Mina 2.0 工作原理以及配置中的注意事项

1. Mina 是什么?

Apache MINA 是一个网络应用程序框架，用来帮助用户简单地开发高性能和高可靠性的网络应用程序。它提供了一个通过 Java NIO 在不同的传输例如 TCP/IP 和 UDP/IP 上抽象的事件驱动的异步 API。

Apache MINA 也称为:

- NIO 框架库
- 客户端服务器框架库
- 一个网络套接字库

MINA 虽然简单但是仍然提供了全功能的网络应用程序框架：

- 为不同的传输类型提供了统一的 API:
  - 通过 Java NIO 提供 TCP/IP 和 UDP/IP 支持
  - 通过 RXTX 提供串口通讯(RS232)
  - In-VM 管道通讯
  - 你能实现你自己的 API!
- 过滤器作为一个扩展特性; 类似 Servlet 过滤器
- 低级和高级的 API:
  - 低级: 使用字节缓存(ByteBuffers)
  - 高级: 使用用户定义的消息对象(objects)和编码(codecs)
- 高度定制化线程模型:
  - 单线程
  - 一个线程池
  - 一个以上的线程池(也就是 SEDA)
- 使用 Java5 SSL 引擎提供沙盒(Out-of-the-box) SSL・TLS・StartTLS 支持
- 超载保护和传输流量控制
- 利用模拟对象进行单元测试
- JMX 管理能力
- 通过 StreamIoHandler 提供基于流的 I/O 支持
- 和知名的容器(例如 PicoContainer、Spring)集成
- 从 Netty 平滑的迁移到 MINA， Netty 是 MINA 的前辈。

2. 框架架构和执行流程

框架同时提供了网络通信的 server 端和 client 的封装，无论在哪端，mina 都处于用户应用程序和底层实现的中间层，起到了软件分层的作用。在实际开发过

程中，用户只需要关心要发送的数据和处理逻辑就可以了。

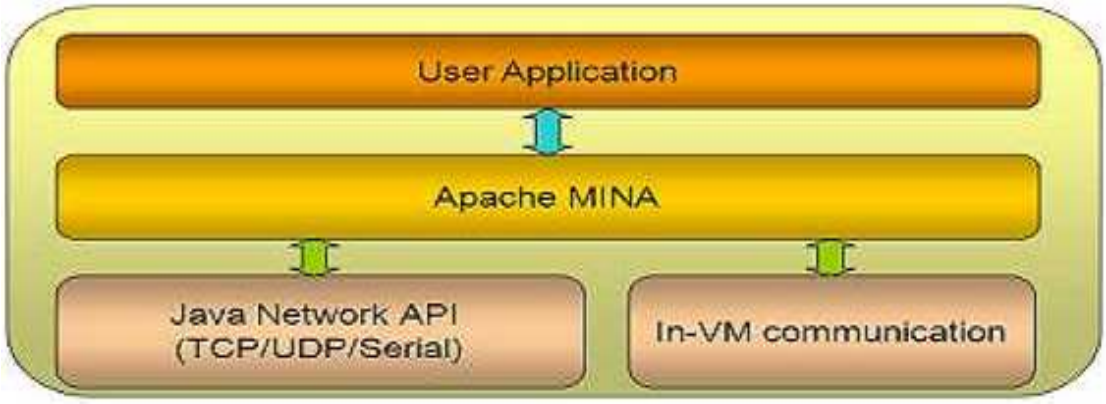

图 1 mina 架构图

mina 的执行流程如下：

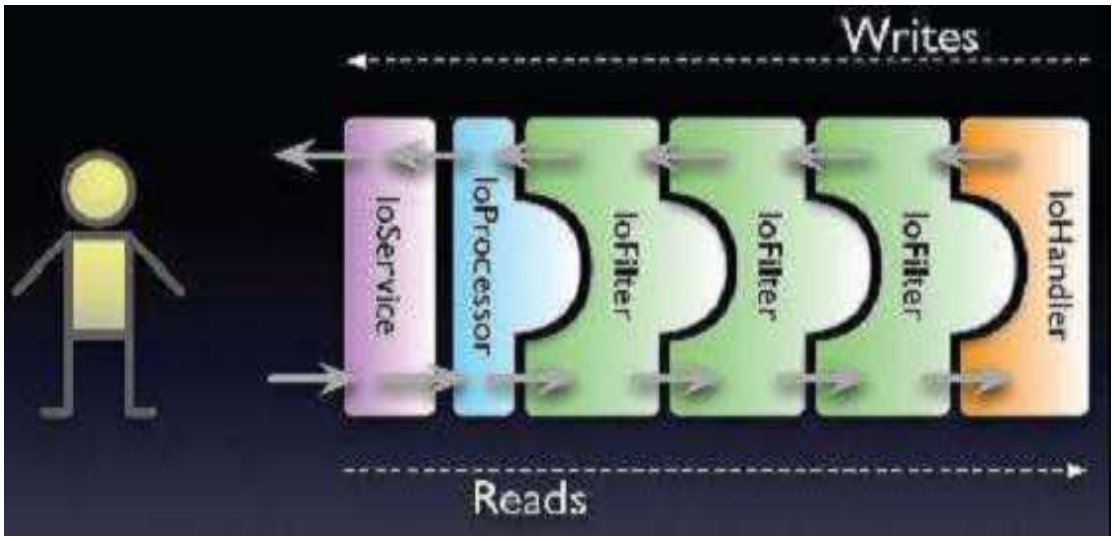

图 2 mina 工作流程图

其中 Reads 操作是指 server/clinet 端从网络上收到 Message 的过程；Writes 操作是指 server/clinet 端将 Message 写到网路上的过程。

(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。这也是实际开发过程中需要用户自己编写的部分代码。

3.基于 Mina 的应用程序的开发步骤

接上节，介绍基于 mina 开发的一般步骤：

- **简单的 TCPServer**

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。

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 {
    public static void main(String[] args){
        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。

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。

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 {
    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。

第一步：编写 IoService 并注册过滤器

```
public class MyClient {
    mainmainmainmain 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));
}
```

这里由于使用 mina 内置的过滤器，所以少了一步，如果自定义，加之。

第二步：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。

4.线程机制

Mina 中的很多执行环节都使用了多线程机制，用于提高性能，图 3 所示为 Mina 的线程模型图。

图 3 是 Mina 的 Server 端内部运行图，Client 处表示外部的客户端通过 Socket 建立连接。

图中 IoAcceptor 对应 NioSocketAcceptor 类，是用来接受 Socket 请求的。

图中用灰色的齿轮表示，小齿轮表示它一直可以干活，运转不息。黄色的小齿轮则表示一个运行在线程池上的任务，表示它是运转在线程池之上的。

1、服务端在创建 NioSocketAcceptor 实现时，会生成一个线程池，此线程池用来执行一个接受请求的任务，这个任务叫 Acceptor(可以在 AbstractPollingIoAcceptor 类中找到其实现类),Acceptor 会开一个 Selector，用来监听 NIO 中的 ACCEPT 事件。任务初始化时并没有执行，而在调用 NioSocketAcceptor 实例的 bind 方法时，则会启动对指定端口的 ACCEPT 事件的监听。

2、SimpleIoProcessorPool 是在 NioSocketAcceptor 实例化时创建的，其上有 N+1（N=CPU 的个数）个 NIOProcessor 来处理实际 IO 的读写事件，每个 NIOProcessor 都会对应一个 Selector(和 1 中的 Selector 一起构成了 Mina 独有的双 Selector 模型，这种设计的优点是不会导致阻塞)，来监听 Socket 中的读写事件。实际对读写的操作也是在一个 SimpleIoProcessorPool 实例化好的一个线程池中以任务的形式执行，这个任务叫 Processor(可以在 AbstractPollingIoProcessor 类中找到其实现)。

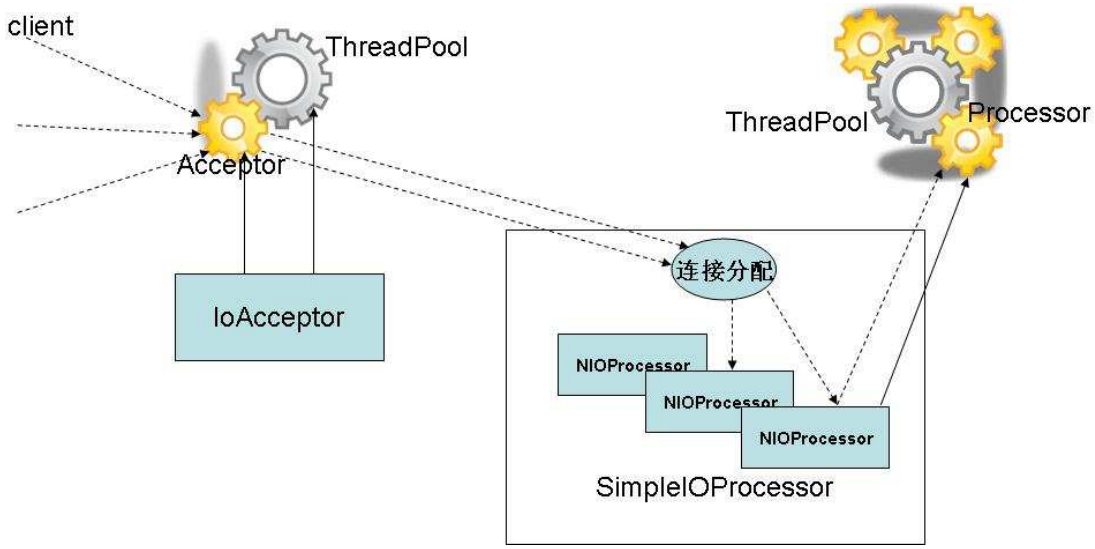

图 3 Mina 线程模型图

一次请求的过程如下：

Client 通过 Socket 连接服务器，先是由 Acceptor 接收到请求连接的事件（即 ACCEPT 事件）。此事件由 Acceptor 进行处理，会创建一条 Socket 连接，并将此连接和一个 NIOProcessor 关联，这个过程通过图中的连接分配器进行，连接分配器会均衡的将 Socket 和不同的 NIOProcessor 绑定(轮流分配)，绑定完成后，会在 NIOProcessor 上进行读写事件的监听，而读写的实际处理则分配给 Processor 任务完成。当有读写事件发生时，就会通知到对应的 Processor 进行数据处理。

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（无读写数据发生），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。

5.带线程机制的 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，监听套接字端口上的连接请求；

(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。

6.配置注意事项

(1) 将 jvm 的工作模式调整到 server 模式下，这个不是 mina 特有的配置，事实上所有的 java 服务器都应该这样配置。这个配置对服务器的性能影响是相当大的，有测试分析 server 模式下的方法调用速度几乎是 client 模式下的 10 倍左右。具体原因没查过，估计 jvm 有优化吧。

(2) protocol codec 编解码器和 Iohandler(CMS 中对应 HppIohandler)的效率，这是利用 mina 框架时通常需要手工编写的两处代码，因为每次 IoService 都需要进行这两个操作，所以，相对的优化对高并发的系统的效率提升是相当明显的。

(3)线程池大小的配置，这部分在**线程机制**一节有详细的介绍，实际开发中

应该通过调节以确定服务器性能。

(4)每个 IoSession 对应的 ReceiveBufferSize 和 ReceiveBufferSize 缓冲区的大小，这取决于通信传递的 Message 的大小。

(5) WriteTimeout 写超时时间的设置，这可以根据应用的并发率和每次请求的时间开销来调节，mina 默认的超时时间是 60s。

附 1　Codec 过滤器的编写示例

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

L: 21

Hello World!

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：定义协议对象：

```
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：实现编码器

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：实现解码器

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：实现编解码工厂

```
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：添加 codec 到 acceptor 的 FilterChain 中

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"))));
```

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!]

附 2　阅读部分源代码的心得

对于大多数人来说(没有一定的积淀的情况下)，试图通过源代码来掌握一门从未接触的框架或者新技术是非常困难的。个人觉得阅读源代码不在于读了多少，而在于你透过它看到的编程的一种思想和风格。举个简单的例子，抽象类和接口大家都会用吧，但是感觉我们用得似乎有点别扭，甚至有些时候完全是为了接口而接口，为了抽象类而抽象类。其实，通过优秀的源代码，可以发现外国人写的程序真的是很经典，能够让抽象类和接口出现在正确的地方。怎么才能让它们出现在正确的地方呢？这和设计模式是有很大的关系，设计模式(几乎所有优秀的开源框架基本都包含大量的设计模式)恰恰又是我们大多数人的薄弱环节，这样的情况下抽象类和接口的乱用也就变得顺理成章了。

有时我们会认为接口和抽象类是可以划等号的，我起初学 java 的时候也这样认为，唯一让我觉得接口存在的理由是 java 中类不能实现多继承，仅此而已；其实不然，关于接口和抽象类的区别，参考了网上的说法，总结起来：抽象类侧重于方法的复用，而接口更侧重于类的复用。抽象类实现的是一类事物公共的行为，而把个性的行为留给它的子类去实现，这样达到了对抽象类中公共行为(方法)的重用；关于接口，一般是对某个类赋予某种特性，比如一台计算机(看成一

个类)，我们给它装上电视卡(接口)、录音软件(接口)、学习软件(接口)等一系列功能，它相应的就有了电视机、录音机和学习机的功能，然而电脑还是那台电脑，类还是那个类，达到了类的复用。当然，这只是一个很简单的例子，但我们关注的应该是为什么要这么做。