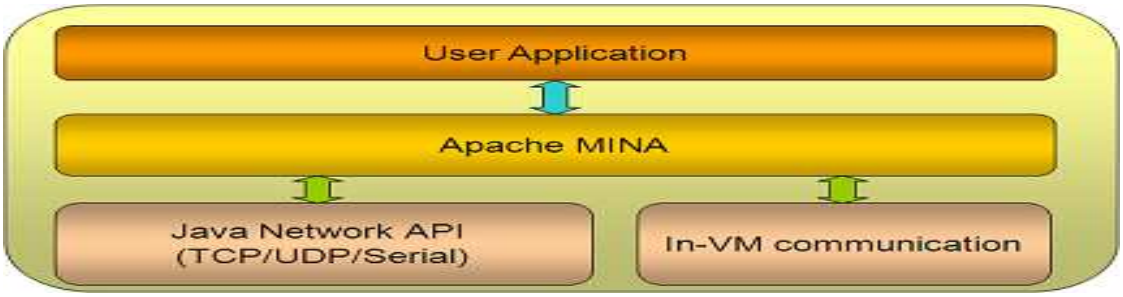
Apache Mina Server 2.0 中文参考手册

李海峰（QQ:61673110）-Andrew830314@163.com

Apache Mina Server 是一个网络通信应用框架，也就是说，它主要是对基于 TCP/IP、UDP/IP 协议栈的通信框架（当然，也可以提供 JAVA 对象的序列化服务、虚拟机管道通信服务等），Mina 可以帮助我们快速开发高性能、高扩展性的网络通信应用，Mina 提供了事件驱动、异步（Mina 的异步 IO 默认使用的是 JAVA NIO 作为底层支持）操作的编程模型。

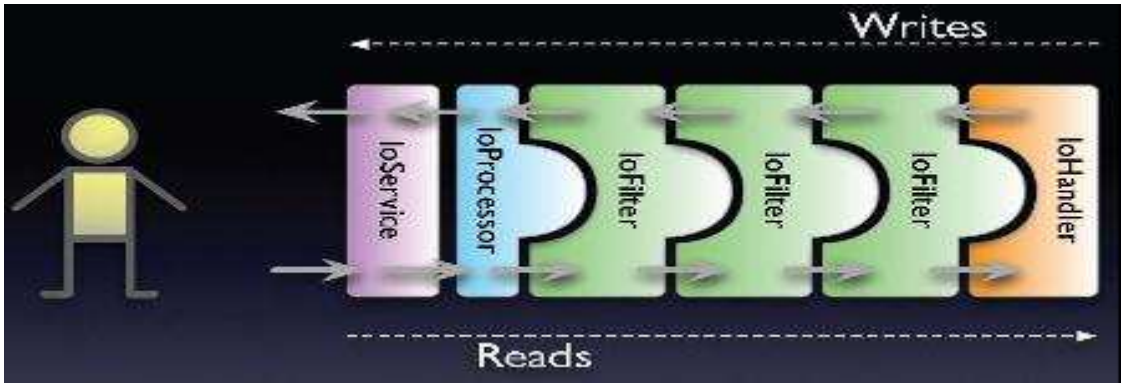
Mina 主要有 1.x 和 2.x 两个分支，这里我们讲解最新版本 2.0，如果你使用的是 Mina 1.x，那么可能会有一些功能并不适用。学习本文档，需要你已掌握 **JAVA IO、JAVA NIO、JAVA Socket、JAVA 线程及并发库(java.util.concurrent.*)**的知识。

Mina 同时提供了网络通信的 Server 端、Client 端的封装，无论是哪端，Mina 在整个网通通信结构中都处于如下的位置：



可见 Mina 的 API 将真正的网络通信与我们的应用程序隔离开来，你只需要关心你要发送、接收的数据以及你的业务逻辑即可。

同样的，无论是哪端，Mina 的执行流程如下所示：



(1.) IoService：这个接口在一个线程上负责套接字的建立，拥有自己的 Selector，监听是否有连接被建立。

(2.) IoProcessor：这个接口在另一个线程上负责检查是否有数据在通道上读写，也就是说它也拥有自己的 Selector，这是与我们使用 JAVA NIO 编码时的一个不同之处，通常在 JAVA NIO 编码中，我们都是使用一个 Selector，也就是不区分 IoService 与 IoProcessor 两个功能接口。另外，IoProcessor 负责调用注册在 IoService 上的过滤器，并在过滤器链之后调用 IoHandler。

(3.) IoFilter：这个接口定义一组拦截器，这些拦截器可以包括日志输出、黑名单过滤、数据的编码（write 方向）与解码（read 方向）等功能，其中数据的 encode 与 decode 是最为重要的、也是你在使用 Mina 时最主要关注的地方。

(4.) IoHandler：这个接口负责编写业务逻辑，也就是接收、发送数据的地方。

---

1. 简单的 TCPServer：

(1.) 第一步：编写 IoService

按照上面的执行流程，我们首先需要编写 IoService，IoService 本身既是服务端，又是客户端，我们这里编写服务端，所以使用 IoAcceptor 实现，由于 IoAcceptor 是与协议无关的，因为我们要编写 TCPServer，所以我们使用 IoAcceptor 的实现 NioSocketAcceptor，实际上底层就是调用 java.nio.channels.ServerSocketChannel 类。当然，如果你使用了 Apache 的 APR 库，那么你可以选择使用 AprSocketAcceptor 作为 TCPServer 的实现，据传说 Apache APR 库的性能比 JVM 自带的本地库高出很多。

那么 IoProcessor 是由指定的 IoService 内部创建并调用的，我们并不需要关心。

```
public class MyServer {

    main 方法：
    IoAcceptor acceptor=new NioSocketAcceptor();
    acceptor.getSessionConfig().setReadBufferSize(2048);
    acceptor.getSessionConfig.setIdleTime(IdleStatus.BOTH_IDLE,10);
    acceptor.bind(new InetSocketAddress(9123));

}
```

这段代码我们初始化了服务端的 TCP/IP 的基于 NIO 的套接字，然后调用 IoSessionConfig 设置读取数据的缓冲区大小、读写通道均在 10 秒内无任何操作就进入空闲状态。

(2.) 第二步：编写过滤器

这里我们处理最简单的字符串传输，Mina 已经为我们提供了 TextLineCodecFactory 编解码器工厂来对字符串进行编解码处理。

```
acceptor.getFilterChain().addLast("codec",
        new ProtocolCodecFilter(
            new TextLineCodecFactory(
                Charset.forName("UTF-8"),
                LineDelimeter. WINDOWS.getValue(),
                LineDelimiter. WINDOWS.getValue()
                )
```

```
            )
);
```

这段代码要在 acceptor.bind()方法之前执行，因为绑定套接字之后就不能再做这些准备工作了。

这里先不用清楚编解码器是如何工作的，这个是后面重点说明的内容，这里你只需要清楚，我们传输的以换行符为标识的数据，所以使用了 Mina 自带的换行符编解码器工厂。

(3.)　第三步：编写 IoHandler

这里我们只是简单的打印 Client 传说过来的数据。

```
public class MyIoHandler extends IoHandlerAdapter {

    // 这里我们使用的SLF4J作为日志门面，至于为什么在后面说明。
    private final static Logger log = LoggerFactory
            .getLogger(MyIoHandler.class);

    @Override
    public void messageReceived(IoSession session, Object message)
            throws Exception {
        String str = message.toString();
        log.info("The message received is [" + str + "]");
        if (str.endsWith("quit")) {
            session.close(true);
            return;
        }
    }
}
```

然后我们把这个 IoHandler 注册到 IoService：

```
acceptor.setHandler(new MyIoHandler());
```

当然这段代码也要在 acceptor.bind()方法之前执行。

然后我们运行 MyServer 中的 main 方法，你可以看到控制台一直处于阻塞状态，此时，我们用 telnet 127.0.0.1 9123 访问，然后输入一些内容，当按下回车键，你会发现数据在 Server 端被输出，但要注意不要输入中文，因为 Windows 的命令行窗口不会对传输的数据进行 UTF-8 编码。当输入 quit 结尾的字符串时，连接被断开。

这里注意你如果使用的操作系统，或者使用的 Telnet 软件的换行符是什么，如果不清楚，可以删掉第二步中的两个红色的参数，使用 TextLineCodec 内部的自动识别机制。

2.　简单的 TCPClient：

这里我们实现 Mina 中的 TCPClient，因为前面说过无论是 Server 端还是 Client 端，在 Mina 中的执行流程都是一样的。唯一不同的就是 IoService 的 Client 端实现是 IoConnector。

(1.) 第一步：编写 IoService 并注册过滤器

```
public class MyClient {

        main 方法：
        IoConnector connector=new NioSocketConnector();
        connector.setConnectTimeoutMillis(30000);
        connector.getFilterChain().addLast("codec",
                    new ProtocolCodecFilter(
                            new TextLineCodecFactory(
                                    Charset.forName("UTF-8"),
                                    LineDelimiter.WINDOWS.getValue(),
                                    LineDelimiter.WINDOWS.getValue()
                                )
                        )
        );
        connector.connect(new InetSocketAddress("localhost", 9123));

}
```

(2.) 第三步：编写 IoHandler

```
public class ClientHandler extends IoHandlerAdapter {

    private final static Logger LOGGER = LoggerFactory
            .getLogger(ClientHandler.class);

    private final String values;

    public ClientHandler(String values) {
        this.values = values;
    }

    @Override
    public void sessionOpened(IoSession session) {
        session.write(values);
    }
}
```

注册 IoHandler：

```
connector.setHandler(new ClientHandler("你好！\r\n 大家好！"));
```

然后我们运行 MyClient，你会发现 MyServer 输出如下语句：

```
The message received is [你好！]
The message received is [大家好！]
```

我们看到服务端是按照收到两条消息输出的，因为我们用的编解码器是以换行符判断数据是

否读取完毕的。

---

3. 介绍 Mina 的 TCP 的主要接口：

通过上面的两个示例，你应该对 Mina 如何编写 TCP/IP 协议栈的网络通信有了一些感性的认识。

(1.)IoService：

这个接口是服务端 IoAcceptor、客户端 IoConnector 的抽象，提供 IO 服务和管理 IoSession 的功能，它有如下几个常用的方法：

A. TransportMetadata getTransportMetadata()：

这个方法获取传输方式的元数据描述信息，也就是底层到底基于什么的实现，譬如：nio、apr 等。

B. void addListener(IoServiceListener listener)：

这个方法可以为 IoService 增加一个监听器，用于监听 IoService 的创建、活动、失效、空闲、销毁，具体可以参考 IoServiceListener 接口中的方法，这为你参与 IoService 的生命周期提供了机会。

C. void removeListener(IoServiceListener listener)：

这个方法用于移除上面的方法添加的监听器。

D. void setHandler(IoHandler handler)：

这个方法用于向 IoService 注册 IoHandler，同时有 getHandler()方法获取 Handler。

E. Map<Long,IoSession> getManagedSessions()：

这个方法获取 IoService 上管理的所有 IoSession，Map 的 key 是 IoSession 的 id。

F. IoSessionConfig getSessionConfig()：

这个方法用于获取 IoSession 的配置对象，通过 IoSessionConfig 对象可以设置 Socket 连接的一些选项。

(2.)IoAcceptor：

这个接口是 TCPServer 的接口，主要增加了 void bind()监听端口、void unbind()解除对套接字的监听等方法。这里与传统的 JAVA 中的 ServerSocket 不同的是 IoAcceptor 可以多次调用 bind()方法（或者在一个方法中传入多个 SocketAddress 参数）同时监听多个端口。

(3.)IoConnector：

这个接口是 TCPClient 的接口，主要增加了 ConnectFuture connect(SocketAddress remoteAddress,SocketAddress localAddress)方法，用于与 Server 端建立连接，第二个参数如果不传递则使用本地的一个随机端口访问 Server 端。这个方法是异步执行的，同样的，也可以同时连接多个服务端。

(4.)IoSession:

这个接口用于表示 Server 端与 Client 端的连接，IoAcceptor.accept()的时候返回实例。这个接口有如下常用的方法：

A. WriteFuture write(Object message)：

这个方法用于写数据，该操作是异步的。

B. CloseFuture close(boolean immediately)：

这个方法用于关闭 IoSession，该操作也是异步的，参数指定 true 表示立即关闭，否则就在所有的写操作都 flush 之后再关闭。

C. Object setAttribute(Object key,Object value)：

这个方法用于给我们向会话中添加一些属性，这样可以在会话过程中都可以使用，类似于 HttpSession 的 setAttrbute()方法。IoSession 内部使用同步的 HashMap 存储你添加的自定义属性。

D. SocketAddress getRemoteAddress()：

这个方法获取远端连接的套接字地址。

E. void suspendWrite()：

这个方法用于挂起写操作，那么有 void resumeWrite()方法与之配对。对于 read()方法同样适用。

F. ReadFuture read()：

这个方法用于读取数据，但默认是不能使用的，你需要调用 IoSessionConfig 的 setUseReadOperation(true)才可以使用这个异步读取的方法。一般我们不会用到这个方法，因为这个方法的内部实现是将数据保存到一个 BlockingQueue，假如是 Server 端，因为大量的 Client 端发送的数据在 Server 端都这么读取，那么可能会导致内存泄漏，但对于 Client，可能有的时候会比较便利。

G. IoService getService()：

这个方法返回与当前会话对象关联的 IoService 实例。

关于 TCP 连接的关闭:

无论在客户端还是服务端，IoSession 都用于表示底层的一个 TCP 连接，那么你会发现无论是 Server 端还是 Client 端的 IoSession 调用 close()方法之后，TCP 连接虽然显示关闭，但主线程仍然在运行，也就是 JVM 并未退出，这是因为 IoSession 的 close()仅仅是关闭了 TCP 的连接通道，并没有关闭 Server 端、Client 端的程序。你需要调用 IoService 的 dispose()方法停止 Server 端、Client 端。

(5.)IoSessionConfig:

这个方法用于指定此次会话的配置，它有如下常用的方法：

A. void setReadBufferSize(int size)：

这个方法设置读取缓冲的字节数，但一般不需要调用这个方法，因为 IoProcessor 会自动调整缓冲的大小。你可以调用 setMinReadBufferSize()、setMaxReadBufferSize()方法，这样无论 IoProcessor 无论如何自动调整，都会在你指定的区间。

B. void setIdleTime(IdleStatus status,int idleTime)：
这个方法设置关联在通道上的读、写或者是读写事件在指定时间内未发生，该通道就进入空闲状态。一旦调用这个方法，则每隔 idleTime 都会回调过滤器、IoHandler 中的 sessionIdle()方法。

C. void setWriteTimeout(int time)：
这个方法设置写操作的超时时间。

D. void setUseReadOperation(boolean useReadOperation)：
这个方法设置 IoSession 的 read()方法是否可用，默认是 false。

(6.)IoHandler：
这个接口是你编写业务逻辑的地方，从上面的示例代码可以看出，读取数据、发送数据基本都在这个接口总完成，这个实例是绑定到 IoService 上的，有且只有一个实例（没有给一个 IoService 注入一个 IoHandler 实例会抛出异常）。它有如下几个方法：

A. void sessionCreated(IoSession session)：
这个方法当一个 Session 对象被创建的时候被调用。对于 TCP 连接来说，连接被接受的时候调用，但要注意此时 TCP 连接并未建立，此方法仅代表字面含义，也就是连接的对象 IoSession 被创建完毕的时候，回调这个方法。
对于 UDP 来说，当有数据包收到的时候回调这个方法，因为 UDP 是无连接的。

B. void sessionOpened(IoSession session)：
这个方法在连接被打开时调用，它总是在 sessionCreated()方法之后被调用。对于 TCP 来说，它是在连接被建立之后调用，你可以在这里执行一些认证操作、发送数据等。
对于 UDP 来说，这个方法与 sessionCreated()没什么区别，但是紧跟其后执行。如果你每隔一段时间，发送一些数据，那么 sessionCreated()方法只会在第一次调用，但是 sessionOpened()方法每次都会调用。

C. void sessionClosed(IoSession session) ：
对于 TCP 来说，连接被关闭时，调用这个方法。
对于 UDP 来说，IoSession 的 close()方法被调用时才会毁掉这个方法。

D. void sessionIdle(IoSession session, IdleStatus status) ：
这个方法在 IoSession 的通道进入空闲状态时调用，对于 UDP 协议来说，这个方法始终不会被调用。

E. void exceptionCaught(IoSession session, Throwable cause) ：
这个方法在你的程序、Mina 自身出现异常时回调，一般这里是关闭 IoSession。

F. void messageReceived(IoSession session, Object message) ：
接收到消息时调用的方法，也就是用于接收消息的方法，一般情况下，message 是一个 IoBuffer 类，如果你使用了协议编解码器，那么可以强制转换为你需要的类型。通常我们都是会使用协议编解码器的，就像上面的例子，因为协议编解码器是 TextLineCodecFactory，所以我们可以强制转 message 为 String 类型。

G. void messageSent(IoSession session, Object message) ：
当发送消息成功时调用这个方法，注意这里的措辞，发送成功之后，也就是说发送消息是不能用这个方法的。

发送消息的时机：
发送消息应该在 sessionOpened()、messageReceived()方法中调用 IoSession.write()方法完成。因为在 sessionOpened()方法中，TCP 连接已经真正打开，同样的在 messageReceived()方法 TCP 连接也是打开状态，只不过两者的时机不同。sessionOpened()方法是在 TCP 连接建立之后，接收到数据之前发送；messageReceived()方法是在接收到数据之后发送，你可以完成依据收到的内容是什么样子，决定发送什么样的数据。

因为这个接口中的方法太多，因此通常使用适配器模式的 IoHandlerAdapter，覆盖你所感兴趣的方法即可。

(7.)IoBuffer：
这个接口是对 JAVA NIO 的 ByteBuffer 的封装，这主要是因为 ByteBuffer 只提供了对基本数据类型的读写操作，没有提供对字符串等对象类型的读写方法，使用起来更为方便，另外，ByteBuffer 是定长的，如果想要可变，将很麻烦。IoBuffer 的可变长度的实现类似于 StringBuffer。IoBuffer 与 ByteBuffer 一样，都是非线程安全的。本节的一些内容如果不清楚，可以参考 java.nio.ByteBuffer 接口。
这个接口有如下常用的方法：
A. static IoBuffer allocate(int capacity,boolean useDirectBuffer)：
这个方法内部通过 SimpleBufferAllocator 创建一个实例，第一个参数指定初始化容量，第二个参数指定使用直接缓冲区还是 JAVA 内存堆的缓存区，默认为 false。

B. void free()：
释放缓冲区，以便被一些 IoBufferAllocator 的实现重用，一般没有必要调用这个方法，除非你想提升性能（但可能未必效果明显）。

C. IoBuffer setAutoExpand(boolean autoExpand)：
这个方法设置 IoBuffer 为自动扩展容量，也就是前面所说的长度可变，那么可以看出长度可变这个特性默认是不开启的。

D. IoBuffer setAutoShrink(boolean autoShrink)：
这个方法设置 IoBuffer 为自动收缩，这样在 compact()方法调用之后，可以裁减掉一些没有使用的空间。如果这个方法没有被调用或者设置为 false，你也可以通过调用 shrink()方法手动收缩空间。

E. IoBuffer order(ByteOrder bo)：
这个方法设置是 Big Endian 还是 Little Endian，JAVA 中默认是 Big Endian，C++和其他语言一般是 Little Endian。

F. IoBuffer asReadOnlyBuffer()：
这个方法设置 IoBuffer 为只读的。

G. Boolean prefixedDataAvailable(int prefixLength,int maxDataLength)：
这个方法用于数据的最开始的 1、2、4 个字节表示的是数据的长度的情况，prefixLentgh 表示这段数据的前几个字节（只能是 1、2、4 的其中一个）的代表的是这段数据的长度，maxDataLength 表示最多要读取的字节数。返回结果依赖于等式 remaining()-prefixLength>=maxDataLength，也就是总的数据-表示长度的字节，剩下的字节数要比打算读取的字节数大或者相等。

H. String getPrefixedString(int prefixLength,CharsetDecoder decoder)：
如果上面的方法返回 true，那么这个方法将开始读取表示长度的字节之后的数据，注意要保持这两个方法的 prefixLength 的值是一样的。

G、H 两个方法在后面讲到的 PrefixedStringDecoder 中的内部实现使用。

IoBuffer 剩余的方法与 ByteBuffer 都是差不多的，额外增加了一些便利的操作方法，例如：**IoBuffer putString(String value,CharsetEncoder encoder)**可以方便的以指定的编码方式存储字符串、**InputStream asInputStream()**方法从 IoBuffer 剩余的未读的数据中转为输入流等。

(8.)IoFuture：
在 Mina 的很多操作中，你会看到返回值是 XXXFuture，实际上他们都是 IoFuture 的子类，看到这样的返回值，这个方法就说明是异步执行的，主要的子类有 ConnectFuture、CloseFuture 、 ReadFuture 、 WriteFuture 。这个接口的大部分操作都和 java.util.concurrent.Future 接口是类似的，譬如：await()、awaitUninterruptibly()等，一般我们常用 awaitUninterruptibly()方法可以等待异步执行的结果返回。
这个接口有如下常用的方法：
A. IoFuture addListener(IoFutureListener<?> listener)：
这个方法用于添加一个监听器，在异步执行的结果返回时监听器中的回调方法 operationComplete(IoFuture future)，也就是说，这是替代 awaitUninterruptibly()方法另一种等待异步执行结果的方法，它的好处是不会产生阻塞。

B. IoFuture removeListener(IoFutureListener<?> listener)：
这个方法用于移除指定的监听器。

C. IoSession getSession()：
这个方法返回当前的 IoSession。

举个例子，我们在客户端调用 connect()方法访问 Server 端的时候，实际上这就是一个异步执行的方法，也就是调用 connect()方法之后立即返回，执行下面的代码，而不管是否连接成功。那么如果我想在连接成功之后执行一些事情（譬如：获取连接成功后的 IoSession 对象），该怎么办呢？按照上面的说明，你有如下两种办法：

第一种：

```
ConnectFuture future = connector.connect(new InetSocketAddress(
                HOSTNAME, PORT));
// 等待是否连接成功，相当于是转异步执行为同步执行。
future.awaitUninterruptibly();
// 连接成功后获取会话对象。如果没有上面的等待，由于connect()方法是异步的，session可能会无法获取。
session = future.getSession();
```

第二种：

```
ConnectFuture future = connector.connect(new InetSocketAddress(
                HOSTNAME, PORT));
future.addListener(new IoFutureListener<ConnectFuture>() {

        @Override
        public void operationComplete(ConnectFuture future) {
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                e.printStackTrace();
                }
            IoSession session = future.getSession();
            System.out.println("++++++++++++++++++++++++++++");
        }
    });
System.out.println("**************");
```

为了更好的看清楚使用监听器是异步的，而不是像 awaitUninterruptibly()那样会阻塞主线程的执行，我们在回调方法中暂停 5 秒钟，然后输出+++，在最后输出***。我们执行代码之后，你会发现首先输出***(这证明了监听器是异步执行的)，然后 IoSession 对象 Created，系统暂停 5 秒，然后输出+++，最后 IoSession 对象 Opened，也就是 TCP 连接建立。

---

4. 日志配置：

前面的示例代码中提到了使用 SLF4J 作为日志门面，这是因为 Mina 内部使用的就是 SLF4J，你也使用 SLF4J 可以与之保持一致性。

Mina 如果想启用日志跟踪 Mina 的运行细节，你可以配置 LoggingFilter 过滤器，这样你可以看到 Session 建立、打开、空闲等一系列细节在日志中输出，默认 SJF4J 是按照 DEBUG 级别输出跟踪信息的，如果你想给某一类别的 Mina 运行信息输出指定日志输出级别，可以调用 LoggingFilter 的 setXXXLogLevel(LogLevel.XXX)。

例：

```
LoggingFilter lf = new LoggingFilter();
lf.setSessionOpenedLogLevel(LogLevel.ERROR);
acceptor.getFilterChain().addLast("logger", lf);
```

这里 IoSession 被打开的跟踪信息将以 ERROR 级别输出到日志。

5.过滤器：

前面我们看到了 LoggingFilter、ProtocolCodecFilter 两个过滤器，一个负责日志输出，一个负责数据的编解码，通过最前面的 Mina 执行流程图，在 IoProcessor 与 IoHandler 之间可以有很多的过滤器，这种设计方式为你提供可插拔似的扩展功能提供了非常便利的方式，目前的 Apache CXF、Apache Struts2 中的拦截器也都是一样的设计思路。

Mina 中的 IoFilter 是单例的，这与 CXF、Apache Struts2 没什么区别。

IoService 实例上会绑定一个 DefaultIoFilterChainBuilder 实例，

DefaultIoFilterChainBuilder 会把使用内部的 EntryImpl 类把所有的过滤器按照顺序连在一起，组成一个过滤器链。

DefaultIoFilterChainBuilder 类如下常用的方法：

A. void addFirst(String name,IoFilter filter)：

这个方法把过滤器添加到过滤器链的头部，头部就是 IoProcessor 之后的第一个过滤器。同样的 addLast()方法把过滤器添加到过滤器链的尾部。

B. void addBefore(String baseName,String name,IoFilter filter)：

这个方法将过滤器添加到 baseName 指定的过滤器的前面，同样的 addAfter()方法把过滤器添加到 baseName 指定的过滤器的后面。这里要注意无论是那种添加方法，每个过滤器的名字（参数 name）必须是唯一的。

C. IoFilter remove(Stirng name)：

这个方法移除指定名称的过滤器，你也可以调用另一个重载的 remove()方法，指定要移除的 IoFilter 的类型。

D. List<Entry> getAll()：

这个方法返回当前 IoService 上注册的所有过滤器。

默认情况下，过滤器链中是空的，也就是 getAll()方法返回长度为 0 的 List，但实际 Mina 内部有两个隐藏的过滤器：HeadFilter、TailFilter，分别在 List 的最开始和最末端，很明显，TailFilter 在最末端是为了调用过滤器链之后，调用 IoHandler。但这两个过滤器对你来说是透明的，可以忽略它们的存在。

编写一个过滤器很简单，你需要实现 IoFilter 接口，如果你只关注某几个方法，可以继承 IoFilterAdapter 适配器类。IoFilter 接口中主要包含两类方法，一类是与 IoHandler 中的方法名一致的方法，相当于拦截 IoHandler 中的方法，另一类是 IoFilter 的生命周期回调方法，这些回调方法的执行顺序和解释如下所示：

(1.)init()在首次添加到链中的时候被调用，但你必须将这个 IoFilter 用 ReferenceCountingFilter 包装起来，否则 init()方法永远不会被调用。
(2.)onPreAdd()在调用添加到链中的方法时被调用，但此时还未真正的加入到链。
(3.)onPostAdd()在调用添加到链中的方法后被调，如果在这个方法中有异常抛出，则过滤器会立即被移除，同时 destroy()方法也会被调用（前提是使用 ReferenceCountingFilter 包装）。
(4.)onPreRemove()在从链中移除之前调用。
(5.)onPostRemove()在从链中移除之后调用。
(6.)destory()在从链中移除时被调用，使用方法与 init()要求相同。

无论是哪个方法，要注意必须在实现时调用参数 nextFilter 的同名方法，否则，过滤器链的执行将被中断，IoHandler 中的同名方法一样也不会被执行，这就相当于 Servlet 中的 Filter 必须调用 filterChain.doFilter(request,response)才能继续前进是一样的道理。
示例：

```
public class MyIoFilter implements IoFilter {

    @Override
    public void destroy() throws Exception {
        System.out.println("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%destroy");
    }

    @Override
    public void exceptionCaught(NextFilter nextFilter, IoSession
session,
            Throwable cause) throws Exception {

    System.out.println("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%exceptionCaught");
        nextFilter.exceptionCaught(session, cause);
    }

    @Override
    public void filterClose(NextFilter nextFilter, IoSession session)
            throws Exception {
        System.out.println("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%filterClose");
        nextFilter.filterClose(session);
    }

    @Override
    public void filterWrite(NextFilter nextFilter, IoSession session,
            WriteRequest writeRequest) throws Exception {
        System.out.println("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%filterWrite");
        nextFilter.filterWrite(session, writeRequest);
    }
```

```
    @Override
    public void init() throws Exception {
        System.out.println("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%init");
    }

    @Override
    public void messageReceived(NextFilter nextFilter, IoSession
session,
            Object message) throws Exception {

    System.out.println("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%messageReceived");
        nextFilter.messageReceived(session, message);
    }

    @Override
    public void messageSent(NextFilter nextFilter, IoSession session,
            WriteRequest writeRequest) throws Exception {
        System.out.println("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%messageSent");
        nextFilter.messageSent(session, writeRequest);
    }

    @Override
    public void onPostAdd(IoFilterChain parent, String name,
            NextFilter nextFilter) throws Exception {
        System.out.println("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%onPostAdd");
    }

    @Override
    public void onPostRemove(IoFilterChain parent, String name,
            NextFilter nextFilter) throws Exception {
        System.out.println("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%onPostRemove");
    }

    @Override
    public void onPreAdd(IoFilterChain parent, String name,
            NextFilter nextFilter) throws Exception {
        System.out.println("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%onPreAdd");
    }

    @Override
    public void onPreRemove(IoFilterChain parent, String name,
            NextFilter nextFilter) throws Exception {
        System.out.println("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%onPreRemove");
    }
```

```
    @Override
    public void sessionClosed(NextFilter nextFilter, IoSession session)
            throws Exception {

    System.out.println("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%sessionClosed");
        nextFilter.sessionClosed(session);
    }

    @Override
    public void sessionCreated(NextFilter nextFilter, IoSession session)
            throws Exception {

    System.out.println("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%sessionCreated");
        nextFilter.sessionCreated(session);
    }

    @Override
    public void sessionIdle(NextFilter nextFilter, IoSession session,
            IdleStatus status) throws Exception {
        System.out.println("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%sessionIdle");
        nextFilter.sessionIdle(session, status);
    }

    @Override
    public void sessionOpened(NextFilter nextFilter, IoSession session)
            throws Exception {

    System.out.println("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%sessionOpened");
        nextFilter.sessionOpened(session);
    }

}
```

我们将这个拦截器注册到上面的 TCPServer 的 IoAcceptor 的过滤器链中的最后一个：

```
acceptor.getFilterChain().addLast("myIoFilter",
                new ReferenceCountingFilter(new MyIoFilter()));
```

这里我们将 MyIoFilter 用 ReferenceCountingFilter 包装起来，这样你可以看到 init()、destroy()方法调用。我们启动客户端访问，然后关闭客户端，你会看到执行顺序如下所示：init→onPreAdd→onPostAdd→sessionCreated→sessionOpened→messageReceived→filterClose→sessionClosed→onPreRemove→onPostRemove→destroy。

单个过滤器方法执行的顺序

IoHandler 的对应方法会跟在上面的对应方法之后执行，这也就是说从横向（单独的看一个过滤器中的所有方法的执行顺序）上看，每个过滤器的执行顺序是上面所示的顺序；从纵向

（方法链的调用）上看，如果有 filter1、filter2 两个过滤器，sessionCreated()方法的执行顺序如下所示：
filter1-sessionCreated→filter2-sessionCreated→IoHandler-sessionCreated。

这里你要注意 init、onPreAdd、onPostAdd 三个方法并不是在 Server 启动时调用的，而是 IoSession 对象创建之前调用的，也就是说 IoFilterChain.addXXX()方法仅仅负责初始化过滤器并注册过滤器，但并不调用任何方法，包括 init()初始化方法也是在 IoProcessor 开始工作的时候被调用。

IoFilter 是单例的，那么 init()方法是否只被执行一次呢？这个是不一定的，因为 IoFilter 是被 IoProcessor 调用的，而每个 IoService 通常是关联多个 IoProcessor，所以 IoFilter 的 init()方法是在每个 IoProcessor 线程上只执行一次。关于 Mina 的线程问题，我们后面会详细讨论，这里你只需要清楚，init()与 destroy()的调用次数与 IoProceesor 的个数有关，假如一个 IoService 关联了 3 个 IoProcessor，有五个并发的客户端请求，那么你会看到三次 init()方法被调用，以后将不再会调用。

Mina 中自带的过滤器：

过滤器	说明
BlacklistFilter	设置一些 IP 地址为黑名单，不允许访问。
BufferedWriteFilter	设置输出时像 BufferedOutputStream 一样进行缓冲。
CompressionFilter	设置在输入、输出流时启用 JZlib 压缩。
ConnectionThrottleFilter	这个过滤器指定同一个 IP 地址（不含端口号）上的请求在多长的毫秒值内可以有一个请求，如果小于指定的时间间隔就有连续两个请求，那么第二个请求将被忽略（IoSession.close()）。正如 Throttle 的名字一样，调节访问的频率。这个过滤器最好放在过滤器链的前面。
FileRegionWriteFilter	如果你想使用 File 对象进行输出，请使用这个过滤器。要注意，你需要使用 WriteFuture 或者在 messageSent()方法中关闭 File 所关联的 FileChannel 通道。
StreamWriteFilter	如果你想使用 InputStream 对象进行输出，请使用这个过滤器。要注意，你需要使用 WriteFuture 或者在 messageSent()方法中关闭 File 所关联的 FileChannel 通道。
NoopFilter	这个过滤器什么也不做，如果你想测试过滤器链是否起作用，可以用它来测试。
ProfilerTimerFilter	这个过滤器用于检测每个事件方法执行的时间，所以最好放在过滤器链的前面。
ProxyFilter	这个过滤器在客户端使用 ProxyConnector 作为实现时，会自动加入到过滤器链中，用于完成代理功能。
RequestResponseFilter	暂不知晓。

SessionAttributeInitializingFilter	这个过滤器在 IoSession 中放入一些属性（Map），通常放在过滤器的前面，用于放置一些初始化的信息。
MdcInjectionFilter	针对日志输出做 MDC 操作,可以参考 LOG4J 的 MDC、NDC 的文档。
WriteRequestFilter	CompressionFilter、RequestResponseFilter 的基类，用于包装写请求的过滤器。

还有一些过滤器，会在各节中详细讨论，这里没有列出，譬如：前面的 LoggingFilger 日志过滤器。

6.协议编解码器:

前面说过，协议编解码器是在使用 Mina 的时候你最需要关注的对象，因为在网络传输的数据都是二进制数据（byte），而你在程序中面向的是 JAVA 对象，这就需要你实现在发送数据时将 JAVA 对象编码二进制数据，而接收数据时将二进制数据解码为 JAVA 对象（这个可不是 JAVA 对象的序列化、反序列化那么简单的事情）。

Mina 中的协议编解码器通过过滤器 ProtocolCodecFilter 构造，这个过滤器的构造方法需要一个 ProtocolCodecFactory,这从前面注册 TextLineCodecFactory 的代码就可以看出来。

ProtocolCodecFactory 中有如下两个方法：

```
public interface ProtocolCodecFactory {

    ProtocolEncoder getEncoder(IoSession session) throws Exception;

    ProtocolDecoder getDecoder(IoSession session) throws Exception;
}
```

因此,构建一个 ProtocolCodecFactory 需要 ProtocolEncoder、ProtocolDecoder 两个实例。

你可能要问 JAVA 对象和二进制数据之间如何转换呢？这个要依据具体的通信协议，也就是 Server 端要和 Client 端约定网络传输的数据是什么样的格式，譬如：第一个字节表示数据长度,第二个字节是数据类型,后面的就是真正的数据(有可能是文字、有可能是图片等等)，然后你可以依据长度从第三个字节向后读，直到读取到指定第一个字节指定长度的数据。

简单的说，HTTP 协议就是一种浏览器与 Web 服务器之间约定好的通信协议，双方按照指定的协议编解码数据。我们再直观一点儿说，前面一直使用的 TextLine 编解码器就是在读取网络上传递过来的数据时，只要发现哪个字节里存放的是 ASCII 的 10、13 字符（\r、\n），就认为之前的字节就是一个字符串（默认使用 UTF-8 编码）。

以上所说的就是各种协议实际上就是网络七层结构中的应用层协议，它位于网络层（IP）、传输层（TCP）之上，Mina 的协议编解码器就是让你实现一套自己的应用层协议栈。

(6-1.)简单的编解码器示例：

下面我们举一个模拟电信运营商短信协议的编解码器实现，假设通信协议如下所示：

M sip:wap.fetion.com.cn SIP-C/2.0
S: 1580101xxxx
R: 1889020xxxx

这里的第一行表示状态行，一般表示协议的名字、版本号等，第二行表示短信的发送号码，第三行表示短信接收的号码，第四行表示短信的字节数，最后的内容就是短信的内容。
上面的每一行的末尾使用 ASC II 的 10（\n）作为换行符，因为这是纯文本数据，协议要求双方使用 UTF-8 对字符串编解码。
实际上如果你熟悉 HTTP 协议，上面的这个精简的短信协议和 HTTP 协议的组成是非常像的，第一行是状态行，中间的是消息报头，最后面的是消息正文。

在解析这个短信协议之前，你需要知晓 TCP 的一个事项，那就是数据的发送没有规模性，所谓的规模性就是作为数据的接收端，不知道到底什么时候数据算是读取完毕，所以应用层协议在制定的时候，必须指定数据读取的截至点。一般来说，有如下三种方式设置数据读取的长度：
(1.)使用分隔符，譬如：TextLine 编解码器。你可以使用\r、\n、NUL 这些 ASC II 中的特殊的字符来告诉数据接收端，你只要遇见分隔符，就表示数据读完了，不用在那里傻等着不知道还有没有数据没读完啊？我可不可以开始把已经读取到的字节解码为指定的数据类型了啊？
(2.)定长的字节数，这种方式是使用长度固定的数据发送，一般适用于指令发送，譬如：数据发送端规定发送的数据都是双字节，AA 表示启动、BB 表示关闭等等。
(3.)在数据中的某个位置使用一个长度域，表示数据的长度，这种处理方式最为灵活，上面的短信协议中的那个 L 就是短信文字的字节数，其实 HTTP 协议的消息报头中的 Content-Length 也是表示消息正文的长度，这样数据的接收端就知道我到底读到多长的字节数就表示不用再读取数据了。

相比较解码（字节转为 JAVA 对象，也叫做拆包）来说，编码（JAVA 对象转为字节，也叫做打包）就很简单了，你只需要把 JAVA 对象转为指定格式的字节流，write()就可以了。

下面我们开始对上面的短信协议进行编解码处理。
第一步，协议对象：

```
public class SmsObject {

    private String sender;// 短信发送者

    private String receiver;// 短信接受者

    private String message;// 短信内容

    public String getSender() {
        return sender;
    }

    public void setSender(String sender) {
        this.sender = sender;
```

```
    }

    public String getReceiver() {
        return receiver;
    }

    public void setReceiver(String receiver) {
        this.receiver = receiver;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

}
```

第二步，编码器：

在 Mina 中编写编码器可以实现 ProtocolEncoder，其中有 encode()、dispose()两个方法需要实现。这里的 dispose()方法用于在销毁编码器时释放关联的资源，由于这个方法一般我们并不关心，所以通常我们直接继承适配器 ProtocolEncoderAdapter。

```
public class CmccSipcEncoder extends ProtocolEncoderAdapter {

    private final Charset charset;

    public CmccSipcEncoder(Charset charset) {
        this.charset = charset;
    }

    @Override
    public void encode(IoSession session, Object message,
            ProtocolEncoderOutput out) throws Exception {
        SmsObject sms = (SmsObject) message;
        CharsetEncoder ce = charset.newEncoder();
        IoBuffer buffer = IoBuffer.allocate(100).setAutoExpand(true);
        String statusLine = "M sip:wap.fetion.com.cn SIP-C/2.0";
        String sender = sms.getSender();
        String receiver = sms.getReceiver();
        String smsContent = sms.getMessage();
        buffer.putString(statusLine + '\n', ce);
        buffer.putString("S: " + sender + '\n', ce);
```

```
        buffer.putString("R: " + receiver + '\n', ce);
        buffer
                .putString("L: " + (smsContent.getBytes(charset).length)
+ "\n",
                        ce);
        buffer.putString(smsContent, ce);
        buffer.flip();
        out.write(buffer);
    }

}
```

这里我们依据传入的字符集类型对 message 对象进行编码，编码的方式就是按照短信协议拼装字符串到 IoBuffer 缓冲区，然后调用 ProtocolEncoderOutput 的 write()方法输出字节流。这里要注意生成短信内容长度时的红色代码，我们使用 String 类与 Byte[]类型之间的转换方法获得转为字节流后的字节数。

解码器的编写有以下几个步骤：

A. 将 encode()方法中的 message 对象强制转换为指定的对象类型；
B. 创建 IoBuffer 缓冲区对象，并设置为自动扩展；
C. 将转换后的 message 对象中的各个部分按照指定的应用层协议进行组装，并 put()到 IoBuffer 缓冲区；
D. 当你组装数据完毕之后，调用 flip()方法，为输出做好准备，切记在 write()方法之前，要调用 IoBuffer 的 flip()方法，否则缓冲区的 position 的后面是没有数据可以用来输出的，你必须调用 flip()方法将 position 移至 0，limit 移至刚才的 position。这个 flip()方法的含义请参看 java.nio.ByteBuffer。
E. 最后调用 ProtocolEncoderOutput 的 write()方法输出 IoBuffer 缓冲区实例。

第三步，解码器：

在 Mina 中编写解码器，可以实现 ProtocolDecoder 接口，其中有 decode()、finishDecode()、dispose()三个方法。这里的 finishDecode()方法可以用于处理在 IoSession 关闭时剩余的未读取数据，一般这个方法并不会被使用到，除非协议中未定义任何标识数据什么时候截止的约定，譬如：Http 响应的 Content-Length 未设定，那么在你认为读取完数据后，关闭 TCP 连接（IoSession 的关闭）后，就可以调用这个方法处理剩余的数据，当然你也可以忽略调剩余的数据。同样的，一般情况下，我们只需要继承适配器 ProtocolDecoderAdapter，关注 decode()方法即可。

但前面说过解码器相对编码器来说，最麻烦的是数据发送过来的规模，以聊天室为例，一个 TCP 连接建立之后，那么隔一段时间就会有聊天内容发送过来，也就是 decode()方法会被往复调用，这样处理起来就会非常麻烦。那么 Mina 中幸好提供了 CumulativeProtocolDecoder 类，从名字上可以看出累积性的协议解码器，也就是说只要有数据发送过来，这个类就会去读取数据，然后累积到内部的 IoBuffer 缓冲区，但是具体的拆包（把累积到缓冲区的数据解码为 JAVA 对象）交由子类的 doDecode()方法完成，实际上 CumulativeProtocolDecoder 就是在 decode()反复的调用暴漏给子类实现的 doDecode()方法。

具体执行过程如下所示：

A. 你的 doDecode()方法返回 true 时，CumulativeProtocolDecoder 的 decode()方法会首先判断你是否在 doDecode()方法中从内部的 IoBuffer 缓冲区读取了数据，如果没有，

则会抛出非法的状态异常，也就是你的 doDecode()方法返回 true 就表示你已经消费了本次数据（相当于聊天室中一个完整的消息已经读取完毕），进一步说，也就是此时你必须已经消费过内部的 IoBuffer 缓冲区的数据（哪怕是消费了一个字节的数据）。如果验证过通过，那么 CumulativeProtocolDecoder 会检查缓冲区内是否还有数据未读取，如果有就继续调用 doDecode()方法，没有就停止对 doDecode()方法的调用，直到有新的数据被缓冲。

B. 当你的 doDecode()方法返回 false 时，CumulativeProtocolDecoder 会停止对 doDecode()方法的调用，但此时如果本次数据还有未读取完的，就将含有剩余数据的 IoBuffer 缓冲区保存到 IoSession 中，以便下一次数据到来时可以从 IoSession 中提取合并。如果发现本次数据全都读取完毕，则清空 IoBuffer 缓冲区。

简而言之，当你认为读取到的数据已经够解码了，那么就返回 true，否则就返回 false。这个 CumulativeProtocolDecoder 其实最重要的工作就是帮你完成了数据的累积，因为这个工作是很烦琐的。

```
public class CmccSipcDecoder extends CumulativeProtocolDecoder {

    private final Charset charset;

    public CmccSipcDecoder(Charset charset) {
        this.charset = charset;
    }

    @Override
    protected boolean doDecode(IoSession session, IoBuffer in,
            ProtocolDecoderOutput out) throws Exception {
        IoBuffer buffer = IoBuffer.allocate(100).setAutoExpand(true);
        CharsetDecoder cd = charset.newDecoder();
        int matchCount = 0;
         String statusLine = "", sender = "", receiver = "", length = "",
                             sms = "";
        int i = 1;
        while (in.hasRemaining()) {
            byte b = in.get();
            buffer.put(b);
            if (b == 10 && i < 5) {
                matchCount++;
                if (i == 1) {
                    buffer.flip();
                    statusLine = buffer.getString(matchCount, cd);
                    statusLine = statusLine.substring(0,
                            statusLine.length() - 1);
                    matchCount = 0;
                    buffer.clear();
                }
```

```
                if (i == 2) {
                    buffer.flip();
                    sender = buffer.getString(matchCount, cd);
                    sender = sender.substring(0, sender.length() - 1);
                    matchCount = 0;
                    buffer.clear();
                }
                if (i == 3) {
                    buffer.flip();
                    receiver = buffer.getString(matchCount, cd);
                    receiver = receiver.substring(0, receiver.length() -
1);
                    matchCount = 0;
                    buffer.clear();
                }
                if (i == 4) {
                    buffer.flip();
                    length = buffer.getString(matchCount, cd);
                    length = length.substring(0, length.length() - 1);
                    matchCount = 0;
                    buffer.clear();
                }
                i++;
            } else if (i == 5) {
                matchCount++;
                if (matchCount == Long.parseLong(length.split(": ")[1]))
{
                    buffer.flip();
                    sms = buffer.getString(matchCount, cd);
                    i++;
                    break;
                }
            } else {
                matchCount++;
            }
        }
        SmsObject smsObject = new SmsObject();
        smsObject.setSender(sender.split(": ")[1]);
        smsObject.setReceiver(receiver.split(": ")[1]);
        smsObject.setMessage(sms);
        out.write(smsObject);
        return false;
    }
}
```

我们的这个短信协议解码器使用\n（ASCII 的 10 字符）作为分解点，一个字节一个字节的读取，那么第一次发现\n 的字节位置之前的部分，必然就是短信协议的状态行，依次类推，你就可以解析出来发送者、接受者、短信内容长度。然后我们在解析短信内容时，使用获取到的长度进行读取。全部读取完毕之后，然后构造 SmsObject 短信对象，使用 ProtocolDecoderOutput 的 write()方法输出，最后返回 false，也就是本次数据全部读取完毕，告知 CumulativeProtocolDecoder 在本次数据读取中不需要再调用 doDecode()方法了。

这里需要注意的是两个状态变量 i、matchCount，i 用于记录解析到了短信协议中的哪一行（\n），matchCount 记录在当前行中读取到了哪一个字节。状态变量在解码器中经常被使用，我们这里的情况比较简单，因为我们假定短信发送是在一次数据发送中完成的，所以状态变量的使用也比较简单。假如数据的发送被拆成了多次（譬如：短信协议的短信内容、消息报头被拆成了两次数据发送），那么上面的代码势必就会存在问题，因为当第二次调用 doDecode()方法时，状态变量 i、matchCount 势必会被重置，也就是原来的状态值并没有被保存。那么我们如何解决状态保存的问题呢？
答案就是将状态变量保存在 IoSession 中或者是 Decoder 实例自身，但推荐使用前者，因为虽然 Decoder 是单例的，其中的实例变量保存的状态在 Decoder 实例销毁前始终保持，但 Mina 并不保证每次调用 doDecode()方法时都是同一个线程（这也就是说第一次调用 doDecode()是 IoProcessor-1 线程，第二次有可能就是 IoProcessor-2 线程），这就会产生多线程中的实例变量的可视性（Visibility，具体请参考 JAVA 的多线程知识）问题。IoSession 中使用一个同步的 HashMap 保存对象，所以你不需要担心多线程带来的问题。
使用 IoSession 保存解码器的状态变量通常的写法如下所示：

A. 在解码器中定义私有的内部类 Context，然后将需要保存的状态变量定义在 Context 中存储。
B. 在解码器中定义方法获取这个 Context 的实例，这个方法的实现要优先从 IoSession 中获取 Context。

具体代码示例如下所示：

```
// 上下文作为保存状态的内部类的名字，意思很明显，就是让状态跟随上下文，在整个调用过程中都可以被保持。
public class XXXDecoder extends CumulativeProtocolDecoder{

    private    final   AttributeKey CONTEXT =
                                    new   AttributeKey(getClass(),   "context" );
    public Context getContext(IoSession session){
        Context ctx=(Context)session.getAttribute(CONTEXT);
        if(ctx==null){
                ctx=new Context();
                session.setAttribute(CONTEXT,ctx);
         }
     }
     private    class   Context {
           //状态变量
     }
}
```

注意这里我们使用了 Mina 自带的 AttributeKey 类来定义保存在 IoSession 中的对象的键值，这样可以有效的防止键值重复。另外，要注意在全部处理完毕之后，状态要复位，譬如：聊天室中的一条消息读取完毕之后，状态变量要变为初始值，以便下次处理时重新使用。

第四步，编解码工厂：

```
public class CmccSipcCodecFactory implements ProtocolCodecFactory {

    private final CmccSipcEncoder encoder;

    private final CmccSipcDecoder decoder;

    public CmccSipcCodecFactory() {
        this(Charset.defaultCharset());
    }

    public CmccSipcCodecFactory(Charset charSet) {
        this.encoder = new CmccSipcEncoder(charSet);
        this.decoder = new CmccSipcDecoder(charSet);
    }

    @Override
    public ProtocolDecoder getDecoder(IoSession session) throws
Exception {
        return decoder;
    }

    @Override
    public ProtocolEncoder getEncoder(IoSession session) throws
Exception {
        return encoder;
    }

}
```

实际上这个工厂类就是包装了编码器、解码器，通过接口中的 getEncoder()、getDecoder() 方法向 ProtocolCodecFilter 过滤器返回编解码器实例，以便在过滤器中对数据进行编解码处理。

第五步，运行示例：

下面我们修改最一开始的示例中的 MyServer、MyClient 的代码，如下所示：

```
acceptor.getFilterChain().addLast(
                "codec",
                new ProtocolCodecFilter(new CmccSipcCodecFactory(Charset
                        .forName("UTF-8"))));
```

```
connector.getFilterChain().addLast(
                        "codec",
                        new ProtocolCodecFilter(new
CmccSipcCodecFactory(
                                Charset.forName("UTF-8"))));
```

然后我们在 ClientHandler 中发送一条短信：

```
public void sessionOpened(IoSession session) {
    SmsObject sms = new SmsObject();
    sms.setSender("15801012253");
    sms.setReceiver("18869693235");
    sms.setMessage("你好! Hello World!");
    session.write(sms);
}
```

最后我们在 MyIoHandler 中接收这条短信息：

```
public void messageReceived(IoSession session, Object message)
          throws Exception {
    SmsObject sms = (SmsObject) message;
    log.info("The message received is [" + sms.getMessage() + "]");
}
```

你会看到 Server 端的控制台输出如下信息：

```
The message received is [你好! Hello World!]
```

(6-2.)复杂的解码器：

下面我们讲解一下如何在解码器中保存状态变量，也就是真正的实现上面所说的 Context。

我们假设这样一种情况，有两条短信：

M sip:wap.fetion.com.cn SIP-C/2.0
S: 1580101xxxx
R: 1889020xxxx
L: 21
Hello World!
M sip:wap.fetion.com.cn SIP-C/2.0
S: 1580101xxxx
R: 1889020xxxx
L: 21
Hello World!

他们按照上面的颜色标识发送，也就是说红色部分、蓝色部分、绿色部分分别发送（调用三次 IoSession.write()方法），那么如果你还用上面的 CmccSipcDecoder，将无法工作，因为第一次数据流（红色部分）发送过取时，数据是不完整的，无法解析出一条短信息，当二次数据流（蓝色部分）发送过去时，已经可以解析出第一条短信息了，但是第二条短信还是不完整的，需要等待第三次数据流（绿色部分）的发送。

注意：由于模拟数据发送的规模性问题很麻烦，所以这里采用了这种极端的例子说明问题，

虽不具有典型性，但很能说明问题，这就足够了，所以不要追究这种发送消息是否在真实环境中存在，更不要追究其合理性。

CmccSispcDecoder 类改为如下的写法：

```
public class CmccSipcDecoder extends CumulativeProtocolDecoder {

    private final Charset charset;

    private final AttributeKey CONTEXT = new AttributeKey(getClass(),
"context");

    public CmccSipcDecoder(Charset charset) {
        this.charset = charset;
    }

    @Override
    protected boolean doDecode(IoSession session, IoBuffer in,
            ProtocolDecoderOutput out) throws Exception {
        Context ctx = getContext(session);
        CharsetDecoder cd = charset.newDecoder();
        int matchCount = ctx.getMatchCount();
        int line = ctx.getLine();
        IoBuffer buffer = ctx.innerBuffer;
        String statusLine = ctx.getStatusLine(),
                sender = ctx.getSender(),
                receiver = ctx.getReceiver(),
                length = ctx.getLength(),
                sms = ctx.getSms();
        while (in.hasRemaining()) {
            byte b = in.get();
            matchCount++;
            buffer.put(b);
            if (line < 4 && b == 10) {
                if (line == 0) {
                    buffer.flip();
                    statusLine = buffer.getString(matchCount, cd);
                    statusLine = statusLine.substring(0,
                            statusLine.length() - 1);
                    matchCount = 0;
                    buffer.clear();
                    ctx.setStatusLine(statusLine);
                }
                if (line == 1) {
                    buffer.flip();
```

```
                    sender = buffer.getString(matchCount, cd);
                    sender = sender.substring(0, sender.length() - 1);
                    matchCount = 0;
                    buffer.clear();
                    ctx.setSender(sender);
                }
                if (line == 2) {
                    buffer.flip();
                    receiver = buffer.getString(matchCount, cd);
                    receiver = receiver.substring(0, receiver.length() -
1);
                    matchCount = 0;
                    buffer.clear();
                    ctx.setReceiver(receiver);
                }
                if (line == 3) {
                    buffer.flip();
                    length = buffer.getString(matchCount, cd);
                    length = length.substring(0, length.length() - 1);
                    matchCount = 0;
                    buffer.clear();
                    ctx.setLength(length);
                }
                line++;
            } else if (line == 4) {
                if (matchCount == Long.parseLong(length.split(": ")[1]))
{
                    buffer.flip();
                    sms = buffer.getString(matchCount, cd);
                    ctx.setSms(sms);
                    // 由于下面的break，这里需要调用else外面的两行代码
                    ctx.setMatchCount(matchCount);
                    ctx.setLine(line);
                    break;
                }
            }
            ctx.setMatchCount(matchCount);
            ctx.setLine(line);
        }
        if (ctx.getLine() == 4
                && Long.parseLong(ctx.getLength().split(": ")[1]) == ctx
                        .getMatchCount()) {
            SmsObject smsObject = new SmsObject();
            smsObject.setSender(sender.split(": ")[1]);
```

```
            smsObject.setReceiver(receiver.split(": ")[1]);
            smsObject.setMessage(sms);
            out.write(smsObject);
            ctx.reset();
            return true;
        } else {
            return false;
        }
    }

    private Context getContext(IoSession session) {
        Context context = (Context) session.getAttribute(CONTEXT);
        if (context == null) {
            context = new Context();
            session.setAttribute(CONTEXT, context);
        }
        return context;
    }

    private class Context {

        private final IoBuffer innerBuffer;

        private String statusLine = "";

        private String sender = "";

        private String receiver = "";

        private String length = "";

        private String sms = "";

        public Context() {
            innerBuffer = IoBuffer.allocate(100).setAutoExpand(true);
        }

        private int matchCount = 0;

        private int line = 0;

        public int getMatchCount() {
            return matchCount;
        }
```

```
public void setMatchCount(int matchCount) {
    this.matchCount = matchCount;
}

public int getLine() {
    return line;
}

public void setLine(int line) {
    this.line = line;
}

public String getStatusLine() {
    return statusLine;
}

public void setStatusLine(String statusLine) {
    this.statusLine = statusLine;
}

public String getSender() {
    return sender;
}

public void setSender(String sender) {
    this.sender = sender;
}

public String getReceiver() {
    return receiver;
}

public void setReceiver(String receiver) {
    this.receiver = receiver;
}

public String getLength() {
    return length;
}

public void setLength(String length) {
    this.length = length;
}
```

```
        public String getSms() {
            return sms;
        }

        public void setSms(String sms) {
            this.sms = sms;
        }

        public void reset() {
            this.innerBuffer.clear();
            this.matchCount = 0;
            this.line = 0;
            this.statusLine = "";
            this.sender = "";
            this.receiver = "";
            this.length = "";
            this.sms = "";
        }

    }
}
```

这里我们做了如下的几步操作：

(1.) 所有记录状态的变量移到了 Context 内部类中，包括记录读到短信协议的哪一行的 line。每一行读取了多少个字节的 matchCount，还有记录解析好的状态行、发送者、接受者、短信内容、累积数据的 innerBuffer 等。这样就可以在数据不能完全解码，等待下一次 doDecode()方法的调用时，还能承接上一次调用的数据。

(2.) 在 doDecode()方法中主要的变化是各种状态变量首先是从 Context 中获取，然后操作之后，将最新的值 setXXX()到 Context 中保存。

(3.) 这里注意 doDecode()方法最后的判断，当认为不够解码为一条短信息时，返回 false，也就是在本次数据流解码中不要再调用 doDecode()方法；当认为已经解码出一条短信息时，输出短消息，然后重置所有的状态变量，返回 true，也就是如果本次数据流解码中还有没解码完的数据，继续调用 doDecode()方法。

下面我们对客户端稍加改造，来模拟上面的红、蓝、绿三次发送聊天短信息的情况：

MyClient：

```
ConnectFuture future = connector.connect(new InetSocketAddress(
                HOSTNAME, PORT));
future.awaitUninterruptibly();
session = future.getSession();
for (int i = 0; i < 3; i++) {
    SmsObject sms = new SmsObject();
    session.write(sms);
```

```
    System.out.println("****************" + i);
}
```

这里我们为了方便演示，不在 IoHandler 中发送消息，而是直接在 MyClient 中发送，你要注意的是三次发送都要使用同一个 IoSession，否则就不是从同一个通道发送过去的了。

CmccSipcEncoder：

```
public void encode(IoSession session, Object message,
            ProtocolEncoderOutput out) throws Exception {
        SmsObject sms = (SmsObject) message;
        CharsetEncoder ce = charset.newEncoder();
        String statusLine = "M sip:wap.fetion.com.cn SIP-C/2.0";
        String sender = "15801012253";
        String receiver = "15866332698";
        String smsContent = "你好! Hello World!";

        IoBuffer buffer = IoBuffer.allocate(100).setAutoExpand(true);
        buffer.putString(statusLine + '\n', ce);
        buffer.putString("S: " + sender + '\n', ce);
        buffer.putString("R: " + receiver + '\n', ce);
        buffer.flip();
        out.write(buffer);

        IoBuffer buffer2 = IoBuffer.allocate(100).setAutoExpand(true);
       buffer2.putString("L: " + (smsContent.getBytes(charset).length)
               + "\n",ce);
        buffer2.putString(smsContent, ce);
        buffer2.putString(statusLine + '\n', ce);
        buffer2.flip();
        out.write(buffer2);

        IoBuffer buffer3 = IoBuffer.allocate(100).setAutoExpand(true);
        buffer3.putString("S: " + sender + '\n', ce);
        buffer3.putString("R: " + receiver + '\n', ce);
        buffer3.putString("L: " + (smsContent.getBytes(charset).length)
              + "\n",ce);
        buffer3.putString(smsContent, ce);
        buffer3.putString(statusLine + '\n', ce);
        buffer3.flip();
        out.write(buffer3);

}
```

上面的这段代码要配合MyClient来操作，你需要做的是在MyClient中的红色输出语句处设置断点，然后第一调用时CmccSipcEncoder中注释掉蓝、绿色的代码，也就是发送两条短信息的第一部分（红色的代码），依次类推，也就是MyClient的中的三次断点中，分别执行

CmccSipcEncoder中的红、蓝、绿三段代码，也就是模拟两条短信的三段发送。
你会看到Server端的运行结果是：当MyClient第一次到达断点时，没有短信息被读取到，当MyClient第二次到达断点时，第一条短信息输出，当MyClient第三次到达断点时，第二条短信息输出。

Mina 中自带的解码器：

解码器	说明
CumulativeProtocolDecoder	累积性解码器，上面我们重点说明了这个解码器的用法。
SynchronizedProtocolDecoder	这个解码器用于将任何一个解码器包装为一个线程安全的解码器，用于解决上面说的每次执行 decode()方法时可能线程不是上一次的线程的问题，但这样会在高并发时，大大降低系统的性能。
TextLineDecoder	按照文本的换行符（Windows:\r\n、Linux:\n、Mac:\r）解码数据。
PrefixedStringDecoder	这个类继承自 CumulativeProtocolDecoder 类，用于读取数据最前端的 1、2、4 个字节表示后面的数据长度的数据。譬如：一个段数据的前两个字节表示后面的真实数据的长度，那么你就可以用这个方法进行解码。

(6-3.)多路分离的解码器：
假设一段数据发送过来之后,需要根据某种条件决定使用哪个解码器,而不是像上面的例子，固定使用一个解码器，那么该如何做呢？
幸好 Mina 提供了 org.apache.mina.filter.codec.demux 包来完成这种多路分离（Demultiplexes）的解码工作，也就是同时注册多个解码器，然后运行时依据传入的数据决定到底使用哪个解码器来工作。所谓多路分离就是依据条件分发到指定的解码器，譬如：上面的短信协议进行扩展，可以依据状态行来判断使用 1.0 版本的短信协议解码器还是 2.0 版本的短信协议解码器。

下面我们使用一个简单的例子，说明这个多路分离的解码器是如何使用的，需求如下所示：
(1.) 客户端传入两个 int 类型的数字，还有一个 char 类型的符号。
(2.) 如果符号是+，服务端就是用 1 号解码器，对两个数字相加，然后把结果返回给客户端。
(3.) 如果符号是-，服务端就使用 2 号解码器，将两个数字变为相反数，然后相加，把结果返回给客户端。

Demux 开发编解码器主要有如下几个步骤：
A. 定义 Client 端、Server 端发送、接收的数据对象。
B. 使用 Demux 编写编码器是实现 MessageEncoder<T>接口，T 是你要编码的数据对象，这个 MessageEncoder 会在 DemuxingProtocolEncoder 中调用。
C. 使用 Demux 编写编码器是实现 MessageDecoder 接口，这个 MessageDecoder 会在

DemuxingProtocolDecoder 中调用。

D. 在 DemuxingProtocolCodecFactory 中调用 addMessageEncoder()、addMessageDecoder()方法组装编解码器。

MessageEncoder 的接口如下所示：

```
public interface MessageEncoder<T> {
    void encode(IoSession session, T message, ProtocolEncoderOutput out)
            throws Exception;
}
```

你注意到消息编码器接口与在 ProtocolEncoder 中没什么不同，区别就是 Object message 被泛型具体化了类型，你不需要手动的类型转换了。

MessageDecoder 的接口如下所示：

```
public interface MessageDecoder {

    static MessageDecoderResult OK = MessageDecoderResult.OK;

    static MessageDecoderResult NEED_DATA =
                                    MessageDecoderResult.NEED_DATA;

    static MessageDecoderResult NOT_OK = MessageDecoderResult.NOT_OK;

    MessageDecoderResult decodable(IoSession session, IoBuffer in);

    MessageDecoderResult decode(IoSession session, IoBuffer in,
            ProtocolDecoderOutput out) throws Exception;

    void finishDecode(IoSession session, ProtocolDecoderOutput out)
            throws Exception;
}
```

(1.)decodable()方法有三个返回值，分别表示如下的含义：

A. MessageDecoderResult.NOT_OK：表示这个解码器不适合解码数据，然后检查其它解码器，如果都不满足会抛异常；

B. MessageDecoderResult.NEED_DATA：表示当前的读入的数据不够判断是否能够使用这个解码器解码，然后再次调用 decodable()方法检查其它解码器，如果都是 NEED_DATA，则等待下次输入；

C. MessageDecoderResult.OK：表示这个解码器可以解码读入的数据，然后则调用 MessageDecoder 的 decode()方法。

这里注意 decodable()方法对参数 IoBuffer in 的任何操作在方法结束之后，都会复原，也就是你不必担心在调用 decode()方法时，position 已经不在缓冲区的起始位置。这个方法相当于是预读取，用于判断是否是可用的解码器。

(2.)decode()方法有三个返回值，分别表示如下的含义：

A. MessageDecoderResult.NOT_OK：表示解码失败，会抛异常；

B. MessageDecoderResult.NEED_DATA：表示数据不够，需要读到新的数据后，再次调用decode()方法。
C. MessageDecoderResult.OK：表示解码成功。

代码演示：

(1.)客户端发送的数据对象：

```
public class SendMessage {

    private int i = 0;

    private int j = 0;

    private char symbol = '+';

    public char getSymbol() {
        return symbol;
    }

    public void setSymbol(char symbol) {
        this.symbol = symbol;
    }

    public int getI() {
        return i;
    }

    public void setI(int i) {
        this.i = i;
    }

    public int getJ() {
        return j;
    }

    public void setJ(int j) {
        this.j = j;
    }

}
```

(2.)服务端发送的返回结果对象：

```
public class ResultMessage {

    private int result = 0;
```

```
    public int getResult() {
        return result;
    }

    public void setResult(int result) {
        this.result = result;
    }

}
```

(3.)客户端使用的 SendMessage 的编码器：

```
public class SendMessageEncoder implements MessageEncoder<SendMessage>
{

    @Override
    public void encode(IoSession session, SendMessage message,
            ProtocolEncoderOutput out) throws Exception {
        IoBuffer buffer = IoBuffer.allocate(10);
        buffer.putChar(message.getSymbol());
        buffer.putInt(message.getI());
        buffer.putInt(message.getJ());
        buffer.flip();
        out.write(buffer);
    }

}
```

这里我们的 SendMessage、ResultMessage 中的字段都是用长度固定的基本数据类型，这样 IoBuffer 就不需要自动扩展了，提高性能。按照一个 char、两个 int 计算，这里的 IoBuffer 只需要 10 个字节的长度就可以了。

(4.)服务端使用的 SendMessage 的 1 号解码器：

```
public class SendMessageDecoderPositive implements MessageDecoder {

    @Override
   public MessageDecoderResult decodable(IoSession session, IoBuffer in)
{
        if (in.remaining() < 2)
            return MessageDecoderResult.NEED_DATA;
        else {
            char symbol = in.getChar();
            if (symbol == '+') {
                return MessageDecoderResult.OK;
            } else {
```

```
                return MessageDecoderResult.NOT_OK;
            }
        }
    }

    @Override
    public MessageDecoderResult decode(IoSession session, IoBuffer in,
            ProtocolDecoderOutput out) throws Exception {
        SendMessage sm = new SendMessage();
        sm.setSymbol(in.getChar());
        sm.setI(in.getInt());
        sm.setJ(in.getInt());
        out.write(sm);
        return MessageDecoderResult.OK;
    }

    @Override
    public void finishDecode(IoSession session, ProtocolDecoderOutput
out)
            throws Exception {
        // undo
    }

}
```

因为客户端发送的 SendMessage 的前两个字节(char)就是符号位,所以我们在 decodable()方法中对此条件进行了判断，之后读到两个字节，并且这两个字节表示的字符是+时，才认为这个解码器可用。

(5.)服务端使用的 SendMessage 的 2 号解码器：

```
public class SendMessageDecoderNegative implements MessageDecoder {

    @Override
   public MessageDecoderResult decodable(IoSession session, IoBuffer in)
{
        if (in.remaining() < 2)
            return MessageDecoderResult.NEED_DATA;
        else {
            char symbol = in.getChar();
            if (symbol == '-') {
                return MessageDecoderResult.OK;
            } else {
                return MessageDecoderResult.NOT_OK;
            }
        }
```

```
    }

    @Override
    public MessageDecoderResult decode(IoSession session, IoBuffer in,
            ProtocolDecoderOutput out) throws Exception {
        SendMessage sm = new SendMessage();
        sm.setSymbol(in.getChar());
        sm.setI(-in.getInt());
        sm.setJ(-in.getInt());
        out.write(sm);
        return MessageDecoderResult.OK;
    }

    @Override
    public void finishDecode(IoSession session, ProtocolDecoderOutput
out)
            throws Exception {
        // undo
    }

}
```

(6.)服务端使用的 ResultMessage 的编码器：

```
public class ResultMessageEncoder implements
MessageEncoder<ResultMessage> {

    @Override
    public void encode(IoSession session, ResultMessage message,
            ProtocolEncoderOutput out) throws Exception {
        IoBuffer buffer = IoBuffer.allocate(4);
        buffer.putInt(message.getResult());
        buffer.flip();
        out.write(buffer);
    }

}
```

(7.)客户端使用的 ResultMessage 的解码器：

```
public class ResultMessageDecoder implements MessageDecoder {

    @Override
   public MessageDecoderResult decodable(IoSession session, IoBuffer in)
{
        if (in.remaining() < 4)
```

```
            return MessageDecoderResult.NEED_DATA;
        else if (in.remaining() == 4)
            return MessageDecoderResult.OK;
        else
            return MessageDecoderResult.NOT_OK;
    }

    @Override
    public MessageDecoderResult decode(IoSession session, IoBuffer in,
            ProtocolDecoderOutput out) throws Exception {
        ResultMessage rm = new ResultMessage();
        rm.setResult(in.getInt());
        out.write(rm);
        return MessageDecoderResult.OK;
    }

    @Override
    public void finishDecode(IoSession session, ProtocolDecoderOutput
out)
            throws Exception {
        // undo
    }

}
```

(8.)组装这些编解码器的工厂：

```
public class MathProtocolCodecFactory extends
DemuxingProtocolCodecFactory {

    public MathProtocolCodecFactory(boolean server) {
        if (server) {
            super.addMessageEncoder(ResultMessage.class,
                    ResultMessageEncoder.class);
            super.addMessageDecoder(SendMessageDecoderPositive.class);
            super.addMessageDecoder(SendMessageDecoderNegative.class);
        } else {
            super
                    .addMessageEncoder(SendMessage.class,
                            SendMessageEncoder.class);
            super.addMessageDecoder(ResultMessageDecoder.class);
        }
    }

}
```

这个工厂类我们使用了构造方法的一个布尔类型的参数，以便其可以在 Server 端、Client 端同时使用。我们以 Server 端为例，你可以看到调用两次 addMessageDecoder()方法添加了 1 号、2 号解码器，其实 DemuxingProtocolDecoder 内部在维护了一个 MessageDecoder 数组，用于保存添加的所有的消息解码器，每次 decode()的时候就调用每个 MessageDecoder 的 decodable()方法逐个检查，只要发现一个 MessageDecoder 不是对应的解码器，就从数组中移除，直到找到合适的 MessageDecoder，如果最后发现数组为空，就表示没找到对应的 MessageDecoder，最后抛出异常。

(9.)Server 端：

```
public class Server {

    public static void main(String[] args) throws Exception {
        IoAcceptor acceptor = new NioSocketAcceptor();
        LoggingFilter lf = new LoggingFilter();
        acceptor.getSessionConfig().setIdleTime(IdleStatus.BOTH_IDLE,
5);
        acceptor.getFilterChain().addLast("logger", lf);
        acceptor.getFilterChain().addLast("codec",
                new ProtocolCodecFilter(new
MathProtocolCodecFactory(true)));
        acceptor.setHandler(new ServerHandler());
        acceptor.bind(new InetSocketAddress(9123));
    }
}
```

(10.)Server 端使用的 IoHandler：

```
public class ServerHandler extends IoHandlerAdapter {

    private final static Logger log = LoggerFactory
            .getLogger(ServerHandler.class);

    @Override
    public void sessionIdle(IoSession session, IdleStatus status)
            throws Exception {
        session.close(true);
    }

    @Override
    public void messageReceived(IoSession session, Object message)
            throws Exception {
        SendMessage sm = (SendMessage) message;
        log.info("The message received is [ " + sm.getI() + " "
                + sm.getSymbol() + " " + sm.getJ() + " ]");
        ResultMessage rm = new ResultMessage();
```

```
        rm.setResult(sm.getI() + sm.getJ());
        session.write(rm);
    }

}
```

(11.)Client 端：

```
public class Client {

    public static void main(String[] args) throws Throwable {
        IoConnector connector = new NioSocketConnector();
        connector.setConnectTimeoutMillis(30000);
        connector.getFilterChain().addLast("logger", new
LoggingFilter());
        connector.getFilterChain().addLast("codec",
                new ProtocolCodecFilter(new
MathProtocolCodecFactory(false)));
        connector.setHandler(new ClientHandler());
        connector.connect(new InetSocketAddress("localhost", 9123));
    }
}
```

(12.)Client 端的 IoHandler：

```
public class ClientHandler extends IoHandlerAdapter {

    private final static Logger LOGGER = LoggerFactory
            .getLogger(ClientHandler.class);

    @Override
    public void sessionOpened(IoSession session) throws Exception {
        SendMessage sm = new SendMessage();
        sm.setI(100);
        sm.setJ(99);
        sm.setSymbol('+');
        session.write(sm);
    }

    @Override
    public void messageReceived(IoSession session, Object message) {
        ResultMessage rs = (ResultMessage) message;
        LOGGER.info(String.valueOf(rs.getResult()));
    }

}
```

你尝试改变(12.)中的红色代码中的正负号，会看到服务端使用了两个不同的解码器对其进行处理。

7.线程模型配置：

Mina 中的很多执行环节都使用了多线程机制，用于提高性能。Mina 中默认在三个地方使用了线程：

(1.) IoAcceptor：

这个地方用于接受客户端的连接建立，每监听一个端口（每调用一次 bind()方法），都启用一个线程，这个数字我们不能改变。这个线程监听某个端口是否有请求到来，一旦发现，则创建一个 IoSession 对象。因为这个动作很快，所以有一个线程就够了。

(2.) IoConnector：

这个地方用于与服务端建立连接，每连接一个服务端（每调用一次 connect()方法），就启用一个线程，我们不能改变。同样的，这个线程监听是否有连接被建立，一旦发现，则创建一个 IoSession 对象。因为这个动作很快，所以有一个线程就够了。

(3.) IoProcessor：

这个地方用于执行真正的 IO 操作，默认启用的线程个数是 CPU 的核数+1，譬如：单 CPU 双核的电脑，默认的 IoProcessor 线程会创建 3 个。这也就是说一个 IoAcceptor 或者 IoConnector 默认会关联一个 IoProcessor 池，这个池中有 3 个 IoProcessor。因为 IO 操作耗费资源，所以这里使用 IoProcessor 池来完成数据的读写操作，有助于提高性能。这也就是前面说的 IoAccetor、IoConnector 使用一个 Selector，而 IoProcessor 使用自己单独的 Selector 的原因。

那么为什么 IoProcessor 池中的 IoProcessor 数量只比 CPU 的核数大 1 呢？因为 IO 读写操作是耗费 CPU 的操作，而每一核 CPU 同时只能运行一个线程，因此 IoProcessor 池中的 IoProcessor 的数量并不是越多越好。

这个 IoProcessor 的数量可以调整，如下所示：

```
IoAcceptor acceptor=new NioSocketAcceptor(5);
IoConnector connector=new NioSocketConnector(5);
```

这样就会将 IoProcessor 池中的数量变为 5 个，也就是说可以同时处理 5 个读写操作。

还记得前面说过 Mina 的解码器要使用 IoSession 保存状态变量，而不是 Decoder 本身，这是因为 Mina 不保证每次执行 doDecode()方法的都是同一个 IoProcessor 这句话吗？其实这个问题的根本原因是 IoProcessor 是一个池，每次 IoSession 进入空闲状态时（无读些数据发生），IoProcessor 都会被回收到池中，以便其他的 IoSession 使用，所以当 IoSession 从空闲状态再次进入繁忙状态时，IoProcessor 会再次分配给其一个 IoProcessor 实例，而此时已经不能保证还是上一次繁忙状态时的那个 IoProcessor 了。

你还会发现 IoAcceptor、IoConnector 还有一个构造方法，你可以指定一个 java.util.concurrent.Executor 类作为线程池对象，那么这个线程池对象是做什么用的呢？其实就是用于创建(1.)、(2.)中的用于监听是否有 TCP 连接建立的那个线程，默认情况下，使用 Executors.newCachedThreadPool()方法创建 Executor 实例，也就是一个无界的

线程池（具体内容请参看 JAVA 的并发库）。大家不要试图改变这个 Executor 的实例，也就是使用内置的即可，否则可能会造成一些莫名其妙的问题，譬如：性能在某个访问量级别时，突然下降。因为无界线程池是有多少个 Socket 建立，就分配多少个线程，如果你改为 Executors 的其他创建线程池的方法，创建了一个有界线程池，那么一些请求将无法得到及时响应，从而出现一些问题。

下面我们完整的综述一下 Mina 的工作流程：

(1.) 当 IoService 实例创建的时候，同时一个关联在 IoService 上的 IoProcessor 池、线程池也被创建；

(2.) 当 IoService 建立套接字（IoAcceptor 的 bind()或者是 IoConnector 的 connect()方法被调用）时，IoService 从线程池中取出一个线程，监听套接字端口；

(3.) 当 IoService 监听到套接字上有连接请求时，建立 IoSession 对象，从 IoProcessor 池中取出一个 IoProcessor 实例执行这个会话通道上的过滤器、IoHandler；

(4.) 当这条 IoSession 通道进入空闲状态或者关闭时，IoProcessor 被回收。

上面说的是 Mina 默认的线程工作方式，那么我们这里要讲的是如何配置 IoProcessor 的多线程工作方式。因为一个 IoProcessor 负责执行一个会话上的所有过滤器、IoHandler，也就是对于 IO 读写操作来说，是单线程工作方式（就是按照顺序逐个执行）。假如你想让某个事件方法（譬如：sessionIdle()、sessionOpened()等）在单独的线程中运行（也就是非 IoProcessor 所在的线程），那么这里就需要用到一个 ExecutorFilter 的过滤器。

你可以看到 IoProcessor 的构造方法中有一个参数是 java.util.concurrent.Executor，也就是可以让 IoProcessor 调用的过滤器、IoHandler 中的某些事件方法在线程池中分配的线程上独立运行，而不是运行在 IoProcessor 所在的线程。

例：

```
acceptor.getFilterChain().addLast("exceutor", new ExecutorFilter());
```

我们看到是用这个功能，简单的一行代码就可以了。那么 ExecutorFilter 还有许多重载的构造方法，这些重载的有参构造方法，参数主要用于指定如下信息：

(1.) 指定线程池的属性信息，譬如：核心大小、最大大小、等待队列的性质等。

你特别要关注的是 ExecutorFilter 内部默认使用的是 OrderedThreadPoolExecutor 作为线程池的实现，从名字上可以看出是保证各个事件在多线程执行中的顺序（譬如：各个事件方法的执行是排他的，也就是不可能出现两个事件方法被同时执行；messageReceived()总是在 sessionClosed()方法之前执行），这是因为多线程的执行是异步的，如果没有 OrderedThreadPoolExecutor 来保证 IoHandler 中的方法的调用顺序，可能会出现严重的问题。但是如果你的代码确实没有依赖于 IoHandler 中的事件方法的执行顺序，那么你可以使用 UnorderedThreadPoolExecutor 作为线程池的实现。

因此，你也最好不要改变默认的 Executor 实现，否则，事件的执行顺序就会混乱，譬如：messageReceived()、messageSent()方法被同时执行。

(2.) 哪些事件方法被关注，也就哪些事件方法用这个线程池执行。

线程池可以异步执行的事件类型是位于 IoEventType 中的九个枚举值中除了 SESSION_CREATED 之外的其余八个，这说明 Session 建立的事件只能与 IoProcessor 在同一个线程上执行。

```
public enum IoEventType {
    SESSION_CREATED,
    SESSION_OPENED,
    SESSION_CLOSED,
    MESSAGE_RECEIVED,
    MESSAGE_SENT,
    SESSION_IDLE,
    EXCEPTION_CAUGHT,
    WRITE,
    CLOSE,
}
```

默认情况下,没有配置关注的事件类型,有如下六个事件方法会被自动使用线程池异步执行:

```
IoEventType.EXCEPTION_CAUGHT,
IoEventType.MESSAGE_RECEIVED,
IoEventType.MESSAGE_SENT,
IoEventType.SESSION_CLOSED,
IoEventType.SESSION_IDLE,
IoEventType.SESSION_OPENED
```

其实 ExecutorFilter 的工作机制很简单，就是在调用下一个过滤器的事件方法时，把其交给 Executor 的 execute(Runnable runnable)方法来执行，其实你自己在 IoHandler 或者某个过滤器的事件方法中开启一个线程，也可以完成同样的功能，只不过这样做，你就失去了程序的可配置性，线程调用的代码也会完全耦合在代码中。但要注意的是绝对不能开启线程让其执行 sessionCreated()方法。

如果你真的打算使用这个 ExecutorFilter，那么最好想清楚它该放在过滤器链的哪个位置，针对哪些事件做异步处理机制。一般 ExecutorFilter 都是要放在 ProtocolCodecFilter 过滤器的后面，也就是不要让编解码运行在独立的线程上，而是要运行在 IoProcessor 所在的线程，因为编解码处理的数据都是由 IoProcessor 读取和发送的，没必要开启新的线程，否则性能反而会下降。一般使用 ExecutorFilter 的典型场景是将业务逻辑（譬如：耗时的数据库操作）放在单独的线程中运行，也就是说与 IO 处理无关的操作可以考虑使用 ExecutorFilter 来异步执行。

---

8. Spring 集成:

Mina 与 Spring 集成很简单，我们这里以 Server 端为例，如下所示：

```
<!-- 构造协议编解码过滤器 -->
<bean id="cmccSipcCodecFilter"
      class="org.apache.mina.filter.codec.ProtocolCodecFilter">
    <constructor-arg>
        <bean
class="com.cmcc.fetionwap.ms.mina.codec.sipc.CmccSipcCodecFactory">
            <constructor-arg index="0" type="java.nio.charset.Charset">
```

```
                <value>UTF-8</value>
            </constructor-arg>
        </bean>
    </constructor-arg>
</bean>

<!-- 构造日志过滤器 -->
<bean id="cmccSipcloggingFilter"
class="org.apache.mina.filter.logging.LoggingFilter" />

<!-- 构造过滤器链-->
<bean id="cmccSipcFilterChainBuilder"
    class="org.apache.mina.core.filterchain.DefaultIoFilterChainBuild
er">
    <property name="filters">
        <map>
            <entry key="cmccSipcloggingFilter"
                value-ref="cmccSipcloggingFilter" />
            <entry key="cmccSipcCodecFilter"
                value-ref="cmccSipcCodecFilter" />
        </map>
    </property>
</bean>

<!-- 构造属性编辑器 -->
<bean
class="org.springframework.beans.factory.config.CustomEditorConfigure
r">
    <property name="customEditors">
        <map>
            <entry key="java.net.SocketAddress">
                <bean
class="org.apache.mina.integration.beans.InetSocketAddressEditor" />
            </entry>
            <entry key="java.nio.charset.Charset">
               <bean
        class="com.cmcc.fetionwap.ms.mina.beans.CharsetEditor" />
            </entry>
        </map>
    </property>
</bean>

<!-- 构造Server端-->
<bean id="cmccSipcIoAcceptor"
```

```
class="org.apache.mina.transport.socket.nio.NioSocketAcceptor"
        init-method="bind" destroy-method="unbind">
    <property name="defaultLocalAddress" value=":9123" />
    <property name="handler" ref="sipcServerHandler" />
    <property name="filterChainBuilder"
ref="cmccSipcFilterChainBuilder" />
</bean>
```

这里我们看到与 Spring 集成只不过是把代码方式的写法按照编写顺序改成了 XML 的形式，无外乎没有脱离 Spring 使用构造器、setXXX()的注入属性的方法。这里你唯一需要注意的就是“属性编辑器”，这是因为一些属性在 XML 中写为 String 类型，但实际 JAVA 类型中要求注入的是一个其他的对象类型，你需要对此做出转换。譬如：上面的“构造 Server 端”的参数 *defaultLocalAddress* 我们传入的是一个字符串，但实际上 NioSocketAcceptor 中需要的是一个 InetSocketAddress，这里就需要一个编辑器将 XML 中注入的字符串构造为 InetSocketAddress 对象。在 Mina 自带的包 org.apache.mina.integration.beans 中提供了很多的属性编辑器，如果你发现某个属性的编辑器没有提供，可以自行编写，譬如：上面的协议编解码过滤器中的字符集实际需要的是一个 java.nio.charset.Charset，因此我编写了如下的属性编辑器：

```
public class CharsetEditor extends PropertyEditorSupport {

    private Object value;

    @Override
    public void setAsText(String text) throws IllegalArgumentException
    {
        if (text != null)
            this.value = Charset.forName(text);
        else
            this.value = Charset.forName("UTF-8");
    }

    @Override
    public Object getValue() {
        return value;
    }

}
```

我们看到这里完成了 UTF-8 这个字符串到 java.nio.charset.Charset 的转换。具体的关于 Spring 的属性编辑器的内容，请参看 Spring 的文档。

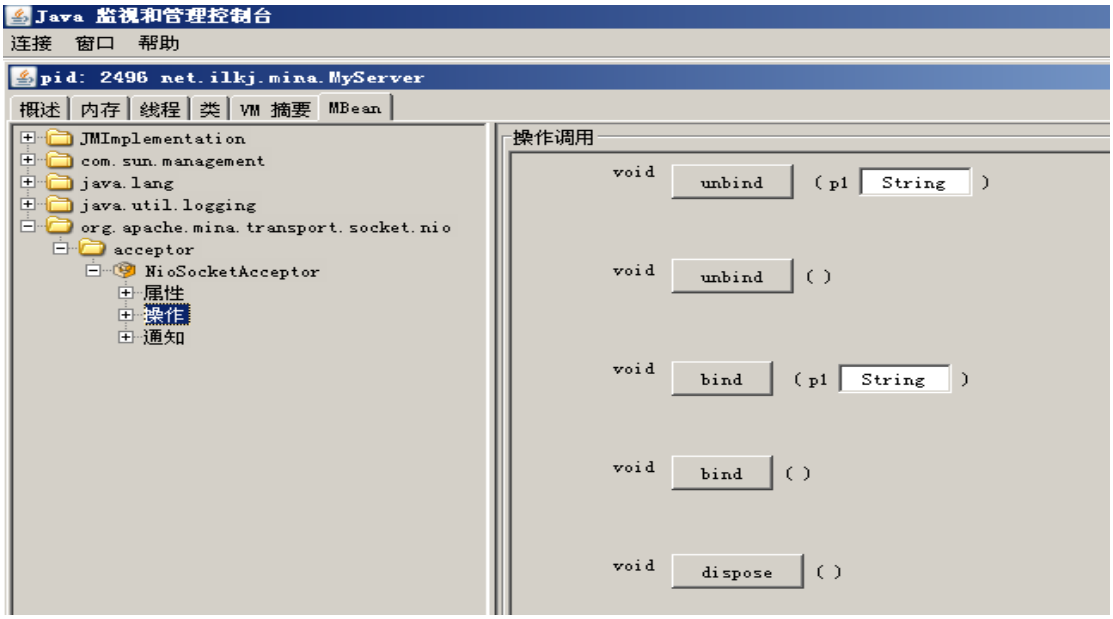
---

9. JMX 集成：

Mina 提供 org.apache.mina.integration.jmx 包可以让你管理 Mina 中的一些对象，我们举例如下所示：

```
MBeanServer mBeanServer = ManagementFactory.getPlatformMBeanServer();
IoServiceMBean acceptorMBean = new IoServiceMBean(acceptor);
ObjectName acceptorName = new ObjectName(acceptor.getClass()
                .getPackage().getName()+ ":type=acceptor,name=" +
acceptor.getClass().getSimpleName());
mBeanServer.registerMBean(acceptorMBean, acceptorName);
```

通过这段代码，我们就将 IoAcceptor 的实例 acceptor 纳入到 JMX 的管理，我们运行 Server 端，然后我们启用 JConsole，连接 MyServer 所在的 JVM 进程，如下图所示：



我们看到NioSocketAcceptor成为了 JMX 的受管组件，你可以在这里查看并修改每一个属性，调用每一个方法，譬如：你单击 dispose 按钮，会发现 MyServer 所在的进程被结束掉，因为 NioSocketAcceptor 所关联的所有资源都被销毁。

除了上面演示所用的 IoServiceMBean 之外，Mina 还未我们提供了 IoFilterMBean、IoSessionMBean、ObjectMBean 分别用于将 IoFilter、IoSession 以及普通的 JAVA Bean 对象包装为 JMX 的受管组件。

---

10. UDP 协议的示例：

Mina 中使用 UDP 协议与 TCP 协议没什么太大的区别，只不过是实现类换成了 XXXDatagramYYY，这就是使用框架的好处，统一了编程模型，所以我们只是简单的看一下例子。

(1.)UDPServer：

```
public class UDPServer {

    public static void main(String[] args) throws Exception {
        IoAcceptor acceptor = new NioDatagramAcceptor();
        acceptor.setHandler(new UDPServerHandler());
        acceptor.getFilterChain().addLast("logger", new
                                    LoggingFilter());
```

```
        ((DatagramSessionConfig) acceptor.getSessionConfig())
                .setReuseAddress(true);
        acceptor.bind(new InetSocketAddress(9122));
    }

}
```

我们这里使用 UDP 协议绑定了 9122 端口，同时设置了这个端口可以被重用。

(2.) UDPServerHandler：

```
public class UDPServerHandler extends IoHandlerAdapter {

    private final static Logger log = LoggerFactory
            .getLogger(UDPServerHandler.class);

    public void messageReceived(IoSession session, Object message)
            throws Exception {
        IoBuffer buffer = (IoBuffer) message;
        String msg = buffer.getString(3,
Charset.forName("UTF-8").newDecoder());
        log.info("The message received is [" + msg + "]");
    }

    @Override
    public void sessionClosed(IoSession session) throws Exception {
        log.debug("******************* Session Closed!");
    }

    @Override
    public void sessionCreated(IoSession session) throws Exception {
        log.debug("******************* Session Created!");
    }

    @Override
    public void sessionIdle(IoSession session, IdleStatus status)
            throws Exception {
        log.debug(session + "******************* Session Idle!");
    }

    @Override
    public void sessionOpened(IoSession session) throws Exception {
        log.debug("******************* Session Opened!");
    }
```

```
    @Override
    public void exceptionCaught(IoSession session, Throwable cause)
            throws Exception {
        log.error(cause.getMessage());
        session.close(false);
    }

    @Override
    public void messageSent(IoSession session, Object message) throws
Exception {
        log.debug("******************* messageSent!");
    }

}
```

这里我们在 messageReceived()中接收数据，为了简单，我们这里不使用编解码过滤器，因此 message 转换为 IoBuffer，稍后你会看到客户端只发送了三个英文字母，所以这里我们读取三个字节就可以了。

(3.) UDPClient：

```
public class UDPClient {

    public static void main(String[] args) throws Exception {
        IoConnector connector = new NioDatagramConnector();
        connector.setHandler(new UDPClientHandler());
        ConnectFuture connFuture = connector.connect(new
InetSocketAddress(
                "localhost", 9122));
        connFuture.awaitUninterruptibly();
        IoSession session = connFuture.getSession();
        IoBuffer buffer = IoBuffer.allocate(3);
        buffer.putString("OK!",
Charset.forName("UTF-8").newEncoder());
        buffer.flip();
        WriteFuture future = session.write(buffer);
        future.awaitUninterruptibly();
        connector.dispose();
    }

}
```

UDPClientHandler 的代码全都是空实现，因为我们在 main()方法中发送数据。

运行程序，你会发现数据发送、接收成功。