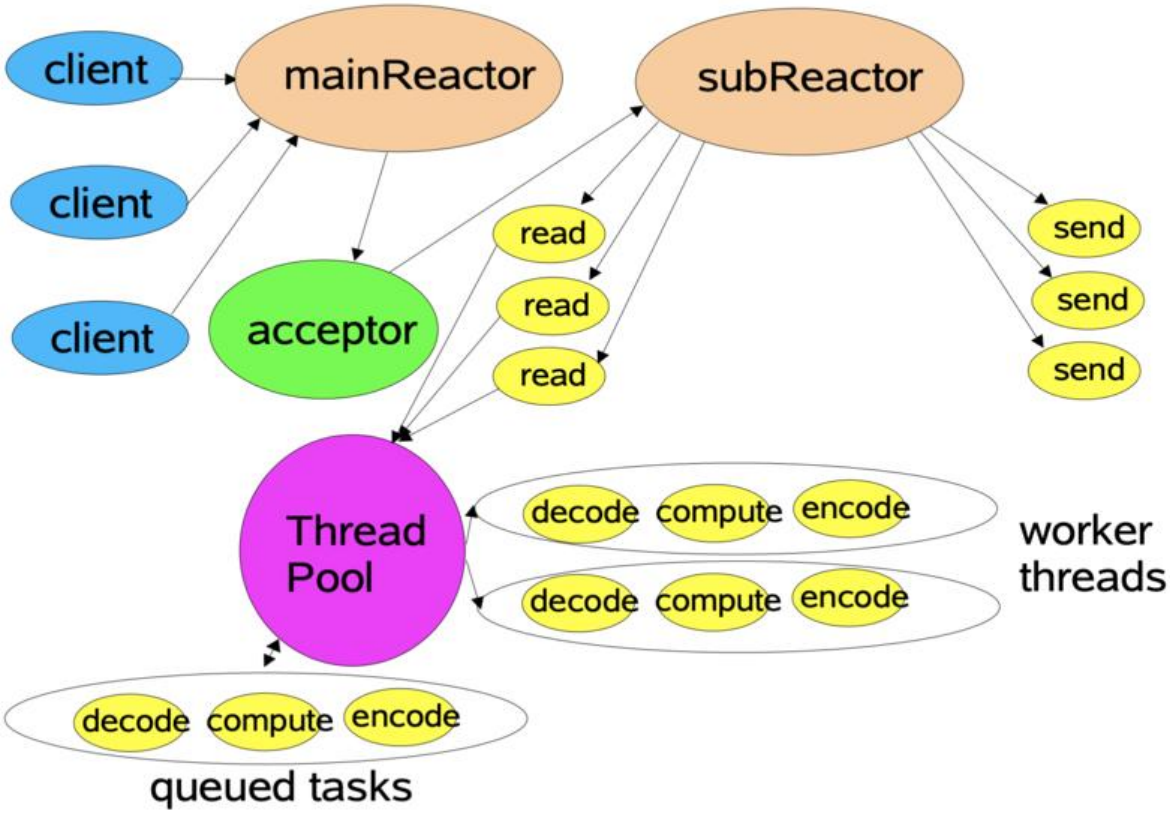
原文链接：从 Java NIO 到 Reactor 模式

```
    @Override
    public void run() {
        readData();
        if (hasReceivedCompleteMsg()){
            EXECUTOR_SERVICE.submit(()->{
                decode();
                process();
            });
            this.selectionKey.interestOps(SelectionKey.OP_READ);
            this.selectionKey.attach(new Sender(selectionKey));
        }
    }
}
```

主从多线程的Reactor模式

现代计算机，基本上都是多CPU多核机器，为了充分利用计算机性能，可以进一步地将Reactor也拆为两部分：一个主Reactor和多个从Reactor。其中，主Reactor负责监听连接，从Reactor负责读取/送数据、向线程池提交业务处理请求等。

另外一方面，linux系统中每个Selector默认情况下只能监听1024个SocketChannel，主从Reactor可提高监听的SocketChannel的个数。




```
public class ReactorAcceptorV2 extends Thread {
    /**
     * 每个子Reactor都对应一个子Selector
     */
    Selector subSelectors[];
```

原文链接：从 Java NIO 到 Reactor 模式

```
    int index = 0;
    @Override
    public void run() {
        try {
            ServerSocketChannel serverSocketChannel = (ServerSocketChannel) this.selectionKey.
hannel();
            SocketChannel socketChannel = serverSocketChannel.accept();
            socketChannel.configureBlocking(false);

            //轮询的向子Reactor中添加SocketChannel
            Selector  selector  = subSelectors[index];
            SelectionKey key = socketChannel.register(selector,SelectionKey.OP_READ);
            key.attach(new ReactorHandler(key));
            index = (index + 1) % subSelectors.length;
        } catch (IOException e) {
        }
    }
}
```

注：本文的相关代码可以查看Github