远程通信 Mina2

学习笔记

作者：李少华
邮箱：xiaosanshaoli@126.com

QQ：305409913
2010-12-23　初稿

引言

最近使用 Mina 开发一个 Java 的 NIO 服务端程序，因此也特意学习了 Apache 的这个 Mina 框架。

首先，Mina 是个什么东西？看下官方网站（http://mina.apache.org/）对它的解释：

Apache 的 Mina（Multipurpose Infrastructure Networked Applications）是一个网络应用框架，可以帮助用户开发高性能和高扩展性的网络应用程序；它提供了一个抽象的、事件驱动的异步 API，使 Java NIO 在各种传输协议（如 TCP/IP，UDP/IP 协议等）下快速高效开发。

Apache Mina 也称为：

- NIO 框架
- 客户端/服务端框架（典型的 C/S 架构）
- 网络套接字（networking socket）类库

总之：我们简单理解它是一个封装底层 IO 操作，提供高级操作 API 的通讯框架！

（本文所有内容仅针对 Mina2.0 在 TCP/IP 协议下的应用开发）

一、 Mina 入门

先用 Mina 做一个简单的应用程序。

第一步. 下载使用的 Jar 包

a. 登录 http://mina.apache.org/downloads.html 下载 mina2.0.1.zip，解压获得 mina-core-2.0.0-M1.jar

b. 登录 http://www.slf4j.org/download.html 下载 slf4j1.5.2.zip，解压获得 slf4j-api-1.5.2.jar 与 slf4j-log4j12-1.5.2.jar

c. 添加 Log4j 的 jar 包，注意如果使用 slf4j-log4j12-XXX.jar，就需要添加 log4j1.2.X。我这里使用的是 log4j-1.2.14.jar （Logger 配置详情参见 http://mina.apache.org/first-steps.html ）

OK，4 个 jar 都完备了。

第二步. 工程创建配置

创建一个 Java Project 默认使用 UTF-8 编码格式），添加 log4j.properties

```
log4j.rootLogger=DEBUG,MINA,file


log4j.appender.MINA=org.apache.log4j.ConsoleAppender
```

```
log4j.appender.MINA.layout=org.apache.log4j.PatternLayout
log4j.appender.MINA.layout.ConversionPattern=%d{yyyy-MM-dd HH\:mm\:ss,SSS} %-5p
%c{1} %x - %m%n

log4j.appender.file=org.apache.log4j.RollingFileAppender
log4j.appender.file.File=./log/minademos.log
log4j.appender.file.MaxFileSize=5120KB
log4j.appender.file.MaxBackupIndex=10
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=[VAMS][%d] %p | %m | [%t] %C.%M(%L)%n
```

第三步.服务端程序

创建一个简单的服务端程序：（服务端绑定 3005 端口）

```
public class Demo1Server {
    private static Logger logger = Logger.getLogger(Demo1Server.class);

    private static int PORT = 3005;

    public static void main(String[] args) {
        IoAcceptor acceptor = null;
        try {
            // 创建一个非阻塞的server端的Socket
            acceptor = new NioSocketAcceptor();
            // 设置过滤器（使用Mina提供的文本换行符编解码器）
            acceptor.getFilterChain().addLast(
                    "codec",
                    new ProtocolCodecFilter(new TextLineCodecFactory(Charset
                            .forName("UTF-8"),
                            LineDelimiter.WINDOWS.getValue(),
                            LineDelimiter.WINDOWS.getValue())));
            // 设置读取数据的缓冲区大小
            acceptor.getSessionConfig().setReadBufferSize(2048);
            // 读写通道10秒内无操作进入空闲状态
            acceptor.getSessionConfig().setIdleTime(IdleStatus.BOTH_IDLE, 10);
            // 绑定逻辑处理器
            acceptor.setHandler(new Demo1ServerHandler());
            // 绑定端口
            acceptor.bind(new InetSocketAddress(PORT));
            logger.info("服务端启动成功...     端口号为：" + PORT);
        } catch (Exception e) {
            logger.error("服务端启动异常....", e);
            e.printStackTrace();
        }
```

```
    }
}
```

无需解释，大家看上面的注释也许就了解一二了；

注意：创建服务端最主要的就是绑定服务端的消息编码解码过滤器和业务逻辑处理器；

什么是编码与解码哪？大家知道，网络传输的数据都是二进制数据，而我们的程序不可能直接去操作二进制数据；这时候我们就需要来把接收到的字节数组转换为字符串，当然完全可以转换为任何一个 java 基本数据类型或对象，这就是解码！而编码恰好相反，就是把要传输的字符串转换为字节！

比如上面使用的 Mina 自带的根据文本换行符编解码的 TextLineCodec 过滤器------ 指定参数为根据 windows 的换行符编解码，遇到客户端发送来的消息，看到 windows 换行符（\r\n）就认为是一个消息了，而发送给客户端的消息，都会在消息末尾添加上\r\n 文本换行符；

业务逻辑处理器是 Demo1ServerHandler---看它的具体实现：

```
public class Demo1ServerHandler extends IoHandlerAdapter {
    public static Logger logger = Logger.getLogger(Demo1ServerHandler.class);

    @Override
    public void sessionCreated(IoSession session) throws Exception {
        logger.info("服务端与客户端创建连接...");
    }

    @Override
    public void sessionOpened(IoSession session) throws Exception {
        logger.info("服务端与客户端连接打开...");
    }

    @Override
    public void messageReceived(IoSession session, Object message)
            throws Exception {
        String msg = message.toString();
        logger.info("服务端接收到的数据为：" + msg);
        if ("bye".equals(msg)) { // 服务端断开连接的条件
            session.close();
        }
        Date date = new Date();
        session.write(date);
    }

    @Override
    public void messageSent(IoSession session, Object message) throws Exception
{
        logger.info("服务端发送信息成功...");
    }
```

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

    }

    @Override
    public void sessionIdle(IoSession session, IdleStatus status)
            throws Exception {
        logger.info("服务端进入空闲状态...");
    }

    @Override
    public void exceptionCaught(IoSession session, Throwable cause)
            throws Exception {
        logger.error("服务端发送异常...", cause);
    }
}
```

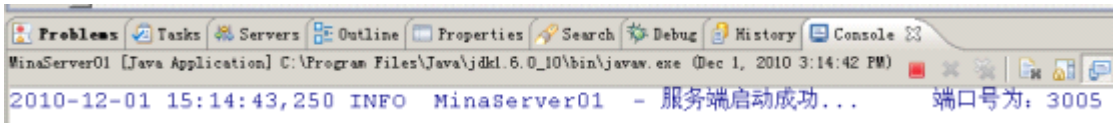
自定义的业务逻辑处理器继承了 IoHandlerAdapter 类，它默认覆盖了父类的 7 个方法，其实我们最关心最常用的只有一个方法：messageReceived() ---- 服务端接收到一个消息后进行业务处理的方法；看代码：

```
@Override
    public void messageReceived(IoSession session, Object message)
            throws Exception {
        String msg = message.toString();
        logger.info("服务端接收到的数据为：" + msg);
        if ("bye".equals(msg)) { // 服务端断开连接的条件
            session.close();
        }
        Date date = new Date();
        session.write(date);
    }
```

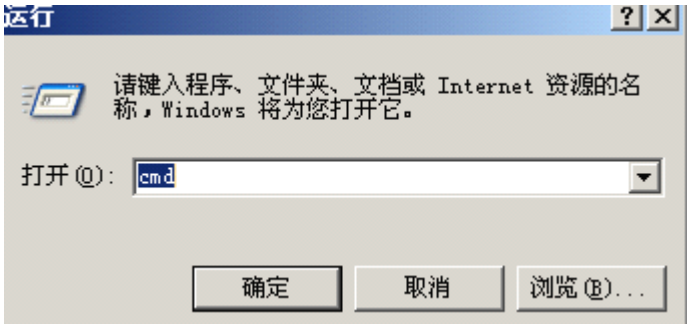
接受到客户端信息，并返回给客户端一个日期；如果客户端传递的消息为“bye”，就是客户端告诉服务端，可以终止通话了，关闭与客户端的连接。

使用命令行的 telnet 来测试下服务端程序！

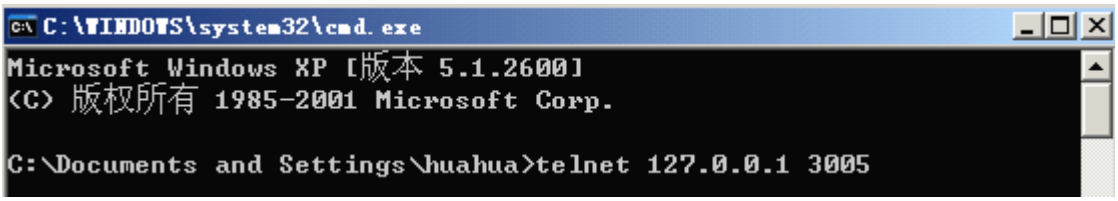
a. 启动服务端程序；



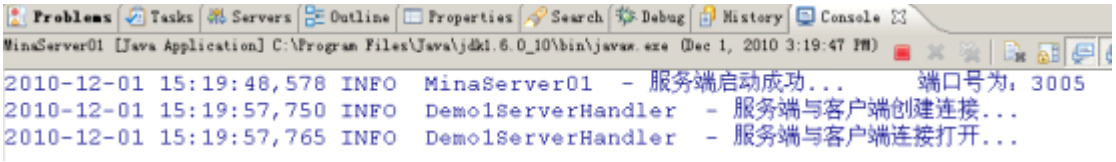
b. Windows 下开始菜单，运行，输入 cmd，回车；



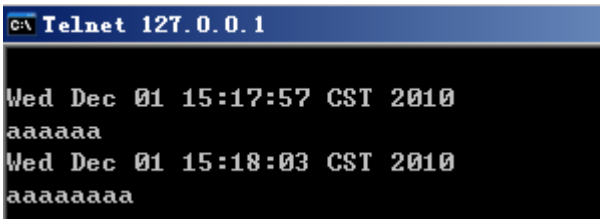
c. 输入：telnet 127.0.0.1 3005　回车



d. 连接成功后，服务端程序的后台会打印如下信息：这个就是业务逻辑逻辑处理器打印的：

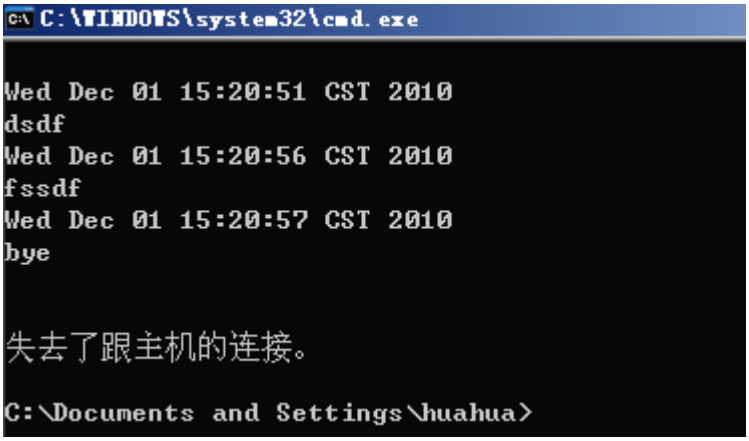


e. telnet 中随便输入一个字符串，回车；则可以看到返回的日期；



f. 输入 bye，回车，提示服务端断开连接

如果需要重新测试，则需要再次重复 C 的步骤；



注意：

不要输入中文字符，windows 不会把中文字符用 utf-8 编码再发送给服务端的；

第四步. 客户端程序

Mina 能做服务端程序，自然也可以做客户端程度啦。最重要的是，客户端程序和服务端程序写法基本一致，很简单的。

客户端代码：

```
public class MinaClient01 {
    private static Logger logger = Logger.getLogger(MinaClient01.class);

    private static String HOST = "127.0.0.1";

    private static int PORT = 3005;

    public static void main(String[] args) {
        // 创建一个非阻塞的客户端程序
        IoConnector connector = new NioSocketConnector();
        // 设置链接超时时间
        connector.setConnectTimeout(30000);
        // 添加过滤器
        connector.getFilterChain().addLast(
                "codec",
                new ProtocolCodecFilter(new TextLineCodecFactory(Charset
                        .forName("UTF-8"), LineDelimiter.WINDOWS.getValue(),
                        LineDelimiter.WINDOWS.getValue())));
        // 添加业务逻辑处理器类
        connector.setHandler(new Demo1ClientHandler());
        IoSession session = null;
        try {
            ConnectFuture future = connector.connect(new InetSocketAddress(
                    HOST, PORT));// 创建连接
            future.awaitUninterruptibly();// 等待连接创建完成
            session = future.getSession();// 获得session
            session.write("我爱你mina");// 发送消息
        } catch (Exception e) {
            logger.error("客户端链接异常...", e);
        }

        session.getCloseFuture().awaitUninterruptibly();// 等待连接断开
        connector.dispose();
    }
}
```

和服务端代码极其相似，不同的是服务端是创建 NioSocketAcceptor 对象，而客户端是创建 NioSocketConnector 对象；同样需要添加编码解码过滤器和业务逻辑过滤器；

业务逻辑过滤器代码：

```
public class Demo1ClientHandler extends IoHandlerAdapter {
    private static Logger logger = Logger.getLogger(Demo1ClientHandler.class);
```

```
    @Override
    public void messageReceived(IoSession session, Object message)
            throws Exception {
        String msg = message.toString();
        logger.info("客户端接收到的信息为：" + msg);
    }

    @Override
    public void exceptionCaught(IoSession session, Throwable cause)
            throws Exception {
        logger.error("客户端发生异常...", cause);
    }
}
```

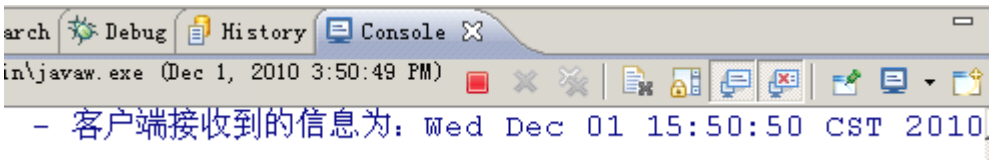
它和服务端的业务逻辑处理类一样，继承了 IoHandlerAdapter 类，因此同样可以覆盖父类的 7 个方法，同样最关心 messageReceived 方法，这里的处理是接收打印了服务端返回的信息；另一个覆盖的方法是异常信息捕获的方法；

测试服务端与客户端程序！

a. 启动服务端，然后再启动客户端（客户端发送的消息是“我爱你 mina”）

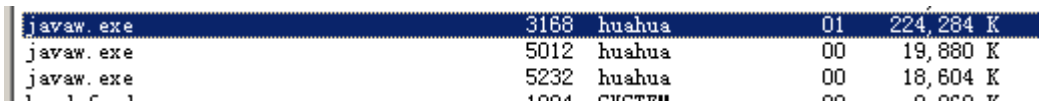
b. 服务端接收消息并处理成功；

客户端接收响应结果



第五步. 长连接 VS 短连接

此时，查看 windows 的任务管理器，会发现：当前操作系统中启动了 3 个 java 进程（注意其中一个进程是 myEclipse 的）。



我们知道，java 应用程序的入口是 main()方法，启动一个 main()方法相当于开始运行一个 java 应用程序，此时会运行一个 Java 虚拟机，操作系统中会启动一个进程，就是刚刚看到的“javaw.exe”。也就是每启动一个 Java 应用程序就是多出一个 Java 进程。因为启动了 Mina 服务端和客户端 2 个服务端程序，所有其他 2 个进程的出现。

测试一下：再次启动一个客户端程序，查看任务管理器，会发现进程又多出

一个，这个是刚刚启动的客户端进程，它和前一个客户端进程一直存在。这就是一个典型的长连接。

javaw.exe	460	huahua	00	18,368 K
javaw.exe	3168	huahua	00	224,400 K
javaw.exe	5012	huahua	00	20,824 K
javaw.exe	5232	huahua	00	18,756 K

长连接的现象在网络中非常普遍,比如我们的 QQ 客户端程序，登录成功后与腾讯的服务器建立的就是长连接；除非主动关闭掉 QQ 客户端，或者是 QQ 服务端挂了，才会断开连接；看我们的服务端程序，就有关闭连接的条件：如果客户端发送信息“bye”，服务端就会主动断开连接！

```
@Override
    public void messageReceived(IoSession session, Object message)
            throws Exception {
        String msg = message.toString();
        logger.info("服务端接收到的数据为：" + msg);
        if ("bye".equals(msg)) { // 服务端断开连接的条件
            session.close();
        }
        Date date = new Date();
        session.write(date);
    }
```

与长连接相对应的是短连接，比如常说的请求/响应模式（HTTP 协议就是典型的请求/响应模式）-----客户端向服务端发送一个请求，建立连接后，服务端处理并响应成功，此时就主动断开连接了！

短连接是一个简单而有效的处理方式，也是应用最广的。问题是哪一方先断开连接呢？可以在服务端，也可以在客户端，但是提倡在服务端主动断开；

Mina 的服务端业务逻辑处理类中有一个方法 messageSent，他是在服务端发送信息成功后调用的：

```
@Override
    public void messageSent(IoSession session, Object message) throws Exception
{
        logger.info("服务端发送信息成功...");
    }
```

修改后为

```
@Override
    public void messageSent(IoSession session, Object message) throws Exception
{
        session.close(); //发送成功后主动断开与客户端的连接
        logger.info("服务端发送信息成功...");
    }
```

这时候客户端与服务端就是典型的短连接了；再次测试，会发现客户端发送请求，接收成功后就自动关闭了，进程只剩下服务端了！

到此为止，我们已经可以运行一个完整的基于 TCP/IP 协议的应用程序啦！

总结：

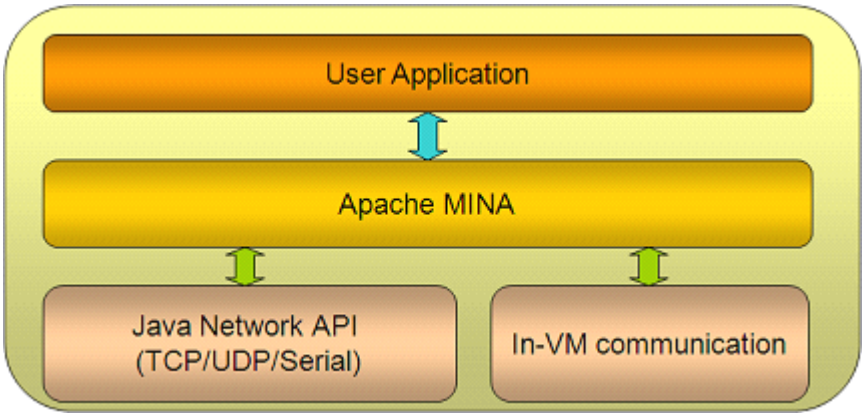
服务端程序或客户端程序创建过程：

创建连接-->添加消息过滤器(编码解码等)-->添加业务处理

二、 Mina 基础

Mina 使用起来多么简洁方便呀，就是不具备 Java NIO 的基础，只要了解了 Mina 常用的 API，就可以灵活使用并完成应用开发。

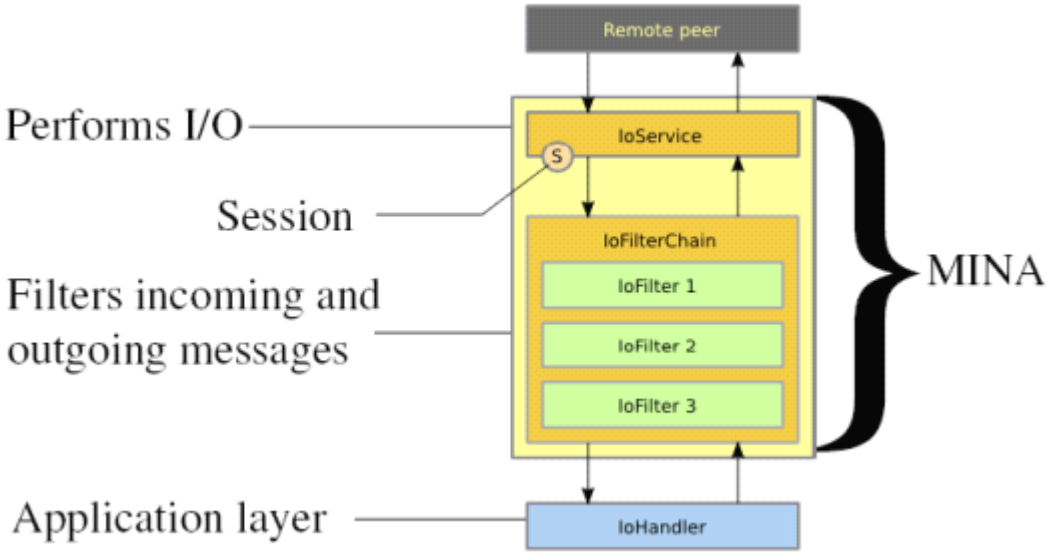
首先，看 Mina 在项目中所处的位置，如下图：



Mina 处于中间层，它不关心底层网络数据如何传输，只负责接收底层数据，过滤并转换为 Java 对象提供给我们的应用程序，然后把应用程序响应值过滤并转换为底层识别的字节，提供给底层传输；

------总之：Mina 是底层数据传输和用户应用程序交互的接口！

Mina 工作流程图如下：



这个流程图不仅很直观的看出了 Mina 的工作流程，也涵盖了 Mina 的三个核心接口:IoService 接口，IoFilter 接口和 IoHandler 接口：

- 第一步. 创建服务对象（客户端或服务端） ---IoService 接口实现
- 第二步. 数据过滤（编码解码等） ---IOFilter 接口实现

- 第三步. 业务处理 ---IoHandler 接口实现

Mina 的精髓是 IOFilter，它可以进行日志记录，信息过滤，编码解码等操作，把数据接收发送从业务层独立出来。

而创建服务对象，则是把 NIO 繁琐的部分进行封装，提供简洁的接口。

业务处理是我们最关心的部分，跟普通的应用程序没任何分别。

1. IoService 接口

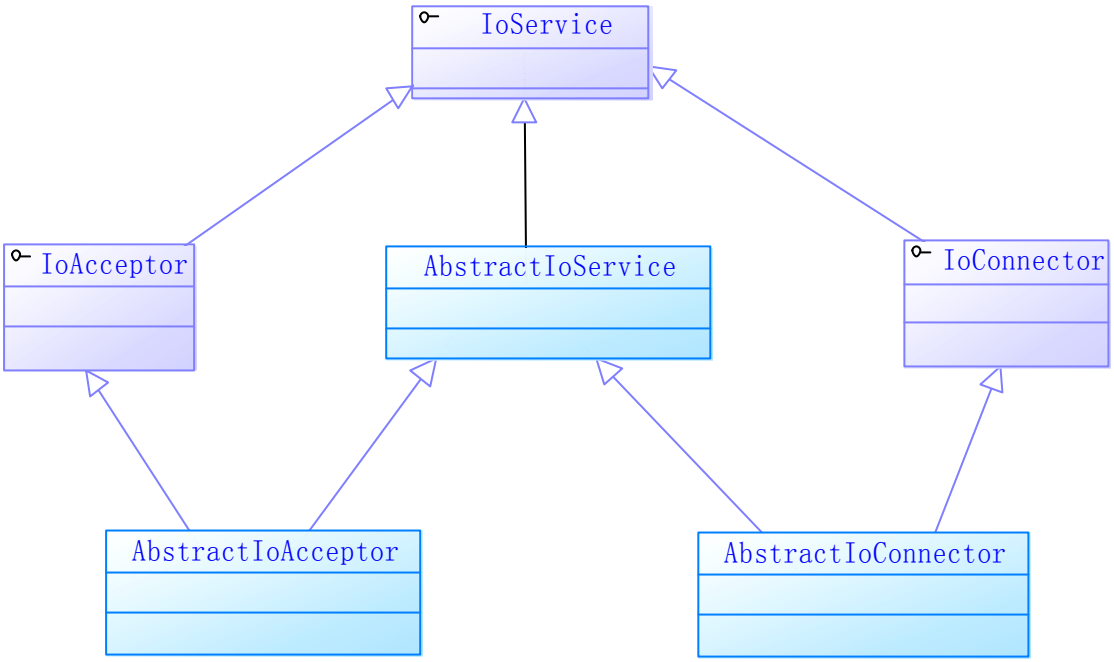
作用：**IoService** 是创建服务的顶层接口，无论客户端还是服务端，都是从它继承实现的。

2.1.1 类结构

常用接口为：**IoService，IoAcceptor，IoConnector**

常用类为：**NioSocketAcceptor，NioSocketConnector**

类图如下：



先提出两个问题：

1. 为什么有了 **IoService** 接口还要定义 **AbstractIoService** 抽象类？
2. **AbstractIoService** 抽象类与 **IoAcceptor(IoConnector)**有什么区别？

分析：

- **IoService** 接口声明了服务端的共有属性和行为；
- **IoAcceptor** 接口继承了 IoService 接口，并添加了服务端特有的接口属性及方法，比如 bind（）方法，成为典型的服务端接口；
- **IoConnector** 接口同样继承了 IoService 接口，并添加了客户端特有的接口属性及方法，比如 connect（）方法，成为典型的客户端接口；

---- IoService 是 IoAcceptor 和 IoConnector 父接口，为什么不直接定义 IoAcceptor 和 IoConnector 接口呢，因为它们有共同的特点，比如共同属性，管理服务的方法等，所有 IoService 的出现是为了代码复用。

- **AbstractIoService** 实现了 IoService 中管理服务的方法，比如 getFilterChainBuilder 方法---获得过滤器链；

----为什么有了 **IoService** 接口还要定义 **AbstractIoService** 抽象类？一样

为了代码的复用！AbstractIoService 抽象类实现了服务端或客户端的共有的管理服务的方法，不需要让 **IoService** 接口的子类重复的实现这些方法；

- **AbstractIoService** 抽象类继承了 AbstractIoService 抽象类并实现了 IoAcceptor 接口，成为了拥有管理服务端实现功能的服务端类；我们常用的 NioSocketAcceptor 就是它的子类；
- **AbstractIoConnector** 抽象类继承了 AbstractIoService 抽象类并实现了 IoConnector 接口，成为了拥有管理客户端实现功能的客户端类；我们常用的 NioSocketConnector 就是它的子类；

----AbstractIoService 抽象类与 IoAcceptor(IoConnector)有什么区别？很清楚，AbstractIoService 抽象类实现的是共有的管理服务的方法，只有管理功能的一个类；而两个接口却是不同的两个服务角色----一个客户端，一个服务端。

2.1.2 应用

在实际应用中，创建服务端和客户端的代码很简单：

创建服务端：

```
IoAcceptor acceptor = null;
        try {
            // 创建一个非阻塞的server端的Socket
            acceptor = new NioSocketAcceptor();
```

创建客户端：

```
// 创建一个非阻塞的客户端程序
        IoConnector connector = new NioSocketConnector();
```

而我们常常关心的就是服务端和客户端的一些参数信息：

1.IoSessionConfig getSessionConfig()

获得 IoSession 的配置对象 IoSessionConfig，通过它可以设置 Socket 连接的一些选项。

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。

```
    // 获得IoSessionConfig对象
    IoSessionConfig cfg=acceptor.getSessionConfig();
    // 设置读取数据的缓冲区大小（）
    cfg.setReadBufferSize(2048);
    // 读写通道10秒内无操作进入空闲状态
```

```
    cfg.setIdleTime(IdleStatus.BOTH_IDLE, 10);
    // 写操作超时时间10秒
    cfg.setWriteTimeout(10);
```

2.DefaultIoFilterChainBuilder getFilterChain()

获得过滤器链，由此来配置过滤器；非常核心的一个配置！

```
// 创建一个非阻塞的server端的Socket
acceptor = new NioSocketAcceptor();
// 设置日志过滤器
acceptor.getFilterChain().addLast(
        "logger",
        new LoggingFilter());
// 设置过滤器（使用Mina提供的文本换行符编解码器）
acceptor.getFilterChain().addLast(
        "codec",
        new ProtocolCodecFilter(new TextLineCodecFactory(Charset
            .forName("UTF-8"),
            LineDelimiter.WINDOWS.getValue(),
            LineDelimiter.WINDOWS.getValue())));
```

3.setHandler(IoHandler handler);

向 IoService 注册 IoHandler 进行业务处理。这是服务（无聊客户端还是服务端）必不可少的配置；

```
// 添加业务逻辑处理器类
connector.setHandler(new Demo1ClientHandler());
```

4.其他配置

服务端必须指定绑定的端口号：

```
// 绑定端口
acceptor.bind(new InetSocketAddress(PORT));
logger.info("服务端启动成功...     端口号为：" + PORT);
```

客户端必须指定请求的服务器地址和端口号：（该方法是异步执行的）

```
ConnectFuture future = connector.connect(new InetSocketAddress(
                    HOST, PORT));// 创建连接
future.awaitUninterruptibly();// 等待连接创建完成
session = future.getSession();// 获得session
session.write("我爱你mina");// 发送消息
```

5.关闭客户端

因为客户端的连接是异步的，所有必须先连接上服务端获得了 session 才能通信；同时，一旦需要关闭，必须指定 disponse()方法关闭客户端，如下：

```
// 添加业务逻辑处理器类
        connector.setHandler(new Demo1ClientHandler());
        IoSession session = null;
```

```
        try {
            ConnectFuture future = connector.connect(new InetSocketAddress(
                    HOST, PORT));// 创建连接
            future.awaitUninterruptibly();// 等待连接创建完成
            session = future.getSession();// 获得session
            session.write("我爱你mina");// 发送消息
        } catch (Exception e) {
            logger.error("客户端链接异常...", e);
        }

        session.getCloseFuture().awaitUninterruptibly();// 等待连接断开
        connector.dispose();
```

这是 Mina2 的处理方式，但在 Mina1.1.7 中，必须使用 setWorkerTimeout() 方法关闭客户端：

```
// 在关闭客户端前进入空闲状态的时间为1秒
//Set how many seconds the connection worker thread should remain alive once idle
  before terminating itself.
connector.setWorkerTimeout(1);
```

2.IoFilter 接口

Mina 最主要的工作就是把底层传输的字节码转换为 Java 对象，提供给应用程序；或者把应用程序返回的结果转换为字节码，交给底层传输。这些都是由 IoFilter 完成的，因此 IoFilter 是 Mina 的精髓所在。

在 Mina 程序中，IoFilter 是必不可少的；有了它，Mina 的层次结构才异常清晰：

```
    IoFilter    ----    消息过滤
    IoHandler   ----    业务处理
```

Filter，过滤器的意思。IoFilter，I/O操作的过滤器。IoFilter和Servlet中的过滤器一样，主要用于拦截和过滤网络传输中I/O操作的各种消息。在Mina的官方文档中已经提到了IoFilter 的作用：

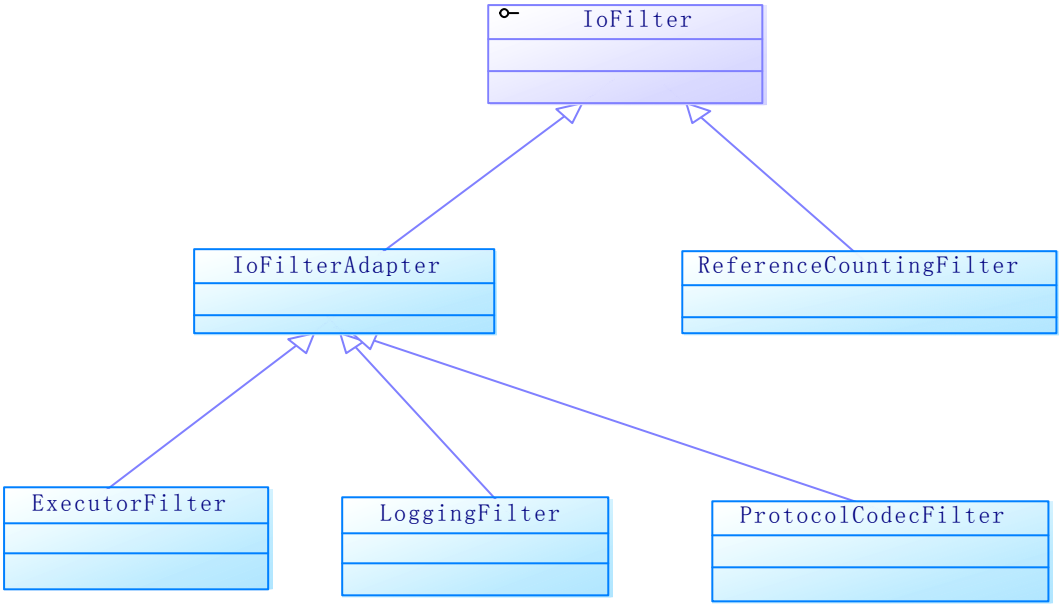
（1）记录事件的日志（Mina默认提供了LoggingFilter）
（2）测量系统性能
（3）信息验证
（4）过载控制
（5）信息的转换(主要就是编码和解码)
（6）和其他更多的信息

IoService 实例会绑定一个 DefaultIoFilterChainBuilder ---- 过滤器链，我们把自定义的各种过滤器（IoFilter）自由的插放在这个过滤器链上了，类似于一种可插拔的功能！

2.2.1 类结构

常用接口为：**IoFilter，IoFilterChainBuilder**
常用类为：**IoFilterAdapter，DefaultIoFilterChainBuilder**
ProtocolCodecFilter，LoggingFilter

类图如下：



同上面，先提出两个问题：

1. 在 IoService 中如何添加多个 IoFilter?
2. 如何自定义协议编解码器

分析：

a. IoFilter 有 2 个实现类：IoFilterAdapter 是个抽象的适配器类，我们可以根据需要扩展这个类，并且有选择的覆盖过滤器的方法；所有方法的默认把事件转发到下一个过滤器；查看源码如下：

```
public void sessionOpened(NextFilter nextFilter, IoSession session)throws
      Exception {
        nextFilter.sessionOpened(session);
  }
```

b .ReferenceCountingFilter 封装了 IoFilter 实例，监看调用该 filter 的对象的个数，如果没有任何对象调用该 IoFilter，就自动销毁 IoFilter；查看源码如下：

```
public class ReferenceCountingFilter implements IoFilter {
    private final IoFilter filter;

    private int count = 0;

    public ReferenceCountingFilter(IoFilter filter) {
        this.filter = filter;
    }

    public void init() throws Exception {
        // no-op, will init on-demand in pre-add if count == 0
    }
    public void destroy() throws Exception {
          } ……略
```

c. 实现 IoFilterAdapter 的类有多个，但是我们使用最多的就是

ProtocolCodecFilter----它是我们自定义编解码器的入口。

2.2.2 应用

我们在应用中解释上面提述的两个问题！

添加过滤器

----在 IoService 中如何添加多个 IoFilter？如何代码，我添加了 2 个过滤器：LoggingFilter 和 TextLineCodecFactory（源码为入门的服务端程序）

```
        // 创建一个非阻塞的server端的Socket
        acceptor = new NioSocketAcceptor();
        // 设置日志过滤器
        acceptor.getFilterChain().addLast("logger",new LoggingFilter());
        // 设置过滤器（使用Mina提供的文本换行符编解码器）
        acceptor.getFilterChain().addLast(
                "codec",
                new ProtocolCodecFilter(new TextLineCodecFactory(Charset
                        .forName("UTF-8"),
                        LineDelimiter.WINDOWS.getValue(),
                        LineDelimiter.WINDOWS.getValue())));
        // 获得IoSessionConfig对象
        IoSessionConfig cfg = acceptor.getSessionConfig();
        // 读写通道10秒内无操作进入空闲状态
        cfg.setIdleTime(IdleStatus.BOTH_IDLE, 10);

        // 绑定逻辑处理器
        acceptor.setHandler(new Demo1ServerHandler());
        // 绑定端口
        acceptor.bind(new InetSocketAddress(PORT));
        logger.info("服务端启动成功...     端口号为：" + PORT);
```

运行主程序：

执行 telnet 127.0.0.1 3005，输入 a,回车，后台打印信息如下：

```
2010-12-16 16:39:27,937 INFO  TestServer01  - 服务端启动成功...     端口号为：
3005
2010-12-16 16:39:31,328 INFO  LoggingFilter  - CREATED
2010-12-16 16:39:31,328 INFO  Demo1ServerHandler  - 服务端与客户端创建连接...
2010-12-16 16:39:31,328 INFO  LoggingFilter  - OPENED
2010-12-16 16:39:31,328 INFO  Demo1ServerHandler  - 服务端与客户端连接打开...
2010-12-16 16:39:32,296 INFO  LoggingFilter  - RECEIVED: HeapBuffer[pos=0 lim=1
cap=2048: 61]
2010-12-16 16:39:32,718 INFO  LoggingFilter  - RECEIVED: HeapBuffer[pos=0 lim=2
cap=2048: 0D 0A]
2010-12-16 16:39:32,734 INFO  Demo1ServerHandler  - 服务端接收到的数据为：a
2010-12-16 16:39:32,750 INFO  LoggingFilter  - SENT: HeapBuffer[pos=0 lim=30
cap=31: 54 68 75 20 44 65 63 20 31 36 20 31 36 3A 33 39...]
2010-12-16 16:39:32,750 INFO  LoggingFilter  - SENT: HeapBuffer[pos=0 lim=0 cap=0:
```

```
empty]
2010-12-16 16:39:32,750 INFO  Demo1ServerHandler  - 服务端发送信息成功...
2010-12-16 16:39:32,750 INFO  LoggingFilter  - CLOSED
```

注意：LoggerFilter 的日志（红色部分）

修改代码，交换 LoggingFilter 和 TextLineCodecFactory 的位置，如下所示：

```
            // 设置过滤器（使用Mina提供的文本换行符编解码器）
            acceptor.getFilterChain().addLast(
                    "codec",
                    new ProtocolCodecFilter(new TextLineCodecFactory(Charset
                            .forName("UTF-8"),
                            LineDelimiter.WINDOWS.getValue(),
                            LineDelimiter.WINDOWS.getValue())));
            // 设置日志过滤器
            acceptor.getFilterChain().addLast("logger",new LoggingFilter());
```

启动服务端，执行 telnet 127.0.0.1 3005，后台打印信息如下：

```
2010-12-16 16:41:36,125 INFO  TestServer01  - 服务端启动成功...     端口号为：
3005
2010-12-16 16:41:38,296 INFO  LoggingFilter  - CREATED
2010-12-16 16:41:38,296 INFO  Demo1ServerHandler  - 服务端与客户端创建连接...
2010-12-16 16:41:38,296 INFO  LoggingFilter  - OPENED
2010-12-16 16:41:38,296 INFO  Demo1ServerHandler  - 服务端与客户端连接打开...
2010-12-16 16:41:39,296 INFO  LoggingFilter  - RECEIVED: a
2010-12-16 16:41:39,296 INFO  Demo1ServerHandler  - 服务端接收到的数据为：a
2010-12-16 16:41:39,328 INFO  LoggingFilter  - SENT: Thu Dec 16 16:41:39 CST 2010
2010-12-16 16:41:39,328 INFO  Demo1ServerHandler  - 服务端发送信息成功...
2010-12-16 16:41:39,328 INFO  LoggingFilter  - CLOSED
```

对比上下日志，会发现，如果 LoggingFilter 在编码器前，它会在编码器处理前打印请求值和返回值的二进制信息，在编码器之后就不会打印！

在 FilterChain 中都是 addLast()的方式添加在过滤链的最后面，这时候，把那个过滤器放在前面，就会先执行那个过滤器！

同 addLast（）方法一样，还提供了 addFirst（），addBefore（）等方法供使用。此时，就不难知道如何添加过滤器了吧！它们的顺序如何，就看你的设置的位置了！

同时发现，日志过滤器是根据IoSession的状态(创建、开启、发送、接收、异常等等)来记录会话的事件信息的！这对我们跟踪IoSession很有用。当地，也可以自定义logger的日志级别，定义记录那些状态的日志。比如：

```
// 设置日志过滤器
LoggingFilter lf=new LoggingFilter();
lf.setMessageReceivedLogLevel(LogLevel.DEBUG);
acceptor.getFilterChain().addLast("logger",lf);
```

自定义编解码器

----如何自定义协议编解码器？

协议编解码器是在使用 Mina 的时候最需要关注的对象，因为网络传输的数

据都是二进制数据（byte），而在程序中面向的是 JAVA 对象，这就需要在发送数据时将 JAVA 对象编码二进制数据，接收数据时将二进制数据解码为 JAVA 对象。

编解码器同样是以过滤器的形式安插在过滤器链上，如下所示：

```
// 设置过滤器（使用Mina提供的文本换行符编解码器）
acceptor.getFilterChain().addLast(
        "codec",
        new ProtocolCodecFilter(new TextLineCodecFactory(Charset
                .forName("UTF-8"),
                LineDelimiter.WINDOWS.getValue(),
                LineDelimiter.WINDOWS.getValue())));
```

协议编解码器是通过 ProtocolCodecFilter 过滤器构造的，看它的构造方法，它需要一个 ProtocolCodecFactory 对象：

```
public ProtocolCodecFilter(ProtocolCodecFactory factory) {
        if (factory == null) {
            throw new NullPointerException("factory");
        }
        this.factory = factory;
}
```

ProtocolCodecFactory 接口非常直接，通过 ProtocolEncoder 和 ProtocolDecoder 对象来构建！

```
public interface ProtocolCodecFactory {
    /**
     * Returns a new (or reusable) instance of {@link ProtocolEncoder} which
     * encodes message objects into binary or protocol-specific data.
     */
    ProtocolEncoder getEncoder(IoSession session) throws Exception;

    /**
     * Returns a new (or reusable) instance of {@link ProtocolDecoder} which
     * decodes binary or protocol-specific data into message objects.
     */
    ProtocolDecoder getDecoder(IoSession session) throws Exception;
}
```

ProtocolEncoder 和 ProtocolDecoder 接口是 Mina 负责编码和解码的顶级接口！

编码和解码的前提就是协议的制定：比如上面我们使用的 Mina 自带的根据文本换行符解码的 TextLineCodecFactory(), 如果遇到文本换行符就开始编解码！

为什么要制定协议呢？常用的协议制定方法有哪些？

我们知道，底层传输的都是二进制数据，服务端和客户端建立连接后进行数据的交互，接受这对方发送来的消息，如何判定发送的请求或者响应的数据结束了呢？总不能一直傻等着，或者随意的就结束消息接收吧。这就需要一个规则！比如 QQ 聊天工具，当输入完一个消息后，点击发送按钮向对方发送时，此时系

统就会在在你的消息后添加一个文本换行符，接收方看到这个文本换行符就认为这是一个完整的消息，解析成字符串显示出来。而这个规则，就称之为协议！

制定协议的方法：

- **定长消息法**：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。
- **字符定界法**：这种方式是使用特殊字符作为数据的结束符，一般适用于简单数据的发送。譬如：在消息的结尾自动加上文本换行符（Windows 使用\r\n，Linux 使用\n），接收方见到文本换行符就认为是一个完整的消息，结束接收数据开始解析。注意：这个标识结束的特殊字符一定要简单，常常使用 ASCII 码中的特殊字符来标识。
- **定长报文头法**：使用定长报文头，在报文头的某个域指明报文长度。该方法最灵活，使用最广。譬如：协议为 - 协议编号（1 字节）+数据长度（4 个字节）+真实数据。请求到达后，解析协议编号和数据长度，根据数据长度来判断后面的真实数据是否接收完整。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到底读到多长的字节数就不用再读取数据了。

根据协议，把二进制数据转换成 Java 对象称为解码（也叫做拆包）；把 Java 对象转换为二进制数据称为编码（也叫做打包）；

我们这里重点讲解下后面两个协议的具体使用！

IoBuffer 常用方法：

Mina 中传输的所有二进制信息都存放在 IoBuffer 中，IoBuffer 是对 Java NIO 中 ByteBuffer 的封装（Mina2.0 以前版本这个接口也是 ByteBuffer），提供了更多操作二进制数据，对象的方法，并且存储空间可以自增长，用起来非常方便；简单理解，它就是个可变长度的 byte 数组！

1. static IoBuffer allocate(int capacity,boolean useDirectBuffer)

创建 IoBuffer 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。

2. 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。

3. IoBuffer flip()

limit=position ， position=0,重置 mask，为了读取做好准备，一般是结束 buf 操作，将 buf 写入输出流时调用；这个必须要调用，否则极有可能 position!=limit，导致 position 后面没有数据；每次写入数据到输出流时，必须确保 position=limit。

4. IoBuffer clear()与IoBuffer reset()

clear：limit=capacity ， position=0,重置mark；它是不清空数据，但从头开始存放数据做准备---相当于覆盖老数据。

reset就是清空数据

5. int remaining()与boolean hasRemaining()

这两个方法一般是在调用了flip()后使用的，remaining()是返回limt-position的值！hasRemaining()则是判断当前是否有数据，返回position < limit的boolean值！

Demo1：模拟根据文本换行符编解码

第一步：编写解码器

实现 ProtocolDecoder 接口，覆盖 decode()方法；

```
import java.nio.charset.Charset;

import org.apache.mina.common.IoBuffer;
import org.apache.mina.common.IoSession;
import org.apache.mina.filter.codec.ProtocolDecoder;
import org.apache.mina.filter.codec.ProtocolDecoderOutput;

public class MyTextLineCodecDecoder implements ProtocolDecoder {
    private Charset charset = Charset.forName("UTF-8");

    IoBuffer buf = IoBuffer.allocate(100).setAutoExpand(true);

    public void decode(IoSession session, IoBuffer in, ProtocolDecoderOutput out)
            throws Exception {
        while (in.hasRemaining()) {
            byte b = in.get();
            buf.put(b);
            if (b == '\n') {
                buf.flip();
                byte[] msg = new byte[buf.limit()];
                buf.get(msg);
                String message = new String(msg, charset);
                //解码成功，把buf重置
                buf = IoBuffer.allocate(100).setAutoExpand(true);
                out.write(message);
            }
        }

    }

    public void dispose(IoSession session) throws Exception {
    }

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

方法解释：

```
public void decode(IoSession session, IoBuffer in, ProtocolDecoderOutput out)
        throws Exception {
```

decode 方法的参数 IoBuffer 是建立连接后接收数据的字节数组；我们不断的从它里面读数据，直到遇上\r\n 就停止读取数据，把上面累加的所有数据转换为一个字符串，输出！

第二步：编写编码器：

实现 ProtocolEncoder 接口，覆盖 encode()方法；

```
import java.nio.charset.Charset;
import java.nio.charset.CharsetEncoder;

import org.apache.mina.common.IoBuffer;
import org.apache.mina.common.IoSession;
import org.apache.mina.filter.codec.ProtocolEncoder;
import org.apache.mina.filter.codec.ProtocolEncoderOutput;

public class MyTextLineCodecEncoder implements ProtocolEncoder {
    private Charset charset = Charset.forName("UTF-8");

    public void encode(IoSession session, Object message,
            ProtocolEncoderOutput out) throws Exception {
        IoBuffer buf = IoBuffer.allocate(100).setAutoExpand(true);
        CharsetEncoder ce = charset.newEncoder();
        buf.putString(message.toString(), ce);
        // buf.put(message.toString().getBytes(charset));
        buf.put((byte) '\r');
        buf.put((byte) '\n');
        buf.flip();
        out.write(buf);
    }

    public void dispose(IoSession session) throws Exception {

    }
}
```

实现很简单，把要编码的数据放进一个 IoBuffer 中，并在 IoBuffer 结尾添加\r\n,输出。

第三步：编辑编解码器工厂类

实现 ProtocolCodecFactory 接口，覆盖其 getDecoder()和 getEncoder()方法；

```
import org.apache.mina.common.IoSession;
import org.apache.mina.filter.codec.ProtocolCodecFactory;
import org.apache.mina.filter.codec.ProtocolDecoder;
import org.apache.mina.filter.codec.ProtocolEncoder;
```

```
public class MyTextLineCodecFactory implements ProtocolCodecFactory {

    public ProtocolDecoder getDecoder(IoSession session) throws Exception {
        return new MyTextLineCodecDecoder();
    }

    public ProtocolEncoder getEncoder(IoSession session) throws Exception {
        return new MyTextLineCodecEncoder();
    }
}
```

第四步：测试

到现在，一个简单的根据\r\n 换行符编解码的过滤器实现了；添加到一个服务端中，测试：

```
// 设置过滤器（使用Mina提供的文本换行符编解码器）
acceptor.getFilterChain().addLast(
        "codec",
         new ProtocolCodecFilter(new MyTextLineCodecFactory()));
```

启动服务端，用 telnet 测试一把，成功编解码啦！

也可以把编解码器绑定和服务端和客户端，测试后也无疑是成功的！

Demo2：改进 Demo1 的代码

Demo1 作为一个简单的例子，虽然实现了根据\r\n 换行符编解码的功能，但是却存在以下问题：

1. 编解码器中编码类型 Charset 硬编码，不便调整；
2. 只能根据 Windows 的换行符\r\n 解码，没有考虑其他操作系统的换行符，不灵活；
3. 解码器中定义了成员变量 IoBuffer，但 Decoder 实例是单例的，因此 Decoder 实例中的成员变量可以被多线程共享访问，可能会因为变量的可见性而造成数据异常；

第 3 个 bug 是致命的，因此，必须首先解决；

为什么要定义成员变量 IoBuffer 呢？因为数据接收并不是一次完成的；比如客户端发送一个请求有 400 个字节，先发送了 200 个字节，这时暂停某段时间，然后又发送了剩余 200 字节；在解码时，Decode 的 IoBuffer 中先解码 200 个接收到的字节，此时，解码工作并未完成；但因为使用了 java NIO，发生 IO 阻塞时会处理其他请求，此时就需要把先接收到的数据暂存在某个变量中，当剩余数据到达时，解码后追加在原来数据后面；

这就是我们定义成员变量 IoBuffer 的理由！

```
IoBuffer buf = IoBuffer.allocate(100).setAutoExpand(true);

    public void decode(IoSession session, IoBuffer in, ProtocolDecoderOutput out)
            throws Exception {
```

此时，问题出现了！

每个IoSession都需要有自己的解码器实例；MINA确保同一时刻只有一个线

程在执行decode() 函数——不允许多线程并发地执行解码函数，但它并不能保证每次解码过程都是同一线程在执行（两次解码用的可能是不同的线程）。假设第一块数据被线程1管理，这时还没接收到足够的数据以供解码，当接收到第二块数据时，被另一个线程2管理，此时可能会出现变量的可视化(Visibility)问题。

因此，每个IoSession都需要独立保存解码器所解码时未完成的数据。办法就是保存在IoSession的属性中，每次解码时，都先从它的属性中拿出上次未完成的任务数据，把新数据追加在它的后面；

源码如下：

```
public class MyTextLineCodecDecoder implements ProtocolDecoder {
    private Charset charset = Charset.forName("utf-8");
    // 定义常量值，作为每个IoSession中保存解码内容的key值
    private static String CONTEXT = MyTextLineCodecDecoder.class.getName()
            + ".context";

    public void decode(IoSession session, IoBuffer in, ProtocolDecoderOutput out)
            throws Exception {
        Context ctx = getContext(session);
        decodeAuto(ctx, in, out);
    }

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

    private void decodeAuto(Context ctx, IoBuffer in, ProtocolDecoderOutput out)
            throws CharacterCodingException {
        boolean mark = false;
        while (in.hasRemaining()) {
            byte b = in.get();
            switch (b) {
            case '\r':
                break;
            case '\n':
                mark = true;
                break; // 跳出switch
            default:
                ctx.getBuf().put(b);
            }
```

```
            if (mark) {
                IoBuffer t_buf = ctx.getBuf();
                t_buf.flip();
                try {
                    out.write(t_buf.getString(charset.newDecoder()));
                } finally {
                    t_buf.clear();
                }
            }
        }
    }

    public void dispose(IoSession session) throws Exception {
        Context ctx = (Context) session.getAttribute(CONTEXT);
        if (ctx != null) {
            session.removeAttribute(CONTEXT);
        }
    }

    public void finishDecode(IoSession session, ProtocolDecoderOutput out)
            throws Exception {
    }

    private class Context {
        private IoBuffer buf;

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

        public IoBuffer getBuf() {
            return buf;
        }
    }
}
```

代码解释：

1. 在解码器中定义一个内部类，内部类中有一个成员变量IoBuffer，用来存储每个IoSesssion解码的内容；

```
private class Context {
        private IoBuffer buf;

        public Context() {
            buf = IoBuffer.allocate(100).setAutoExpand(true);
```

```
        }

        public IoBuffer getBuf() {
            return buf;
        }
    }
```

2. 当IoSession使用解码实例时，第一次使用则新建一个Context对象，保存在IoSession的Attribute中，把解码内容保存在Context对象的成员变量IoBuffer中；如果解码没结束，第二次使用解码实例时，从IoSession的Attribute取出Context对象，保额解码内容追加在Context对象的成员变量IoBuffer中；
   注意：IoSession的Attribute使用用一个同步的HashMap 保存对象，因此定义了常量CONTEXT作为保存Context对象的Key值；

```
private static String CONTEXT = MyTextLineCodecDecoder.class.getName()
            + ".context";

    public void decode(IoSession session, IoBuffer in, ProtocolDecoderOutput out)
            throws Exception {
        Context ctx = getContext(session);
        decodeAuto(ctx, in, out);
    }

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

3. 解码时，解码内容保存在Context对象的成员变量IoBuffer中，因此，一旦解码成功，要把成员变量IoBuffer重置；-----现在是请求/响应的单模式，不存在一个请求过来发送了多条记录的情况，所有重置前其实IoBuffer缓存内容已经为空；

```
private void decodeAuto(Context ctx, IoBuffer in, ProtocolDecoderOutput out)
            throws CharacterCodingException {
        boolean mark = false;
        while (in.hasRemaining()) {
            byte b = in.get();
            switch (b) {
            case '\r':
                break;
            case '\n':
                mark = true;
```

```
                break; // 跳出switch
            default:
                ctx.getBuf().put(b);
            }

            if (mark) {
                IoBuffer t_buf = ctx.getBuf();
                t_buf.flip();
                try {
                    out.write(t_buf.getString(charset.newDecoder()));
                } finally {
                    t_buf.clear();
                }
            }
        }
    }
```

4. 解码成功，则从IoSession的Attribute删除Context对象；

```
public void dispose(IoSession session) throws Exception {
        Context ctx = (Context) session.getAttribute(CONTEXT);
        if (ctx != null) {
            session.removeAttribute(CONTEXT);
        }
    }
```

查看 Mina 对 TextLineCodec 的实现源码会发现，根据换行符解码的消息最大长度是受限制的，默认最大长度是 1024，相当于缓冲区最大能存放 1K 的数据。因此使用时，建议调整参数为 2K；

如果我们希望根据我们自己定义的文本换行符及编码格式编解码，则需要把它们作为参数传递给编解码器；完整代码如下：

解码器：

```
import java.nio.charset.CharacterCodingException;
import java.nio.charset.Charset;
import java.nio.charset.CharsetDecoder;

import org.apache.mina.common.IoBuffer;
import org.apache.mina.common.IoSession;
import org.apache.mina.filter.codec.ProtocolDecoder;
import org.apache.mina.filter.codec.ProtocolDecoderOutput;

public class MyTextLineCodecDecoderII implements ProtocolDecoder {
    private Charset charset; // 编码格式

    private String delimiter; // 文本分隔符

    private IoBuffer delimBuf; // 文本分割符匹配的变量
```

```
    // 定义常量值，作为每个IoSession中保存解码任务的key值
    private static String CONTEXT = MyTextLineCodecDecoder.class.getName()
            + ".context";

    // 构造函数，必须指定Charset和文本分隔符
    public MyTextLineCodecDecoderII(Charset charset, String delimiter) {
        this.charset = charset;
        this.delimiter = delimiter;
    }

    public void decode(IoSession session, IoBuffer in, ProtocolDecoderOutput out)
            throws Exception {
        Context ctx = getContext(session);
        if (delimiter == null || "".equals(delimiter)) { // 如果文本换行符未指定,
使用默认值
            delimiter = "\r\n";
        }
        if (charset == null) {
            charset = Charset.forName("utf-8");
        }
        decodeNormal(ctx, in, out);
    }

    // 从IoSession中获取Context对象
    private Context getContext(IoSession session) {
        Context ctx;
        ctx = (Context) session.getAttribute(CONTEXT);
        if (ctx == null) {
            ctx = new Context();
            session.setAttribute(CONTEXT, ctx);
        }
        return ctx;
    }

    // 解码
    private void decodeNormal(Context ctx, IoBuffer in,
            ProtocolDecoderOutput out) throws CharacterCodingException {
        // 取出未完成任务中已经匹配的文本换行符的个数
        int matchCount = ctx.getMatchCount();

        // 设置匹配文本换行符的IoBuffer变量
        if (delimBuf == null) {
            IoBuffer tmp = IoBuffer.allocate(2).setAutoExpand(true);
```

```
            tmp.putString(delimiter, charset.newEncoder());
            tmp.flip();
            delimBuf = tmp;
        }

        int oldPos = in.position(); // 解码的IoBuffer中数据的原始信息
        int oldLimit = in.limit();
        while (in.hasRemaining()) { // 变量解码的IoBuffer
            byte b = in.get();
            if (delimBuf.get(matchCount) == b) { // 匹配第matchCount位换行符成功
                matchCount++;
                if (matchCount == delimBuf.limit()) { // 当前匹配到字节个数与文本换行符字节个数相同，匹配结束
                    int pos = in.position();    // 获得当前匹配到的position（position前所有数据有效）
                    in.limit(pos);
                    in.position(oldPos);    // position回到原始位置

                    ctx.append(in);    // 追加到Context对象未完成数据后面

                    in.limit(oldLimit); // in中匹配结束后剩余数据
                    in.position(pos);

                    IoBuffer buf = ctx.getBuf();
                    buf.flip();
                    buf.limit(buf.limit() - matchCount);// 去掉匹配数据中的文本换行符
                    try {
                        out.write(buf.getString(ctx.getDecoder())); // 输出解码内容
                    } finally {
                        buf.clear(); // 释放缓存空间
                    }

                    oldPos = pos;
                    matchCount = 0;
                }
            } else {
                // 如果matchCount==0，则继续匹配
                // 如果matchCount>0，说明没有匹配到文本换行符的中的前一个匹配成功字节的下一个字节，
                // 跳转到匹配失败字符处，并置matchCount=0，继续匹配
                in.position(in.position()-matchCount);
                matchCount = 0;  // 匹配成功后，matchCount置空
```

```
            }
        }

        // 把in中未解码内容放回buf中
        in.position(oldPos);
        ctx.append(in);

        ctx.setMatchCount(matchCount);
    }

    public void dispose(IoSession session) throws Exception {

    }

    public void finishDecode(IoSession session, ProtocolDecoderOutput out)
            throws Exception {
    }

    // 内部类，保存IoSession解码时未完成的任务
    private class Context {
        private CharsetDecoder decoder;
        private IoBuffer buf; // 保存真实解码内容
        private int matchCount = 0; // 匹配到的文本换行符个数

        private Context() {
            decoder = charset.newDecoder();
            buf = IoBuffer.allocate(80).setAutoExpand(true);
        }

        // 重置
        public void reset() {
            matchCount = 0;
            decoder.reset();
        }

        // 追加数据
        public void append(IoBuffer in) {
            getBuf().put(in);
        }

        // ======get/set方法====================
        public CharsetDecoder getDecoder() {
            return decoder;
        }
```

```
        public IoBuffer getBuf() {
            return buf;
        }

        public int getMatchCount() {
            return matchCount;
        }

        public void setMatchCount(int matchCount) {
            this.matchCount = matchCount;
        }
    } // end class Context;
}
```

编码器：

```
import java.nio.charset.Charset;

import org.apache.mina.common.IoBuffer;
import org.apache.mina.common.IoSession;
import org.apache.mina.filter.codec.ProtocolEncoder;
import org.apache.mina.filter.codec.ProtocolEncoderOutput;

public class MyTextLineCodecEncoderII implements ProtocolEncoder {
    private Charset charset; // 编码格式

    private String delimiter; // 文本分隔符

    public MyTextLineCodecEncoderII(Charset charset, String delimiter) {
        this.charset = charset;
        this.delimiter = delimiter;
    }

    public void encode(IoSession session, Object message,
            ProtocolEncoderOutput out) throws Exception {
        if (delimiter == null || "".equals(delimiter)) { // 如果文本换行符未指定,
使用默认值
            delimiter = "\r\n";
        }
        if (charset == null) {
            charset = Charset.forName("utf-8");
        }

        String value = message.toString();
        IoBuffer buf = IoBuffer.allocate(value.length()).setAutoExpand(true);
```

```
        buf.putString(value, charset.newEncoder()); // 真实数据
        buf.putString(delimiter, charset.newEncoder()); // 文本换行符
        buf.flip();
        out.write(buf);
    }

    public void dispose(IoSession session) throws Exception {
    }
}
```

编解码器工厂：

```
import java.nio.charset.Charset;

import org.apache.mina.common.IoSession;
import org.apache.mina.filter.codec.ProtocolCodecFactory;
import org.apache.mina.filter.codec.ProtocolDecoder;
import org.apache.mina.filter.codec.ProtocolEncoder;

public class MyTextLineCodecFactoryII implements ProtocolCodecFactory {
    private Charset charset; // 编码格式

    private String delimiter; // 文本分隔符

    public MyTextLineCodecFactoryII(Charset charset, String delimiter) {
        this.charset = charset;
        this.delimiter = delimiter;
    }

    public ProtocolDecoder getDecoder(IoSession session) throws Exception {
        return new MyTextLineCodecDecoderII(charset, delimiter);
    }

    public ProtocolEncoder getEncoder(IoSession session) throws Exception {
        return new MyTextLineCodecEncoderII(charset, delimiter);
    }
}
```

服务端或客户端绑定过滤器：

```
// 添加过滤器
connector.getFilterChain().addLast(
                "codec",
    new ProtocolCodecFilter(new MyTextLineCodecFactoryII(Charset
                        .forName("utf-8"), "\r\n")));
```

Demo3：自定义协议编解码

自定义协议是使用最广泛的，因为它非常的灵活！

第一步：制定协议：

协议需求：向服务端发送请求（频道的 ID 和说明文字），返回响应结果（该频道下所有的节目信息）；

协议格式如下：

请求格式

Syntax	No. of Bits	Identifier
_ description (){		
Descriptor tag	16	0x0001
descriptor length	32	从下一字节开始至末尾的数据长度
ID	16	channel ID 值
chanel_des_len	8	频道说明文字
for(i=0;i< chanel_des_len;i++){		
Byte_data	8	
}		
}		// end _description

响应格式

Syntax	No. of Bits	Identifier
_ description (){		
Tag	16	0x8001
Data_length	32	从下一字节开始至末尾的数据长度
channel_addr	32	频道名称的地址
channel _len	8	频道名称的字符串长度
programme_count	16	节目个数
for(i=0;i< programme_count;i++){		
dayIndex	8	属于哪一天（以当日为基准）(-1 表示前一天;0 表示当天;1 表示下一天;2 表示后两天)
event_addr	32	节目名称的地址
event _len	8	节目名称的字符串长度
StartTime	32	节目偏移开始时间
TotalTime	16	节目总时长(以秒为单位)
Status	8	节目当前状态（已录制【0x01】//待录制【0x00】）
url_addr	32	节目播放地址的 addr
url _len	8	节目播放地址的长度
}		// end for
For(j=0;j<;j++){		
Byte_data	8	真实数据
}		
}		// end _description

协议解释如下：

1. 协议前两个字节(16Bits)是协议的唯一标识值；
   如上：请求部分的 tag = 0x0001, 响应部分的 tag = 0x8001

2. 接着四个字节(32Bits)是传输消息的长度；
3. 接下来是数据区；

分析请求部分：

Syntax	No. of Bits	Identifier
_ description (){		
Descriptor tag	16	0x0001
descriptor length	32	从下一字节开始至末尾的数据长度
ID	16	channel ID 值
chanel_des_len	8	频道说明文字
for(i=0;i< chanel_des_len;i++){		
Byte_data	8	
}		
}		// end _description

请求部分是客户端（机顶盒）向服务端发送的请求；协议 I 的请求只发送了两个个参数：channelID 和 channel_dec（频道描述信息）

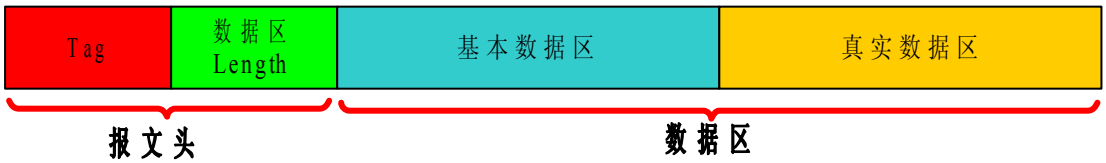
各个参数分析：

a. descriptor tag:请求的唯一标识； -- 2 个字节
b. descriptor length:数据区长度； -- 4 个字节
c. ID：channelID; -- 2 个字节
d. channel_dec_len:频道说明信息的字节长度 -- 1 个字节
e. for 循环：存放频道说明信息的真实数据（字节数组中）

响应部分分析略……

// ==========协议格式总结==========
前面 2 个绿色部分称为报文头，固定 6 个字节；
中间 2 个蓝色部分称为基本数据区，用 Java 的 8 个基本数据类型描述；
最后的红色部分称为真实数据区，所有 String 类型的信息都放在这里；
基本数据区+真实数据区 =数据区
协议格式：报文头+数据区

图示如下：



总之，对于基本数据类型，直接存放在基本数据区，对于 String 类型，在基本数据区描述它的长度和在真实数据区的地址，然后存在在真实数据区；而 Java 对象，则是把对象属性分解为基本数据类型和 String 类型发送；

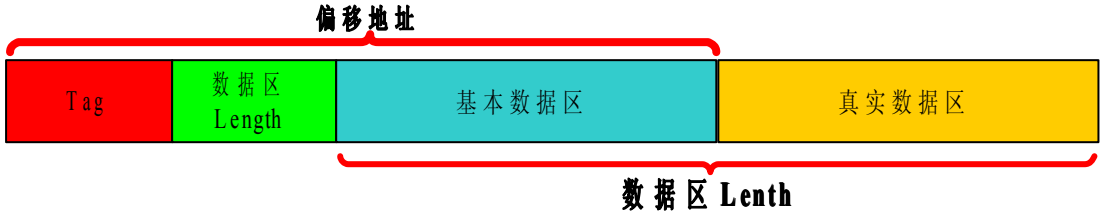
因此，解码必须获得三个信息：

a. 请求标识：根据请求的不同进行不同的解码

b. 数据区总长度：定是否接受数据成功；

c. 偏移地址：知道真实数据区位置，就可以解码 String 数据；

图示如下：



代码实现：

1. 首先定义消息的抽象类，定义获取 3 个解码信息的方法；

```
import java.nio.charset.Charset;

public abstract class AbstrMessage {
    // 协议编号
    public abstract short getTag();

    // 数据区长度
    public abstract int getLen(Charset charset);

    // 真实数据偏移地址
    public abstract int getDataOffset();
}
```

定义请求对象和响应对象；

请求的 Java 对象：

```
import java.nio.charset.Charset;
import org.apache.log4j.Logger;
/*
 * 请求的Java对象
 */
public class ChannelInfoRequest extends AbstrMessage {
    private Logger logger = Logger.getLogger(ChannelInfoRequest.class);

    private String channel_desc;

    private int channel_id;

    @Override
    public short getTag() {
        return (short) 0x0001;
    }

    @Override
    public int getLen(Charset charset) {
```

```
        int len = 2 + 1;
        try {
            if (channel_desc != null && !"".equals(channel_desc)) {
                len += channel_desc.getBytes(charset).length;
            }
        } catch (Exception e) {
            logger.error("频道说明转换为字节码错误...", e);
        }
        return len;
    }

    @Override
    public int getDataOffset() {
        int len = 2 + 4 + 2 + 1;
        return len;
    }

    public String getChannel_desc() {
        return channel_desc;
    }

    public void setChannel_desc(String channel_desc) {
        this.channel_desc = channel_desc;
    }

    public int getChannel_id() {
        return channel_id;
    }

    public void setChannel_id(int channel_id) {
        this.channel_id = channel_id;
    }
}
```

响应的 Java 对象：

```
import java.nio.charset.Charset;
import org.apache.log4j.Logger;
/*
 * 响应的Java对象
 */
public class ChannelInfoResponse extends AbstrMessage {
    private Logger logger = Logger.getLogger(ChannelInfoResponse.class);

    private String ChannelName;
```

```
    private EventDto[] events;

    @Override
    public short getTag() {
        return (short) 0x8001;
    }

    @Override
    public int getLen(Charset charset) {
        int len = 4 + 1 + 2;
        try {
            if (events != null && events.length > 0) {
                for (int i = 0; i < events.length; i++) {
                    EventDto edt = events[i];
                    len += 1 + 4 + 1 + 4 + 2 + 1 + 4 + 1 + edt.getLen(charset);
                }
            }
            if (ChannelName != null && !"".equals(ChannelName)) {
                len += ChannelName.getBytes(charset).length;
            }
        } catch (Exception e) {
            logger.error("频道信息转换为字节码错误...", e);
        }
        return len;
    }

    @Override
    public int getDataOffset() {
        int len = 2 + 4 + 4 + 1 + 2;
        if (events != null && events.length > 0) {
            len += events.length * (1 + 4 + 1 + 4 + 2 + 1 + 4 + 1);
        }
        return len;
    }

    public String getChannelName() {
        return ChannelName;
    }

    public void setChannelName(String channelName) {
        ChannelName = channelName;
    }

    public EventDto[] getEvents() {
```

```
        return events;
    }

    public void setEvents(EventDto[] events) {
        this.events = events;
    }
}
import java.nio.charset.Charset;
import org.apache.log4j.Logger;
public class EventDto {
    private Logger logger = Logger.getLogger(EventDto.class);

    private String eventName;

    private int beginTime;

    private int totalTime;

    private int dayIndex;

    private int status;

    private String url;

    // 节目中字符数据的字节长度
    public int getLen(Charset charset) {
        int len = 0;
        try {
            if (eventName != null && !"".equals(eventName)) {
                len += eventName.getBytes(charset).length;
            }
            if (url != null && !"".equals(url)) {
                len += url.getBytes(charset).length;
            }
        } catch (Exception e) {
            logger.error("节目信息转换为字节码错误...", e);
        }
        return len;
    }
   // ·····get/set方法，略·····
}
```

解码器：

```
import java.nio.charset.Charset;
import java.nio.charset.CharsetDecoder;
```

```
import org.apache.log4j.Logger;
import org.apache.mina.common.IoBuffer;
import org.apache.mina.common.IoSession;
import org.apache.mina.filter.codec.ProtocolDecoderOutput;
import org.apache.mina.filter.codec.demux.MessageDecoder;
import org.apache.mina.filter.codec.demux.MessageDecoderResult;

import com.dvn.li.message.AbstrMessage;
import com.dvn.li.message.ChannelInfoRequest;
import com.dvn.li.message.ChannelInfoResponse;
import com.dvn.li.message.EventDto;

public class MyMessageDecoder implements MessageDecoder {
    private Logger logger = Logger.getLogger(MyMessageDecoder.class);

    private Charset charset;

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

    // 检查给定的IoBuffer是否适合解码
    public MessageDecoderResult decodable(IoSession session, IoBuffer in) {
        // 报头长度==6
        if (in.remaining() < 6) {
            return MessageDecoderResult.NEED_DATA;
        }

        // tag正常
        short tag = in.getShort();
        // 注意先把16进制标识值转换为short类型的十进制数据，然后与tag比较
        if (tag == (short) 0x0001 || tag == (short) 0x8001) {
            logger.info("请求标识符：" + tag);
        } else {
            logger.error("未知的解码类型....");
            return MessageDecoderResult.NOT_OK;
        }

        // 真实数据长度
        int len = in.getInt();
        if (in.remaining() < len) {
            return MessageDecoderResult.NEED_DATA;
        }
```

```
        return MessageDecoderResult.OK;
    }

    public MessageDecoderResult decode(IoSession session, IoBuffer in,
            ProtocolDecoderOutput out) throws Exception {
        logger.info("解码: " + in.toString());
        CharsetDecoder decoder = charset.newDecoder();
        AbstrMessage message = null;
        short tag = in.getShort(); // tag
        int len = in.getInt(); // len

        byte[] temp = new byte[len];
        in.get(temp); // 数据区

        // ==============解析数据做准备====================
        IoBuffer buf = IoBuffer.allocate(100).setAutoExpand(true);
        buf.put(temp);
        buf.flip(); // 为获取基本数据区长度做准备

        IoBuffer databuf = IoBuffer.allocate(100).setAutoExpand(true);
        databuf.putShort(tag);
        databuf.putInt(len);
        databuf.put(temp);
        databuf.flip(); // 为获取真实数据区长度做准备

        // ===============开始解码=======================
        // 注意先把16进制标识值转换为short类型的十进制数据，然后与tag比较
        if (tag == (short) 0x0001) { // 服务端解码
            ChannelInfoRequest req = new ChannelInfoRequest();

            short channel_id = buf.getShort();
            byte channel_desc_len = buf.get();
            String channel_desc = null;
            if (channel_desc_len > 0) {
                channel_desc = buf.getString(channel_desc_len, decoder);
            }

            req.setChannel_id(channel_id);
            req.setChannel_desc(channel_desc);

            message = req;
        } else if (tag == (short) 0x8001) { // 客户端解码
            ChannelInfoResponse res = new ChannelInfoResponse();
```

```
int channel_addr = buf.getInt();
byte channel_len = buf.get();
if (databuf.position() == 0) {
    databuf.position(channel_addr);
}
String channelName = null;
if (channel_len > 0) {
    channelName = databuf.getString(channel_len, decoder);
}
res.setChannelName(channelName);

short event_num = buf.getShort();
EventDto[] events = new EventDto[event_num];
for (int i = 0; i < event_num; i++) {
    EventDto edt = new EventDto();
    byte dayIndex = buf.get();

    buf.getInt();
    byte eventName_len = buf.get();
    String eventName = null;
    if (eventName_len > 0) {
        eventName = databuf.getString(eventName_len, decoder);
    }

    int beginTime = buf.getInt();
    short totalTime = buf.getShort();
    byte status = buf.get();

    buf.getInt();
    byte url_len = buf.get();
    String url = null;
    if (url_len > 0) {
        url = databuf.getString(url_len, decoder);
    }
    edt.setDayIndex(dayIndex);
    edt.setEventName(eventName);
    edt.setBeginTime(beginTime);
    edt.setTotalTime(totalTime);
    edt.setStatus(status);
    edt.setUrl(url);

    events[i] = edt;
}
```

```
            res.setEvents(events);
            message = res;
        } else {
            logger.error("未找到解码器....");
        }
        out.write(message);
        // ===============解码成功=======================
        return MessageDecoderResult.OK;
    }

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

编码器：

```
import java.nio.charset.Charset;
import java.nio.charset.CharsetEncoder;

import org.apache.log4j.Logger;
import org.apache.mina.common.IoBuffer;
import org.apache.mina.common.IoSession;
import org.apache.mina.filter.codec.ProtocolEncoderOutput;
import org.apache.mina.filter.codec.demux.MessageEncoder;

import com.dvn.li.message.AbstrMessage;
import com.dvn.li.message.ChannelInfoRequest;
import com.dvn.li.message.ChannelInfoResponse;
import com.dvn.li.message.EventDto;

public class MyMessageEncoder implements MessageEncoder<AbstrMessage> {
    private Logger logger = Logger.getLogger(MyMessageEncoder.class);

    private Charset charset;

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

    public void encode(IoSession session, AbstrMessage message,
            ProtocolEncoderOutput out) throws Exception {
        IoBuffer buf = IoBuffer.allocate(100).setAutoExpand(true);
        buf.putShort(message.getTag());
        buf.putInt(message.getLen(charset));
```

```
        // ==========编码数据区==============
        if (message instanceof ChannelInfoRequest) {
            ChannelInfoRequest req = (ChannelInfoRequest) message;
            buf.putShort((short) req.getChannel_id());
            buf.put((byte) req.getChannel_desc().getBytes(charset).length);
            buf.putString(req.getChannel_desc(), charset.newEncoder());
        } else if (message instanceof ChannelInfoResponse) {
            ChannelInfoResponse res = (ChannelInfoResponse) message;
            CharsetEncoder encoder = charset.newEncoder();
            IoBuffer dataBuffer = IoBuffer.allocate(100).setAutoExpand(true); //
定义真实数据区
            int offset = res.getDataOffset(); // 偏移地址

            buf.putInt(offset); // 频道名称地址（偏移开始位置）
            byte channelName_len = 0;
            if (res.getChannelName() != null) {
                channelName_len = (byte)
res.getChannelName().getBytes(charset).length;
            }
            buf.put(channelName_len);
            offset += channelName_len;
            if (channelName_len > 0) {
                dataBuffer.putString(res.getChannelName(), encoder);
            }

            EventDto[] events = res.getEvents();
            if (events != null) {
                buf.putShort((short) events.length);
                for (int i = 0; i < events.length; i++) {
                    EventDto edt = events[i];

                    buf.put((byte) edt.getDayIndex());

                    buf.putInt(offset);
                    String eventName = edt.getEventName();
                    byte eventName_len = 0;
                    if (eventName != null) {
                        eventName_len = (byte)
eventName.getBytes(charset).length;
                    }
                    offset += eventName_len;
                    buf.put(eventName_len);
                    if (eventName_len > 0) {
                        dataBuffer.putString(eventName, encoder);
```

```
                    }

                    buf.putInt(edt.getBeginTime());
                    buf.putShort((short) edt.getTotalTime());
                    buf.put((byte) edt.getStatus());

                    buf.putInt(offset);
                    String url = edt.getUrl();
                    byte url_len = 0;
                    if (url != null) {
                        url_len = (byte) url.getBytes(charset).length;
                    }
                    offset += url_len;
                    buf.put(url_len);
                    if (url_len > 0) {
                        dataBuffer.putString(url, encoder);
                    }
                }
            }

            // 真实数据追加在基本数据后面
            if (dataBuffer.position() > 0) {
                buf.put(dataBuffer.flip());
            }
        }

        // ==========编码成功================
        buf.flip();
        logger.info("编码" + buf.toString());
        out.write(buf);
    }
}
```

编解码器工厂：

```
import org.apache.mina.filter.codec.demux.DemuxingProtocolCodecFactory;
import org.apache.mina.filter.codec.demux.MessageDecoder;
import org.apache.mina.filter.codec.demux.MessageEncoder;

import com.dvn.li.message.AbstrMessage;

public class MyMessageCodecFactory extends DemuxingProtocolCodecFactory {
    private MessageDecoder decoder;

    private MessageEncoder<AbstrMessage> encoder;
    // 注册编解码器
```

```
    public MyMessageCodecFactory(MessageDecoder decoder,
            MessageEncoder<AbstrMessage> encoder) {
        this.decoder = decoder;
        this.encoder = encoder;
        addMessageDecoder(this.decoder);
        addMessageEncoder(AbstrMessage.class, this.encoder);
    }
}
```

服务端和服务端处理类：

```
import java.net.InetSocketAddress;
import java.nio.charset.Charset;

import org.apache.log4j.Logger;
import org.apache.mina.common.IdleStatus;
import org.apache.mina.common.IoAcceptor;
import org.apache.mina.common.IoSessionConfig;
import org.apache.mina.filter.codec.ProtocolCodecFilter;
import org.apache.mina.filter.logging.LogLevel;
import org.apache.mina.filter.logging.LoggingFilter;
import org.apache.mina.transport.socket.nio.NioSocketAcceptor;

import com.dvn.li.codec.MyMessageCodecFactory;
import com.dvn.li.codec.MyMessageDecoder;
import com.dvn.li.codec.MyMessageEncoder;
import com.dvn.li.handler.Demo2ServerHandler;

public class TestServer02 {
    private static Logger logger = Logger.getLogger(TestServer02.class);

    private static int PORT = 3005;

    public static void main(String[] args) {
        IoAcceptor acceptor = null;
        try {
            // 创建一个非阻塞的server端的Socket
            acceptor = new NioSocketAcceptor();

            // 设置过滤器（添加自带的编解码器）
            acceptor.getFilterChain().addLast(
                    "codec",
                    new ProtocolCodecFilter(new MyMessageCodecFactory(
                            new MyMessageDecoder(Charset.forName("utf-8")),
                            new MyMessageEncoder(Charset.forName("utf-8")))));
            // 设置日志过滤器
```

```
            LoggingFilter lf = new LoggingFilter();
            lf.setMessageReceivedLogLevel(LogLevel.DEBUG);
            acceptor.getFilterChain().addLast("logger", lf);
            // 获得IoSessionConfig对象
            IoSessionConfig cfg = acceptor.getSessionConfig();
            // 读写通道10秒内无操作进入空闲状态
            cfg.setIdleTime(IdleStatus.BOTH_IDLE, 100);

            // 绑定逻辑处理器
            acceptor.setHandler(new Demo2ServerHandler());
            // 绑定端口
            acceptor.bind(new InetSocketAddress(PORT));
            logger.info("服务端启动成功...     端口号为：" + PORT);
        } catch (Exception e) {
            logger.error("服务端启动异常....", e);
            e.printStackTrace();
        }
    }
}
import org.apache.log4j.Logger;
import org.apache.mina.common.IdleStatus;
import org.apache.mina.common.IoHandlerAdapter;
import org.apache.mina.common.IoSession;

import com.dvn.li.message.ChannelInfoRequest;
import com.dvn.li.message.ChannelInfoResponse;
import com.dvn.li.message.EventDto;

public class Demo2ServerHandler extends IoHandlerAdapter {
    public static Logger logger = Logger.getLogger(Demo2ServerHandler.class);

    @Override
    public void sessionCreated(IoSession session) throws Exception {
        logger.info("服务端与客户端创建连接...");
    }

    @Override
    public void sessionOpened(IoSession session) throws Exception {
        logger.info("服务端与客户端连接打开...");
    }

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

```
        if (message instanceof ChannelInfoRequest) {
            ChannelInfoRequest req = (ChannelInfoRequest) message;
            int channel_id = req.getChannel_id();
            String channel_desc = req.getChannel_desc();
            logger.info("服务端接收到的数据为：channel_id=" + channel_id
                    + "   channel_desc=" + channel_desc);
            // ================具体操作，比如查询数据库等，这里略....=============
            ChannelInfoResponse res = new ChannelInfoResponse();
            res.setChannelName("CCTV1高清频道");
            EventDto[] events = new EventDto[2];
            for (int i = 0; i < events.length; i++) {
                EventDto edt = new EventDto();
                edt.setBeginTime(10);
                edt.setDayIndex(1);
                edt.setEventName("风云第一的" + i);
                edt.setStatus(1);
                edt.setTotalTime(100 + i);
                edt.setUrl("www.baidu.com");
                events[i] = edt;
            }
            res.setEvents(events);
            session.write(res);
        } else {
            logger.info("未知请求！");
        }

    }

    @Override
    public void messageSent(IoSession session, Object message) throws Exception {
        session.close();
        logger.info("服务端发送信息成功...");
    }

    @Override
    public void sessionClosed(IoSession session) throws Exception {

    }

    @Override
    public void sessionIdle(IoSession session, IdleStatus status)
            throws Exception {
        logger.info("服务端进入空闲状态...");
    }
```

```
    @Override
    public void exceptionCaught(IoSession session, Throwable cause)
            throws Exception {
        logger.error("服务端发送异常...", cause);
    }
}
```

客户端和客户端处理类：

```
import java.net.InetSocketAddress;
import java.nio.charset.Charset;

import org.apache.log4j.Logger;
import org.apache.mina.common.ConnectFuture;
import org.apache.mina.common.IoConnector;
import org.apache.mina.common.IoSession;
import org.apache.mina.filter.codec.ProtocolCodecFilter;
import org.apache.mina.transport.socket.nio.NioSocketConnector;

import com.dvn.li.codec.MyMessageCodecFactory;
import com.dvn.li.codec.MyMessageDecoder;
import com.dvn.li.codec.MyMessageEncoder;
import com.dvn.li.handler.Demo2ClientHandler;
import com.dvn.li.message.ChannelInfoRequest;

public class TestClient02 {
    private static Logger logger = Logger.getLogger(TestClient02.class);

    private static String HOST = "127.0.0.1";

    private static int PORT = 3005;

    public static void main(String[] args) {
        // 创建一个非阻塞的客户端程序
        IoConnector connector = new NioSocketConnector();
        // 设置链接超时时间
        connector.setConnectTimeout(30000);
        // 添加过滤器
        connector.getFilterChain().addLast(
                "codec",
                new ProtocolCodecFilter(new MyMessageCodecFactory(
                        new MyMessageDecoder(Charset.forName("utf-8")),
                        new MyMessageEncoder(Charset.forName("utf-8")))));
        // 添加业务逻辑处理器类
        connector.setHandler(new Demo2ClientHandler());
```

```
        IoSession session = null;
        try {
            ConnectFuture future = connector.connect(new InetSocketAddress(
                    HOST, PORT));// 创建连接
            future.awaitUninterruptibly();// 等待连接创建完成
            session = future.getSession();// 获得session
            ChannelInfoRequest req = new ChannelInfoRequest(); // 发送请求
            req.setChannel_id(12345);
            req.setChannel_desc("mina在做测试哦哦....哇呀呀！！！");
            session.write(req);// 发送消息
        } catch (Exception e) {
            logger.error("客户端链接异常...", e);
        }

        session.getCloseFuture().awaitUninterruptibly();// 等待连接断开
        connector.dispose();
    }
}
```

```
import org.apache.log4j.Logger;
import org.apache.mina.common.IoHandlerAdapter;
import org.apache.mina.common.IoSession;

import com.dvn.li.message.ChannelInfoResponse;
import com.dvn.li.message.EventDto;

public class Demo2ClientHandler extends IoHandlerAdapter {
    private static Logger logger = Logger.getLogger(Demo2ClientHandler.class);

    @Override
    public void messageReceived(IoSession session, Object message)
            throws Exception {
        if (message instanceof ChannelInfoResponse) {
            ChannelInfoResponse res = (ChannelInfoResponse) message;
            String channelName = res.getChannelName();
            EventDto[] events = res.getEvents();
            logger.info("客户端接收到的消息为：channelName=" + channelName);
            if(events!=null && events.length>0){
                for (int i = 0; i < events.length; i++) {
                    EventDto edt = events[i];
                    logger.info("客户端接收到的消息为：BeginTime=" + edt.getBeginTime());
                    logger.info("客户端接收到的消息为：DayIndex=" + edt.getDayIndex());
                    logger.info("客户端接收到的消息为：EventName=" + edt.getEventName());
                    logger.info("客户端接收到的消息为：Status=" + edt.getStatus());
                    logger.info("客户端接收到的消息为：TotalTime=" + edt.getTotalTime());
```

```
                    logger.info("客户端接收到的消息为：url=" + edt.getUrl());
                }
            }
        }else{
            logger.info("未知类型！");
        }
    }

    @Override
    public void exceptionCaught(IoSession session, Throwable cause)
            throws Exception {
        logger.error("客户端发生异常...", cause);
    }
}
```

测试……

服务端打印信息：

```
2010-12-23 13:57:36,533 INFO  TestServer02  - 服务端启动成功...     端口号为：3005
2010-12-23 13:57:42,049 INFO  LoggingFilter  - CREATED
2010-12-23 13:57:42,049 INFO  Demo2ServerHandler  - 服务端与客户端创建连接...
2010-12-23 13:57:42,049 INFO  LoggingFilter  - OPENED
2010-12-23 13:57:42,049 INFO  Demo2ServerHandler  - 服务端与客户端连接打开...
2010-12-23 13:57:42,112 INFO  MyMessageDecoder  - 请求标识符：1
2010-12-23 13:57:42,112 INFO  MyMessageDecoder  - 解码：HeapBuffer[pos=0 lim=53 cap=2048: 00 01
00 00 00 2F 30 39 2C 6D 69 6E 61 E5 9C A8...]
2010-12-23 13:57:42,127 DEBUG LoggingFilter  - RECEIVED:
com.dvn.li.message.ChannelInfoRequest@1e3118a
2010-12-23 13:57:42,127 INFO  Demo2ServerHandler  - 服务端接收到的数据为：channel_id=12345
channel_desc=mina在做测试哦哦....哇呀呀！！！
2010-12-23 13:57:42,158 INFO  MyMessageEncoder  - 编码HeapBuffer[pos=0 lim=124 cap=124: 80 01
00 00 00 76 00 00 00 31 11 00 02 01 00 00...]
2010-12-23 13:57:42,174 INFO  LoggingFilter  - SENT:
com.dvn.li.message.ChannelInfoResponse@be0e27
2010-12-23 13:57:42,174 INFO  Demo2ServerHandler  - 服务端发送信息成功...
2010-12-23 13:57:42,174 INFO  LoggingFilter  - CLOSED
```

客户端打印信息：

```
2010-12-23 13:57:42,080 INFO  MyMessageEncoder  - 编码HeapBuffer[pos=0 lim=53 cap=100: 00 01 00
00 00 2F 30 39 2C 6D 69 6E 61 E5 9C A8...]
2010-12-23 13:57:42,158 INFO  MyMessageDecoder  - 请求标识符：-32767
2010-12-23 13:57:42,174 INFO  MyMessageDecoder  - 解码：HeapBuffer[pos=0 lim=124 cap=2048: 80
01 00 00 00 76 00 00 00 31 11 00 02 01 00 00...]
2010-12-23 13:57:42,174 INFO  Demo2ClientHandler  - 客户端接收到的消息为：channelName=CCTV1
高清频道
2010-12-23 13:57:42,190 INFO  Demo2ClientHandler  - 客户端接收到的消息为：BeginTime=10
2010-12-23 13:57:42,190 INFO  Demo2ClientHandler  - 客户端接收到的消息为：DayIndex=1
```

```
2010-12-23 13:57:42,190 INFO  Demo2ClientHandler  - 客户端接收到的消息为：EventName=风云第一的0
2010-12-23 13:57:42,190 INFO  Demo2ClientHandler  - 客户端接收到的消息为：Status=1
2010-12-23 13:57:42,190 INFO  Demo2ClientHandler  - 客户端接收到的消息为：TotalTime=100
2010-12-23 13:57:42,190 INFO  Demo2ClientHandler  - 客户端接收到的消息为：url=www.baidu.com
2010-12-23 13:57:42,190 INFO  Demo2ClientHandler  - 客户端接收到的消息为：BeginTime=10
2010-12-23 13:57:42,190 INFO  Demo2ClientHandler  - 客户端接收到的消息为：DayIndex=1
2010-12-23 13:57:42,190 INFO  Demo2ClientHandler  - 客户端接收到的消息为：EventName=风云第一的1
2010-12-23 13:57:42,190 INFO  Demo2ClientHandler  - 客户端接收到的消息为：Status=1
2010-12-23 13:57:42,205 INFO  Demo2ClientHandler  - 客户端接收到的消息为：TotalTime=101
2010-12-23 13:57:42,205 INFO  Demo2ClientHandler  - 客户端接收到的消息为：url=www.baidu.com
```

3.IoHandler 接口

IoHandler 是 Mina 实现其业务逻辑的顶级接口；在 IoHandler 中定义了 7 个方法，根据 I/O 事件来触发对应的方法：

```
import java.io.IOException;
public interface IoHandler {
    void sessionCreated(IoSession session) throws Exception;
    void sessionOpened(IoSession session) throws Exception;
    void sessionClosed(IoSession session) throws Exception;
    void sessionIdle(IoSession session, IdleStatus status) throws Exception;
    void exceptionCaught(IoSession session, Throwable cause) throws Exception;
    void messageReceived(IoSession session, Object message) throws Exception;
    void messageSent(IoSession session, Object message) throws Exception;
}
```

sessionCreated:当一个新的连接建立时，由 I/O processor thread 调用；
sessionOpened:当连接打开是调用；
messageReceived:当接收了一个消息时调用；
messageSent：当一个消息被(IoSession#write)发送出去后调用；
sessionIdle:当连接进入空闲状态时调用；
sessionClosed:当连接关闭时调用；
exceptionCaught:当实现 IoHandler 的类抛出异常时调用；

一般情况下，我们最关心的只有 messageReceived 方法，接收消息并处理，然后调用 IoSession 的 write 方法发送出消息！(注意：这里接收到的消息都是 Java 对象，在 IoFilter 中所有二进制数据都被解码啦！)

一般情况下很少有人实现 IoHandler 接口，而是继承它的一个实现类 IoHandlerAdapter，这样不用覆盖它的 7 个方法，只需要根据具体需求覆盖其中的几个方法就可以！

三、 Mina 实例

实例 1：继承 CumulativeProtocolDecoder 类实现根据文本换行符解码；
实例 2：根据协议编写 Mina 应用程序；

请求格式

Syntax	No. of Bits	Identifier
_descript (){		
Descriptor tag	16	0x0008
descriptor length	32	从下一字节开始至末尾的数据长度
ID	32	节目的 ID
Program_code_addr	32	Asset_code
Program_code_len	8	
EPIODES_addr	32	是第几集节目字符地址
EPIODES_length	8	长度
For(j=0;j<;j++){		
byte_data	8	数据
}		
}		

响应格式

Syntax	No. of Bits	
_descript (){		
Tag	16	数据结构标志（0x8008）
Data_length	32	从下一字节开始至末尾的数据长度
Count	16	关联节目节目个数
for(i=0;i< count;i++){		
Programme_Type	32	节目类型 0x01 标示电影 0x02 标示电视剧 0x03 标示新闻 0x04 标示时移
TitleID	32	节目的 ID
Program_code_addr	32	Asset_code
Program_code_len	8	
EPIODES_addr	32	是第几集节目字符地址
EPIODES_length	8	长度
TotalTime	16	节目总时长
Offset_time	16	节目偏移时间（非新闻类该值为 0x0）
name_addr	32	节目名称的地址
name_len	8	节目名称的字符串长度
StringID_addr	32	节目媒资 ID 地址
StringID_len	8	节目媒资 ID 长度
}		
For(j=0;j<;j++){		
Byte_data	8	数据
}		
}		

四、 其他

……略……