MINA2 官方教程翻译(1) 2.x 与 1.x 的变化

文章分类:Java 编程

一、包与命名

所有的类和方法严格使用驼峰法命名。

例如 SSLFilter 被更名为 SslFilter，其它很多类也是如此。

所有 NIO 传输类在命名时增加‘Nio’前缀。

因为 NIO 并不只是 socket/datagram 传输的实现，所有‘Nio’前缀加在了所有的 NIO 传输类上。
改变之前：
Java 代码

```
1. SocketAcceptor acceptor = new SocketAcceptor();
```

改变之后：
Java 代码

```
1. SocketAcceptor acceptor = new NioSocketAcceptor();
```

Filter 类被重新整理进多重子包内。

随着框架自带的 filter 实现的数量的增加，所有的 filter 都被移动到适当的子包中(例如，StreamWriteFilter 移至 org.apache.mina.filter.stream)。
*.support 的所有包被移动到了其父包(或者其他包)中。
为了避免循环依赖，*.support 包中的所有类都被移至其父包或者其他包中。你可以在 IDE(例如 Eclipse)中简单的修正这些包的导入从而避免编译错误。

二、Buffers

MINA ByteBuffer 被重命名为 IoBuffer。

因为 MINA ByteBuffer 与 JDK 中 NIO ByteBuffer 同名，很多用户发现与其组员沟通时存在很多困难。根据用户的反馈，我们将 MINA ByteBuffer 重命名为 IoBuffer，这不仅使类名称简化，也是类名称更加明晰。

放弃 Buffer 池，默认使用 IoBuffer.allocate(int)来分配 heap buffer。

- acquire()与 release()两个方法将不再是容易发生错误的。如果你愿意，你可以调用 free()方法，但这是可选的。请自己承担使用这个方法的风险。
- 在大多数 JVM 中，框架内置的 IoBuffer 性能更加强劲、稳定。

Direct buffer 池是 MINA 早期版本所标榜的众多特性之一。然而根据当今的尺度，在主流的 JVM 中 direct buffers 的表现要比 heap buffers 差。此外，当 direct buffer memory 的最大值没有被正确设定时，不可预期的 OutOfMemoryError 也经常出现。
为了使系统内置的 IoBuffer 性能更加强劲、稳定，Apache MINA 项目组将默认的 buffer 类型由 direct 改为 heap。因为 heap buffers 并不需要池化，PooledByteBufferAllocator 也被移除掉了。由于没有了池的概念，ByteBuffer.acquire() 和 ByteBuffer.release()也被移除掉了。
然而，如果使用的速度太快，分配 heap buffers 也会成为瓶颈。这是因为分配字节数据如要将所有的元素都置为 0，这个操作是消耗内存带宽的。CachedBufferAllocator 是针对这种情况使用的，但是在大多数情况下，你还是应该使用默认的 SimpleBufferAllocator。

三、启动和配置

IoService 的配置被简化了。

在 1.x 版本中，有很多种方式来配置 IoService 和它的子接口(例如 IoAcceptor 和 IoConnector)。基本上，有两种配置方法：
在调用 bind() 或 connect()时，具体指定一个 IoServiceConfig
Java 代码

```
1. SocketAcceptor acceptor = new SocketAcceptor();
2. SocketAcceptorConfig myServiceConfig = new SocketAcceptorConfig
   ();
3. myServiceConfig.setReuseAddress(true);
4. acceptor.bind(myHandler, myServiceConfig);
```

使用 IoService.defaultConfig 属性，此时不需要指定一个 IoServiceConfig
Java 代码

```
1. SocketAcceptor acceptor = new SocketAcceptor();
2. acceptor.getDefaultConfig().setReuseAddress(true);
3. acceptor.bind(new InetSocketAddress(8080), myHandler);
```

配置 IoFilterChain 是另一个令人头痛的问题，因为除了 IoServiceConfig 内的 IoFilterChainBuilder 外，还有一个全局的 IoFilterChainBuilder，这就意味着使用两个 IoFilterChainBuilders 来配置一个 IoFilterChain。大多数用户使用全局的 IoFilterChainBuilder 来配置 IoFilterChain，并且这就足够了。
针对这种复杂情况，MINA 2.0 简化了网络应用程序的启动，请比较下面的代码与前面代码的不同
Java 代码

```
1. SocketAcceptor acceptor = new SocketAcceptor();
2. acceptor.setReuseAddress(true);
3. acceptor.getFilterChain().addLast("myFilter1", new MyFirstFilter
   ());
4. acceptor.getFilterChain().addLast("myFilter2", new MySecondFilter
   ());
```

```
5. acceptor.getSessionConfig().setTcpNoDelay(true);
6.
7. // You can specify more than one addresses to bind to multiple addre
   sses or interface cards.
8. acceptor.setLocalAddress(new InetSocketAddress(8080));
9. acceptor.setHandler(myHandler);
10.
11. acceptor.bind();
12.
13. // New API restricts one bind per acceptor, and  you can't bind mor
   e than once.
14. // The following statement will raise an exception.
15. acceptor.bind();
```

你也许意识到与 Spring 框架整合也将变得更加简单。

四、线程

ThreadModel 被移除了。

最初引入 ThreadModel 的概念为的是简化一个 IoService 预定义的线程模式。然而，配置线程模式却变得非常简单以至于不能引入新的组建。与其易用性相比，线程模式带了更多的混乱。在 2.x 中，当你需要的时候，你必须明确的增加一个 ExecutorFilter。

ExecutorFilter 使用一个特定的 Executor 实现来维系事件顺序。
在 1.x 中，可以使用任意的 Executor 实现来来维系事件顺序，但 2.x 提供了两个新的 ThreadPoolExecutor 实现，OrderedThreadPoolExecutor 和 UnorderedThreadPoolExecutor，ExecutorFilter 维系事件顺序，当以下两种情况：当使用默认构造方法时，ExecutorFilter 创建一个 OrderedThreadPoolExecutor，或者
明确指明使用 OrderedThreadPoolExecutor 时
OrderedThreadPoolExecutor 和 UnorderedThreadPoolExecutor 内部使用了一些架构来防止发生 OutOfMemoryError，所以你应该尽量使用这两个类而不是其他 Executor 的实现。

五、协议编解码

DemuxingProtocolCodecFactory 被重写了。

新增了 DemuxingProtocolEncoder 和 DemuxingProtocolDecoder 两个类，
DemuxingProtocolCodecFactory 只是这两个类的外壳。register() 方法被重命名为 addMessageEncoder() 和 addMessageDecoder()，这个变化使混合使用多个 encoders 和 decoders 变得更加自由。
MessageEncoder 接口也发生了改变，MessageEncoder.getMessageTypes()被移除了，当你调用 addMessageEncoder()，你只需要指明信息的类型，encoder 就可以进行正确的编码了。

六、集成

JMX 集成被重新设计了。
Sping 集成被简化了。
七、其他方面的改变
TransportType 更名为 TransportMetadata。
TransportType 改名是因为它的角色是元数据而不仅仅是一种枚举。
IoSessionLogger 被重新设计了。
IoSessionLogger 现在实现了 SLF4J Logger 接口，所以你可以像声明简单 SLF4J logger 实例一样声明它，这个变化使你不必向其他不必要的部分暴露 IoSessionLogger 对象。另外，在使用 MDC 时，请考虑使用简单的 MdcInjectionFilter，这时 IoSessionLogger 是没有必要的。
改变之前：
Java 代码

```
1. IoSessionLogger.debug(session, ...);
```

改变之后：
Java 代码

```
1. Logger logger = IoSessionLogger.getLogger(session);
2. logger.debug(...);
```

BroadcastIoSession 被合并到 IoSession 中。
ReadThrottleFilterBuilder 被 ReadThrottleFilter 替代并最终移除。

MINA2 官方教程翻译(2) 快速上手指南

一、介绍

该教程通过构建一个 time server，带你走进给予 MINA 的应用程序开发的大门，但在开始之前我们需要具备下面的必要条件：

- MINA 2.x 的核心包
- JDK 1.5 或更高版本
- SLF4J 1.3.0 或更高版本

1. Log4J 1.2 的用户：slf4j-api.jar, slf4j-log4j12.jar, and Log4J 1.2.x
2. Log4J 1.3 的用户：slf4j-api.jar, slf4j-log4j13.jar, and Log4J 1.3.x
3. java.util.logging 的用户：slf4j-api.jar and slf4j-jdk14.jar

注意：请务必确认你所使用的 slf4j-*.jar 要与你的日志框架相匹配。例如，slf4j-log4j12.jar 和 log4j-1.3.x.jar 不能在一起使用，否则会引起混乱。
我已经在 Windows? 2000 professional 和 linux 平台上测试了这个程序，如果你在运行这个程序的过程中遇到了问题，请立即联系我们的开发人员。
当然，这个程序是与开发环境(IDE, editors 等等)无关的，它可以在任何你熟悉的平台中运行。另外，为了简化，编译命令与运行脚本都没有体现，如果你需要学习如何编译并运行 java 程序，请参考 Java tutorial。

二、编写基于 MINA 框架的 time server

我们从创建一个名为 MinaTimeServer.java 的文件开始，最初的代码如下：

Java 代码

```
1. public class MinaTimeServer {
2.
3.     public static void main(String[] args) {
4.         // code will go here next
5.     }
6. }
```

对所有人来说，这段代码应该是简单易懂的，我们只是简单的定义了一个 main 方法是这个程序能够正常运行起来。从现在开始，我们将逐步加入代码是其最终成为一个可用的 server。首先，我们需要一个可以监听连接到来的对象，既然我们的程序是基于 TCP/IP 的，所以我们在程序中加入一个 SocketAcceptor。

Java 代码

```
1. import org.apache.mina.core.service.IoAcceptor;
2. import org.apache.mina.transport.socket.nio.NioSocketAcceptor;
3.
4. public class MinaTimeServer
5. {
6.
7.     public static void main( String[] args )
8.     {
9.         IoAcceptor acceptor = new NioSocketAcceptor();
10.     }
11.
12. }
```

加入 NioSocketAcceptor 之后，我们可以继续定义一个 handler 类，并将其与 NioSocketAcceptor 绑定到一个端口上。

下面，我们在配置中增加一个 filter，这个 filter 将把二进制数据或是协议相关的数据转换成为一个消息对象，反之亦然。我们使用现有的 TextLine 工厂类，以为它可以处理基于文本的信息(你不需要自己来实现编解码部分)。

Java 代码

```
1. import java.nio.charset.Charset;
2.
3. import org.apache.mina.core.service.IoAcceptor;
4. import org.apache.mina.filter.codec.ProtocolCodecFilter;
5. import org.apache.mina.filter.codec.textline.TextLineCodecFactor
   y;
6. import org.apache.mina.filter.logging.LoggingFilter;
7. import org.apache.mina.transport.socket.nio.NioSocketAcceptor;
8.
```

```
9. public class MinaTimeServer
10. {
11.     public static void main( String[] args )
12.     {
13.         IoAcceptor acceptor = new NioSocketAcceptor();
14.
15.         acceptor.getFilterChain().addLast( "logger", new LoggingFilt
    er() );
16.         acceptor.getFilterChain().addLast( "codec", new ProtocolCode
    cFilter( new TextLineCodecFactory( Charset.forName( "UTF-8" ))));
17.     }
18. }
```

然后，我们定义一个 handler，这个 handler 将对客户端的连接以及过去当前时间的请求做出服务。handler 类必须实现 IoHandler 接口。对于大多数基于 MINA 的应用程序，这个操作无疑是一个很大的负担，因为它将处理客户端说有的请求。在这个教程中，我们的 handler 将继承自 IoHandlerAdapter，这个类依照适配器模式来简化实现 IoHandler 接口所带来的代码量。

Java 代码

```
1. import java.io.IOException;
2. import java.nio.charset.Charset;
3.
4. import org.apache.mina.core.service.IoAcceptor;
5. import org.apache.mina.filter.codec.ProtocolCodecFilter;
6. import org.apache.mina.filter.codec.textline.TextLineCodecFactor
    y;
7. import org.apache.mina.filter.logging.LoggingFilter;
8. import org.apache.mina.transport.socket.nio.NioSocketAcceptor;
9.
10. public class MinaTimeServer
11. {
12.     public static void main( String[] args ) throws IOException
13.     {
14.         IoAcceptor acceptor = new NioSocketAcceptor();
15.
16.         acceptor.getFilterChain().addLast( "logger", new LoggingFilt
    er() );
17.         acceptor.getFilterChain().addLast( "codec", new ProtocolCode
    cFilter( new TextLineCodecFactory( Charset.forName( "UTF-8" ))));
18.
19.         acceptor.setHandler(  new TimeServerHandler() );
20.     }
21. }
```

现在，我们在 NioSocketAcceptor 增加一些 Socket 相关的配置：

Java 代码

```
1. import java.io.IOException;
2. import java.nio.charset.Charset;
3.
4. import org.apache.mina.core.session.IdleStatus;
5. import org.apache.mina.core.service.IoAcceptor;
6. import org.apache.mina.filter.codec.ProtocolCodecFilter;
7. import org.apache.mina.filter.codec.textline.TextLineCodecFactor
   y;
8. import org.apache.mina.filter.logging.LoggingFilter;
9. import org.apache.mina.transport.socket.nio.NioSocketAcceptor;
10.
11. public class MinaTimeServer
12. {
13.     public static void main( String[] args ) throws IOException
14.     {
15.         IoAcceptor acceptor = new NioSocketAcceptor();
16.
17.         acceptor.getFilterChain().addLast( "logger", new LoggingFilt
   er() );
18.         acceptor.getFilterChain().addLast( "codec", new ProtocolCode
   cFilter( new TextLineCodecFactory( Charset.forName( "UTF-8" ))));
19.
20.         acceptor.setHandler(  new TimeServerHandler() );
21. idle sessions
22.         acceptor.getSessionConfig().setReadBufferSize( 2048 );
23.         acceptor.getSessionConfig().setIdleTime( IdleStatus.BOTH_IDL
   E, 10 );
24.     }
25. }
```

在 MinaTimeServer 增加了两行新的内容，这些 set 方法分别设置了 IoHandler、input buffer size 和 session 对象上的 idle 属性。buffer size 指明了底层操作系统应该给与新到来的数据分配多少空间；第二行指明了什么时候应该检测 idle sessions。在 setIdleTime 这个方法中，第一参数指明了在检测 session 是否 idle 时，应该关心那一种活动，第二个参数指明了 session 变为 idle 状态时需要经过多长的时间。

handler 的代码如下：

Java 代码

```
1. import java.util.Date;
2.
3. import org.apache.mina.core.session.IdleStatus;
4. import org.apache.mina.core.service.IoHandlerAdapter;
5. import org.apache.mina.core.session.IoSession;
6.
```

```
7. public class TimeServerHandler extends IoHandlerAdapter
8. {
9.     @Override
10.     public void exceptionCaught( IoSession session, Throwable caus
    e ) throws Exception
11.     {
12.         cause.printStackTrace();
13.     }
14.
15.     @Override
16.     public void messageReceived( IoSession session, Object messag
    e ) throws Exception
17.     {
18.         String str = message.toString();
19.         if( str.trim().equalsIgnoreCase("quit") ) {
20.             session.close();
21.             return;
22.         }
23.
24.         Date date = new Date();
25.         session.write( date.toString() );
26.         System.out.println("Message written...");
27.     }
28.
29.     @Override
30.     public void sessionIdle( IoSession session, IdleStatus statu
    s ) throws Exception
31.     {
32.         System.out.println( "IDLE " + session.getIdleCount( statu
    s ));
33.     }
34. }
```

该类用到的方法有 exceptionCaught、messageReceived 和 sessionIdle。在 handler 中，一定要定义 exceptionCaught 方法，该方法用来处理在远程连接中处理过程中发生的各种异常，如果这个方法没有被定义，我们可能不能发现这些异常。

在这个 handler 中，exceptionCaught 方法只是简单地打印出异常堆栈信息并关闭连接，对于大多数程序来说，这是一种比较标准的操作，除非连接可以在异常条件下恢复。

messageReceived 方法会接收客户端的数据并返回当前的的系统时间，如果从客户端接收到了消息‘quit’，则 session 会被关闭。与调用 session.write(Object)的情况相同，不同的协议编解码器决定了传入该方法的对象(第二个参数)也是不同的。如果你没有指定协议编解码器，你最有可能接收到一个 IoBuffer 对象，当然，调用 session.write(Object)也是一个 IoBuffer 对象。

当 session 持续 idle 的时间与 acceptor.getSessionConfig().setIdleTime(IdleStatus.BOTH_IDLE, 10)设置的时间一致时，sessionIdle 方法将被调用。

现在剩下的工作只是定义一个 server 监听的地址和端口了，当然还需要启动服务。代码如下：

正如你所见，这里调用了 acceptor.setLocalAddress(new InetSocketAddress(PORT));方法，该方法指明了 server 将在哪个 IP 和端口上监听。最后一步调用了 IoAcceptor.bind()方法，该方法将端口与具体的客户端进程绑定在一起。

三、验证 Time server

现在，我们编译上面的程序，编译完成后就可以运行并查看运行结果了。最简单的测试途径就是启动程序，并使用 telnet 与之建立连接：

Client Output	Server Output
user@myhost:~> telnet 127.0.0.1 9123 Trying 127.0.0.1... Connected to 127.0.0.1. Escape character is '^]'. hello Wed Oct 17 23:23:36 EDT 2007 quit Connection closed by foreign host. user@myhost:~>	MINA Time server started. Message written...

四、接下来

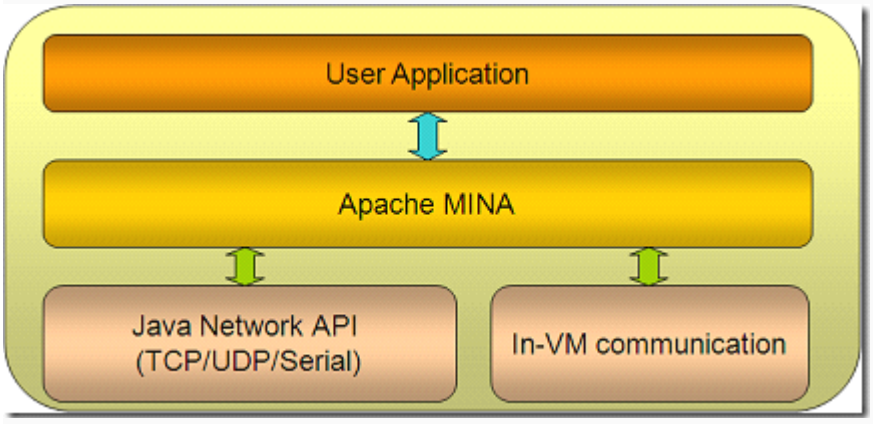
获取更多资源，请浏览 MINA 的 Documentation。你也可以阅读其他教程。

MINA2 官方教程翻译(3) MINA 的应用程序架构

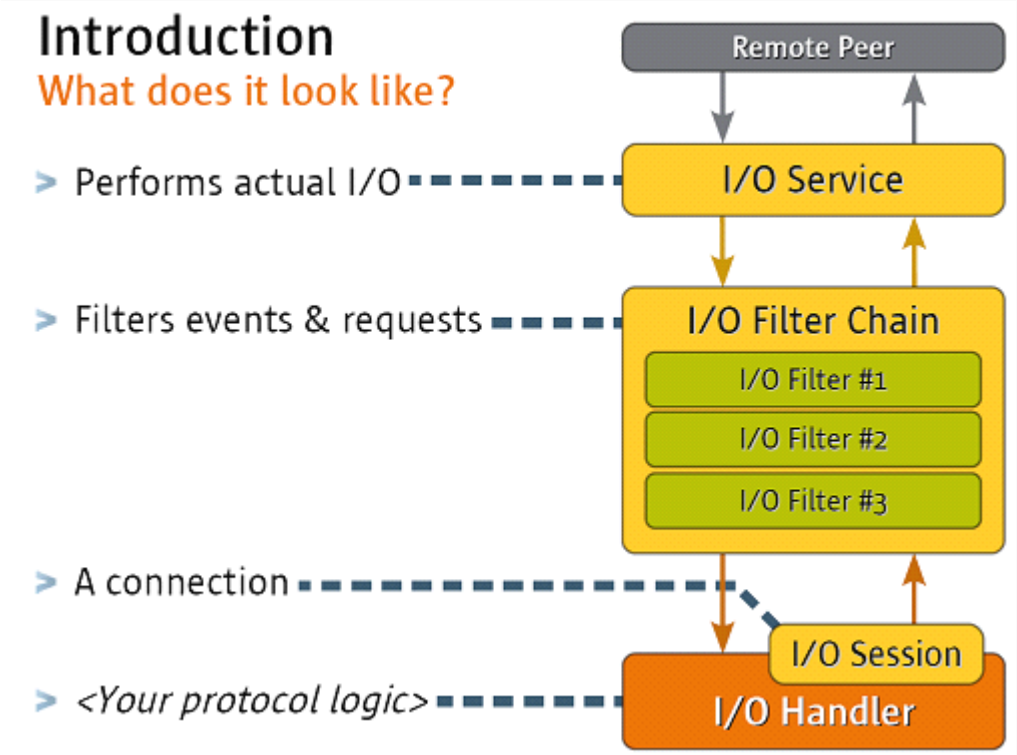
文章分类:Java 编程

一、简介

有个问题经常被提出：基于 MINA 的应用程序应该是什么样的呢？这篇文章将给出一个答案。我们已经收集了很多基于 MINA 的描述信息。下面是架构图：



让我们在来关于一下细节



这张图片选取自 Trustin Lee 在 JavaOne 2008 上的报告"Rapid Network Application Development with Apache MINA“

从广义上讲，基于 MINA 的应用程序分为 3 层

- I/O Service - 完成实际的 I/O 操作
- I/O Filter Chain - 将字节过滤或转换成为预想的数据结构，反之亦然
- I/O Handler - 完成实际的业务逻辑操作

那我们如何创建一个基于 MINA 的应用程序呢？

1. Create I/O service - 从现有的 Services (*Acceptor)中选择一个或者创建自己的
2. Create Filter Chain - 从现有的 Filters 中选择或者创建一个传输 request/response 的自定义 Filter
3. Create I/O Handler - 编写业务逻辑，处理不同的报文

创建 MINA 程序就如上文所述的一样。

MINA2 官方教程翻译(4) 日志配置

一、背景

MINA 框架允许开发人员在编写基于 MINA 的应用程序时使用自己熟悉的日志系统。

二、SLF4J

MINA 框架使用 Simple Logging Facade for Java (SLF4J)。你可以在这里 获取到更多关于 SLF4J 的信息，这种日志系统兼容各种日志系统的实现。你可能会使用 log4j、java.util.logging

或其他的日志系统，使用这种日志框架的好处在于如果你在开发过程中，将日志系统从 java.util.logging 改为 log4j，你根本需要修改你的代码。
选择正确的 jar 包

Logging framework	Required JARs
Log4J 1.2.x	slf4j-api.jar ， slf4j-log4j12.jar
Log4J 1.3.x	slf4j-api.jar ， slf4j-log4j13.jar
java.util.logging	slf4j-api.jar ， slf4j-jdk14.jar
Commons Logging	slf4j-api.jar ， slf4j-jcl.jar

下面几点还需要注意：

- 对于任意一种日志系统，slf4j-api.jar 是必须的；
- 重要：在 classpath 上不能放置多于一个日志系统实现 jar 包(例如 slf4j-log4j12.jar and slf4j-jdk14.jar)，这将导致日志出席不可预知的行为；
- slf4j-api.jar 和 slf4j-<impl>.jar 的版本应该是一致的。

如果 SLF4J 配置正确，你可以继续配置你真正使用的日志系统(例如修改 log4j.properties)。

重载 Jakarta Commons Logging

SLF4J 提供了一种机制可以使现有的应用程序从使用 Jakarta Commons Logging 变更为 SLF4J 而不需要修改代码，只需要将 commons-loggong JAR 文件充 classpath 中除去，并将 jcl104-over-slf4j.jar 加入到 classpath 中。

三、log4j 范例
我们以 log4j 为例，然后将下面的代码片段加入到 log4j.properties 中：

Properties 代码

```
1. # Set root logger level to DEBUG and its only appender to A1.
2. log4j.rootLogger=DEBUG, A1
3.
4. # A1 is set to be a ConsoleAppender.
5. log4j.appender.A1=org.apache.log4j.ConsoleAppender
6.
7. # A1 uses PatternLayout.
8. log4j.appender.A1.layout=org.apache.log4j.PatternLayout
9. log4j.appender.A1.layout.ConversionPattern=%-4r [%t] %-5p %c{1} %
   x - %m%n
```

我们将这个文件放置在工程的 src 目录中，如果你使用 IDE，当你测试代码是，你实际上是想把这个文件放置在 classpath 上。

注意：这里只是在 IoAcceptor 上设置了日志，但 slf4j 可以在程序中广泛使用，有了它的帮助，你可以根据需要获取到有用的信息。

下面我们编写一个简单的 server 从而生成一些日志信息，这里我们使用 EchoServer 的范例工程来增加日志：

Java 代码

```
1. public static void main(String[] args) throws Exception {
2.         IoAcceptor acceptor = new SocketAcceptor();
3.         DefaultIoFilterChainBuilder chain = acceptor.getFilterChain
   ();
4.
5.         LoggingFilter loggingFilter = new LoggingFilter();
6.         chain.addLast("logging", loggingFilte
   r);
7.
8.         acceptor.setLocalAddress(new InetSocketAddress(PORT));
9.         acceptor.setHandler(new EchoProtocolHandler());
10.         acceptor.bind();
11.
12.         System.out.println("Listening on port " + PORT);
13. }
```

正如你所见，在 EchoServer 范例中，我们删除了 addLogger 方法并新增加了两行代码。通过 LoggingFilter 的引用，你可以在这里设置与 IoAcceptor 相关的所有事件的日志级别。在这里，可以使用 LoggingFilter 中的 setLogLevel(IoEventType, LogLevel)方法来区分触发 IoHandler 日志的时间以及对应的日志级别，下面是这个方法选项：

IoEventType	Description
SESSION_CREATED	一个新的 session 被创建时触发
SESSION_OPENED	一个新的 session 打开时触发
SESSION_CLOSED	一个 session 被关闭时触发
MESSAGE_RECEIVED	接收到数据时触发
MESSAGE_SENT	数据被发送后触发
SESSION_IDLE	一个 session 空闲了一定时间后触发
EXCEPTION_CAUGHT	当有异常抛出时触发

下面是日志级别的描述：

LogLevel	Description
NONE	无论如何配置，日志都不会产生
TRACE	在日志系统中创建一个 TRACE 事件
DEBUG	在日志系统中生成 debug 信息

INFO	在日志系统中生成提示信息
WARN	在日志系统中生成警告信息
ERROR	在日志系统中生成错误信息

根据这些信息，你应该可以扩展这些范例来构建一个使用日志的简单系统，这些日志将为你提供有用的信息。

MINA2 官方教程翻译(5) 基本概念之 IoBuffer

文章分类:Java 编程

简介

IoBuffer 是 MINA 应用程序中使用的一种字节缓冲区，它是 JDK 中 ByteBuffer 类的替代品。MINA 框架出于下面两个原因没有直接使用 JDK 中 nio 包内的 ByteBuffe：

- 没有提供可用的 getters 和 putters 方法，例如 fill, get/putString, 和 get/putAsciiInt()；
- 由于它的容量是固定的，所以不利于存储变长数据。

MINA 3 将改变这种情况。MINA 框架对 nio ByteBuffer 做了一层封装的最主要原因是希望能够拥有一种可扩展的缓冲区。这并不是一个很好的决定。缓冲区就是缓冲区：一个用于存储临时数据的临时空间，直到这些数据被使用。其实还有些其他的解决方案，例如可以对一组 nio ByteBuffer 进行包装来避免数据从一个缓冲区向两个容量更大的缓冲区复制，从而得到一个容量可扩展的缓冲区。
或许在 filter 之间传递数据时使用 InputStrea 来代替字节缓冲区会更加舒适，因为这不需要提供一种可以存储数据的特性，这种数据结构可以使字节数组、字符串或者其他类型的消息等等。最后，但并非最不重要的一点是，当前的实现并没有达成一个目标：零拷贝策略(例如当我们从 socket 中读取了一些数据，我们希望避免持续的数据拷贝)。如果我们使用了可以扩展的字节缓冲区，那么我们只需要在管理大数据消息时进行数据拷贝 。请记住 MINA ByteBuffer 只不过是 NIO ByteBuffer 的顶层封装，当我们使用 direct buffers 时，很可能是一个很严重的问题。

IoBuffer 操作

分配一个新的 Buffer

IoBuffer 是一个抽象类，所以它不能直接被实例化。分配 IoBuffer，我们可以使用两种 allocate()方法。
Java 代码

```
1.  // Allocates a new buffer with a specific size, defining its type (d
    irect or heap)
2.  public static IoBuffer allocate(int capacity, boolean direct)
3.
4.  // Allocates a new buffer with a specific size
5.  public static IoBuffer allocate(int capacity)
```

allocate()方法是用一个或两个参数。第一种形式使用两个参数：

- **capacity** - buffer 的容量
- **direct** -buffer 的类型。***true*** 意味着得到一个 direct buffer，***false*** 意味着得到一个 heap buffer

默认的 buffer 分配是由 SimpleBufferAllocator 处理的。
可选的，下面的形式也可以使用：
Java 代码

```
1.  IoBuffer buffer = IoBuffer.allocate(8);
2.  buffer.setAutoExpand(true);
3.
4.  buffer.putString("12345678", encoder);
5.
6.  // Add more to this buffer
7.  buffer.put((byte)10);
```

按照上面的例子，如果数据的长度大于 8byte 的话，IoBuffe 会根据情况重新分配其内置的 ByteBuffer，它的容量会被加倍，它的 limit 会增长到 String 被写入时的最后 position。这种行为与 StringBuffer 工作的方式十分类似。
注意：这种程序结构在 MINA3.0 时会被废弃，因为这并不是增长 buffer 容量的最好方式。这种方式很可能被一种类似 InputStream 的方式所替代，在 InputStream 的背后很可能是一组固定长度的 ByteBuffers。

创建自动收缩的 Buffer

为了节省内存，在有些情形下我们需要释放被额外分配的内存，IoBuffer 提供了 autoShrink 属性来达到此目的。如果 autoShrink 属性被打开，当 compact()方法被调用时，IoBuffer 回将部分的回收其容量，只使用四分之一或是更少的容量。如果需要手动控制收缩行为，请使用 shrink()方法。
让我们实践一下：
Java 代码

```
1.  IoBuffer buffer = IoBuffer.allocate(16);
2.  buffer.setAutoShrink(true);
3.  buffer.put((byte)1);
4.  System.out.println("Initial Buffer capacity = "+buffer.capacity
    ());
5.  buffer.shrink();
6.  System.out.println("Initial Buffer capacity after shrink = "+buffer.
    capacity());
7.
8.  buffer.capacity(32);
9.  System.out.println("Buffer capacity after incrementing capacity to 3
    2 = "+buffer.capacity());
10. buffer.shrink();
```

```
11.  System.out.println("Buffer capacity after shrink= "+buffer.capacity
     ());
```

我们初始化分配一个容量为 16 的 buffer，并将自动收缩设置为 true。

让我们看一下输出的结果：

Java 代码

```
1.  Initial Buffer capacity = 16
2.  Initial Buffer capacity after shrink = 16
3.  Buffer capacity after incrementing capacity to 32 = 32
4.  Buffer capacity after shrink= 16
```

让我们分析一下输出：

- 初始化 buffer 的容量为 16，因为我们使用 16 指定了该 buffer 的容量，16 也就成了该 buffer 的最小容量
- 调用 shrink()方法后，容量仍旧为 16，所以无论怎么调用紧缩方法，容量都不好小于其初始容量
- 增加该 buffer 的容量至 32，该 buffer 的容量达到 32
- 调用 shrink()方法，容量回收至 16，从而剔除了冗余的容量

再次强调，这种方式是一种默认的行为，我们不需要明确指明一个 buffer 是否能被收缩。

Buffer 分配

IoBufferAllocater 负责分配并管理 buffer，如果你希望使用你的方式精确控制分配行为，请自己实现 *IoBufferAllocater* 接口。

MINA 提供了 IoBufferAllocater 的两种实现，如下：

- **SimpleBufferAllocator (默认)** - 每次创建一个新的 buffer
- **CachedBufferAllocator** - 缓存 buffer，使 buffer 在扩展时可以被重用

注意：在新版本的 JVM 中，使用 cached IoBuffer 并不能明显提高性能。

你可以自己实现 IoBufferAllocator 接口并在 IoBuffer 上调用 setAllocator()方法来指定使用你的实现。

MINA2 官方教程翻译(6) 基本概念之 IoHandler

文章分类:Java 编程

简介

Handler 用来处理 MINA 触发的 I/O 事件。IoHandler 是一个核心接口，它定义了 Filter 链末端需要的所有行为。IoHandler 接口包含以下方法：

- sessionCreated
- sessionOpened
- sessionClosed
- sessionIdle
- exceptionCaught

- messageReceived
- messageSent

sessionCreated 事件

一个新的 connection 被创建时，会触发 SessionCreated 事件。对于 TCP 来说，这个事件代表连接的建立；对于 UDP 来说，它代表收到了一个 UDP 数据包。这个方法可以用作初始化 session 的各种属性，也可以用来在一个新建的 connection 上触发一些一次性的行为。
I/O processor 线程会调用这个方法，所以在实现该方法时，只加入一些耗时较少的操作，因为 I/O processor 线程是用来处理多会话的。

sessionOpened 事件

当一个 connection 打开时会触发 sessionOpened 事件，这个事件永远在 sessionCreated 之后触发。如果配置了线程模式，那么这个方法会被非 I/O processor 线程调用。

sessionClosed 事件

当一个 session 关闭的时候会触发 sessionClosed 事件。可以将 session 的清理操作放在这个方法里进行。

sessionIdle 事件

当一个 session 空闲的时候会触发 sessionIdle 事件。当使用 UDP 时该方法将不会被调用。

exceptionCaught 事件

当用户代码或 MINA 框架抛出异常时，会触发事件事件。如果该异常是一个 IOException，那么 connection 会被关闭。

messageReceived 事件

当接收到消息的时候会触发 messageReceived 事件。所有的业务处理代码应该写在这里，但要留心你所要的消息类型。

messageSent 事件

当消息已被远端接收到的时候，会触发 messageSent 事件(调用 IoSession.write()发送消息)。

MINA2 官方教程翻译(7)传输特性之串口

文章分类:Java 编程
使用 MINA2.0，你可以像编写基于 TCP/IP 的程序那样编写基于串口的程序。

获得 MINA2.0

MINA 2.0 的最终版本还没有 release，但是你可以下载最新的版本。如果你希望从 trunk 构建代码，可以参考开发者指南。

前提

在访问串口之前，Java 应用程序需要一个 native 库。MINA 使用 ftp://ftp.qbang.org/pub/rxtx/rxtx-2.1-7-bins-r2.zip，请把它放到你的 JDK 或 JRE 的 lib/i386/ 下，并在程序启动的命令行中加入-Djava.library.path=来指定你的 native 库的位置。

连接到串口

串口通讯通过 IoConnector 来实现，这是有通讯媒介的点对点特性来决定的。我们假定你已经通过 MINA 的教程了解到了 IoConnector 的相关知识。连接到串口需要 SerialConnector：

Java 代码

```
1.  // create your connector
2.  IoConnector connector = new SerialConnector()
3.  connector.setHandler( ... here your buisness logic IoHandle
    r ... );
```

与 SocketConnector，并没有什么不同。让我们创建一个地址来连接串口：

Java 代码

```
1.  SerialAddress portAddress=new SerialAddress( "/dev/ttyS0", 3840
    0, 8, StopBits.BITS_1, Parity.NONE, FlowControl.NONE );
```

第一个参数代表串口的标识符。对于 Windows 系统，串口一般叫做"COM1"、"COM2"以此类推，对于 Linux 或者一些 Unix 系统，通常由"/dev/ttyS0"、"/dev/ttyS1"、"/dev/ttyUSB0"来表示。

剩下的参数取决于你的硬件设备的连接特性。

- 波特率
- 数据位数
- 奇偶校验
- 流控制机制

当这些都具备，就可以连接到该地址：

Java 代码

```
1.  ConnectFuture future = connector.connect( portAddress );
2.  future.await();
3.  IoSession sessin = future.getSession();
```

其他的事情和使用 TCP 协议等一样，你可以加入你的 filters 和 codecs。

MINA2 官方教程翻译(8)传输特性之 UDP

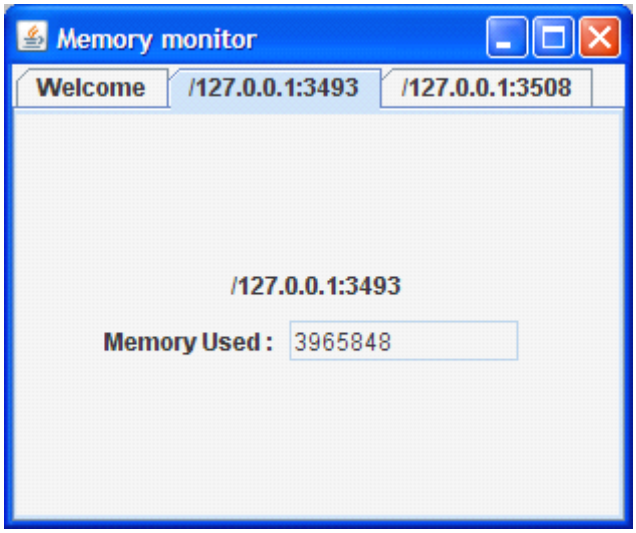
文章分类:Java 编程

该教程可以帮助你使用 MINA 框架编写基于 UDP 的 Socket 应用程序。在这篇教程中，我们将编写一个 server 端程序，server 可以通过连接该程序来展现 client 端程序的内存使用情况。现实中的很多程序都已经具备与该程序类似的功能，可以监控程序来内存使用情况。

构建代码

MINA 2.0 的最终版本还没有 release，但是你可以下载最新的版本。如果你希望从 trunk 构建代码，可以参考开发者指南。

应用程序



如上图所示，该 server 在端口 18567 监听，client 端连接 server 端并向 server 端发送内存使用数据。在上图所展示应用中，共有两个 client 连接到了 server。

程序执行流程

1. server 在 18567 端口监听客户端请求；
2. client 连接 server 端，server 会在 Session Created 事件处理时增加一个 Tab 页；
3. clients 向 server 发送内存使用数据；
4. server 端把接收到的数据展现在 Tab 上。

Server 端代码分析

我们可以在 MINA 的示例代的 org.apache.mina.example.udp 包下找到这些代码。对于每个实例，我们只需要关系 MINA 相关的代码。

要想构建该 server，我们需要做如下操作：

1. 创建一个数据报 socket，监听 client 端请求。（请参考 MemoryMonitor.java）；
2. 创建一个 IoHandler 来处理 MINA 框架生成的各种事件。（请参考 MemoryMonitorHandler.java）。

第一步可以通过该程序片来实现：

Java 代码

```
1.  NioDatagramAcceptor acceptor = new NioDatagramAcceptor();
2.  acceptor.setHandler(new MemoryMonitorHandler(this));
```

这里，我们创建一个 NioDatagramAcceptor 来监听 client 端请求，并且设置它的 IoHandler. 变量"PORT"是一个 int 值。下一步我们家 filter 链中加入了一个 logging filter。LoggingFilter 是一个非常不错的记录日志的方式，它可以在很多节点处生成日志信息，这些热值能够展现出 MINA 框架的工作方式。

Java 代码

```
1.  DefaultIoFilterChainBuilder chain = acceptor.getFilterChain();
2.  chain.addLast("logger", new LoggingFilter());
```

下一步我们深入一下 UDP 传输所特有的代码。我们设置 acceptor 可以重用 address。

Java 代码

```
1.  DatagramSessionConfig dcfg = acceptor.getSessionConfig();
2.  dcfg.setReuseAddress(true);acceptor.bind(new InetSocketAddress(POR
    T));
```

当然，最后一件事情是调用 bind()方法来绑定端口。

IoHandler 实现

Server 端主要关心如下三个事件：

- Session 创建
- Message 接收
- Session 关闭

详细代码如下：

Session Created Event

Java 代码

```
1.  @Override
2.  public void sessionCreated(IoSession session) throws Exception {
3.       SocketAddress remoteAddress = session.getRemoteAddress();
4.       server.addClient(remoteAddress);
5.  }
```

在 session 创建事件中，我们调用 addClient()方法来增加一个 Tab 页。

Message Received Event

Java 代码

```
1.  @Override
2.  public void messageReceived(IoSession session, Object message) throw
    s Exception {
```

```
        if (message instanceof IoBuffer) {
            IoBuffer buffer = (IoBuffer) message;
            SocketAddress remoteAddress = session.getRemoteAddress
    ();
            server.recvUpdate(remoteAddress, buffer.getLong());
        }
    }
```

在 message 接收事件中，我们从接收到的消息中得到所关心的内存使用的数据；应用程序还需要发送响应信息。在这个方法中，处理消息和发送响应都是通过 session 完成的。

Session Closed Event

Java 代码

```
@Override
public void sessionClosed(IoSession session) throws Exception {
    System.out.println("Session closed...");
    SocketAddress remoteAddress = session.getRemoteAddress();
    server.removeClient(remoteAddress);
}
```

在 session 关闭事件中，我们需要删除对应的 Tab 页。

Client 端代码分析

在这一节，我们将解释一下客户端代码。实现客户端我们需要进行如下操作：

- 创建 Socket 并连接到 server 端
- 设置 IoHandler
- 收集内存使用信息
- 发送数据到 server 端

我们从 MemMonClient.java 开始，可以在 org.apache.mina.example.udp.client 包下找到它。开始的几行代码非常简单：

Java 代码

```
connector = new NioDatagramConnector();
connector.setHandler( this );
ConnectFuture connFuture = connector.connect( new InetSocketAddress
("localhost", MemoryMonitor.PORT ));
```

这里我们创建了一个 NioDatagramConnector，设置了它的 handler 并且连接到 server 端。我们必须在 InetSocketAddress 对象中设定 host，否则程序不能正常运行。该程序是在 Windows XP 环境下开发运行的，所以与其他环境可能存在差别。下一步我们等待确认 client 已经连接到 server 端，一旦连接建立，我们可以开始向 server 端发送数据。代码如下：

Java 代码

```
connFuture.addListener( new IoFutureListener(){
    public void operationComplete(IoFuture future) {
        ConnectFuture connFuture = (ConnectFuture)future;
        if( connFuture.isConnected() ){
```

```
5.                 session = future.getSession();
6.                 try {
7.                     sendData();
8.                 } catch (InterruptedException e) {
9.                     e.printStackTrace();
10.                }
11.            } else {
12.                log.error("Not connected...exiting");
13.            }
14.        }
15.    });
```

这里我们在 ConnectFuture 对象中加入一个 listener，当 client 连接到 server 端时，operationComplete 方法将被回调，这时我们开始发送数据。我们通过调用 sendData 方法向 server 端发送数据，该方法如下：

Java 代码

```
1.  private void sendData() throws InterruptedException {
2.      for (int i = 0; i < 30; i++) {
3.          long free = Runtime.getRuntime().freeMemory();
4.          IoBuffer buffer = IoBuffer.allocate(8);
5.          buffer.putLong(free);
6.          buffer.flip();
7.          session.write(buffer);
8.
9.          try {
10.             Thread.sleep(1000);
11.         } catch (InterruptedException e) {
12.             e.printStackTrace();
13.             throw new InterruptedException(e.getMessage());
14.         }
15.     }
16. }
```

该方法会在 30 秒内每秒向 server 端发送当前的剩余内存 Cincinnati。你可以看到我们分配了一个足够大的 IoBuffer 来装载一个 long 型的变量。最后这个 buffer 被 flipped 并发送至 server 端。

MINA2 官方教程翻译(9)传输特性之 APR

文章分类:Java 编程

简介

APR(Apache portable Run-time libraries，Apache 可移植运行库)的目的如其名称一样，主要为上层的应用程序提供一个可以跨越多操作系统平台使用的底层支持接口库。MINA 目前也能够支持 APR。本章我们将讨论一下使用 MINA 进行 APR 传输的基本过程。我们使用 Time Server 为例。

前提

APR 传输依赖于下列组件：

- APR 运行库 - 从 http://www.apache.org/dist/tomcat/tomcat-connectors/native/处下载并安装适当版本。
- JNI 封装 - tomcat-apr-5.5.23.jar 包含该 release，将本地库加入路径。

使用 APR 传输

请参考 Time Server 例子的完整代码。基于 NIO 的 Time server 实现如下列代码所示：

Java 代码

```
1.  IoAcceptor acceptor = new NioSocketAcceptor();
2.
3.  acceptor.getFilterChain().addLast( "logger", new LoggingFilter
    () );
4.  acceptor.getFilterChain().addLast( "codec", new ProtocolCodecFilter
    ( new TextLineCodecFactory( Charset.forName( "UTF-8" ))));
5.
6.  acceptor.setHandler(  new TimeServerHandler() );
7.
8.  acceptor.getSessionConfig().setReadBufferSize( 2048 );
9.  acceptor.getSessionConfig().setIdleTime( IdleStatus.BOTH_IDLE, 1
    0 );
10.
11. acceptor.bind( new InetSocketAddress(PORT) );
```

如何使用 APR 传输如下列代码所示：

Java 代码

```
1.  IoAcceptor acceptor = new AprSocketAcceptor();
2.
3.  acceptor.getFilterChain().addLast( "logger", new LoggingFilter
    () );
4.  acceptor.getFilterChain().addLast( "codec", new ProtocolCodecFilter
    ( new TextLineCodecFactory( Charset.forName( "UTF-8" ))));
5.
6.  acceptor.setHandler(  new TimeServerHandler() );
7.
8.  acceptor.getSessionConfig().setReadBufferSize( 2048 );
9.  acceptor.getSessionConfig().setIdleTime( IdleStatus.BOTH_IDLE, 1
    0 );
10.
11. acceptor.bind( new InetSocketAddress(PORT) );
```

我们只是将 NioSocketAcceptor 改为 AprSocketAcceptor，只通过这一个小改动，我们的程序就可以支持 APR 传输，其余的代码与之前都是相同的。

MINA2 官方教程翻译(10)与 Spring 整合

文章分类:Java 编程
我们通过这篇文章来介绍如何与 Spring 框架整合 MINA 应用。

程序结构

我们将编写一个简单的 MINA 应用程序，其组成包括：

- 一个 Handler
- 两个 Filter - Logging Filter 和 ProtocolCodec Filter
- 数据报 Socket

初始化代码

让我们先看一下代码。为了简化，我们做了一些省略。
Java 代码

```
1.  public void initialize() throws IOException {
2.      // Create an Acceptor
3.      NioDatagramAcceptor acceptor = new NioDatagramAcceptor();
4.
5.      // Add Handler
6.      acceptor.setHandler(new ServerHandler());
7.
8.      acceptor.getFilterChain().addLast("logging", new LoggingFilter
    ());
9.      acceptor.getFilterChain().addLast("codec", new ProtocolCodecFilt
    er(new SNMPCodecFactory()));
10.
11.     // Create Session Configuration
12.     DatagramSessionConfig dcfg = acceptor.getSessionConfig();
13.     dcfg.setReuseAddress(true);
14.     logger.debug("Starting Server......");
15.     // Bind and be ready to listen
16.     acceptor.bind(new InetSocketAddress(DEFAULT_PORT));
17.     logger.debug("Server listening on "+DEFAULT_PORT);
18. }
```

整合过程

与 Spring 框架整合，我们需要以下操作：

- 设置 IO handler
- 创建 Filters 并将它们加入到 Filter 链

- 创建 Socket 并设置相关参数

注意：**如同 MINA 之前的 release，最近 release 的版本中并没有 Spring 特定的包，目前这个包叫做 Integration Beans，它用来实现与所有的 DI 框架整合而不仅限于 Spring。**
让我们看一下 Spring 的 xml 文件。我删除了通用部分，只保留了与我们实现整合相关的内容。这个例子脱胎于 MINA 实例中的 Chat 应用，请参考该实例中完整的 xml 文件。现在我们开始整合，首先是定义 IO Handler：
Xml 代码

```
1.  <!-- The IoHandler implementation -->
2.  <bean id="trapHandler" class="com.ashishpaliwal.udp.mina.server.Serv
    erHandler" />
```

然后创建 Filter 链：
Xml 代码

```
1.  <bean id="snmpCodecFilter" class="org.apache.mina.filter.codec.Proto
    colCodecFilter">
2.      <constructor-arg>
3.          bean class="com.ashishpaliwal.udp.mina.snmp.SNMPCodecFactory
    " />
4.      </constructor-arg>
5.  </bean>
6.
7.  <bean id="loggingFilter" class="org.apache.mina.filter.logging.Loggi
    ngFilter" />
8.
9.  <!-- The filter chain. -->
10. <bean id="filterChainBuilder" class="org.apache.mina.core.filterchai
    n.DefaultIoFilterChainBuilder">
11.     <property name="filters">
12.         <map>
13.             <entry key="loggingFilter" value-ref="loggingFilter"/>
14.             <entry key="codecFilter" value-ref="snmpCodecFilter"/>
15.         </map>
16.     </property>
17. </bean>
```

这里，我们创佳了自己的 IoFilter 实例。对于 ProtocolCodec 来说，注入 SNMPCodecFactory 时我们使用了构造注入。Logging Filter 是被直接创建的，没有注入其他属性。一旦我们定义了所有 filters 的 bean 定义，我们就可以将它们组装成 Filter 链。我们定义一个 id 为 FilterChainBuidler 的 bean，然后将定义好的 filter bean 注入其中。万事俱备了，我们只差创建 Socket 并调用 bind()方法。
让我们完成最后一部分，创建 Socket 并使用 Filter 链：
Xml 代码

```
1.  <bean class="org.springframework.beans.factory.config.CustomEditorCo
    nfigurer">
2.      <property name="customEditors">
```

```
3.          <map>
4.              <entry key="java.net.SocketAddress">
5.                  <bean class="org.apache.mina.integration.beans.InetS
   ocketAddressEditor" />
6.              </entry>
7.          </map>
8.      </property>
9.  </bean>
10.
11. <!-- The IoAcceptor which binds to port 161 -->
12. <bean id="ioAcceptor" class="org.apache.mina.transport.socket.nio.Ni
   oDatagramAcceptor" init-method="bind" destroy-method="unbind">
13.     <property name="defaultLocalAddress" value=":161" />
14.     <property name="handler" ref="trapHandler" />
15.     <property name="filterChainBuilder" ref="filterChainBuilder" /
   >
16. </bean>
```

我们创建了 ioAcceptor，注入了 IO handler 和 Filter 链。现在我们需要编写一个方法去读取 Spring 的 xml 文件并启动应用，代码如下：

Java 代码

```
1.  public void initializeViaSpring() throws Exception {
2.      new ClassPathXmlApplicationContext("trapReceiverContext.xml");
3.  }
```

现在我们需要从 main 方法运行程序，MINA 应用便可以初始化。