Mina 状态机介绍

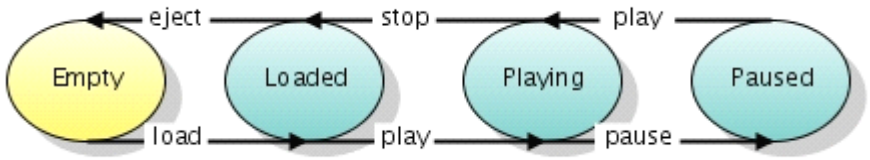
(Introduction to mina-statemachine)

如果你使用 Mina 开发一个复杂的网络应用时，你可能在某些地方会遇到那个古老而又好用的状态模式，来使用这个模式解决你的复杂应用。然而，在你做这个决定之前，你或许想检出 Mina 的状态机的代码，它会根据当前对象的状态来返回对接收到的简短的数据的处理信息。

注意：现在正式发布 Mina 的状态机。因此你要自己在 Mina 的 SVN 服务器上检出该代码，并自己编译，请参考开发指南，来获取更多的关于检出和编译 Mina 源码的信息。Mina 的状态机可以和所有已经发布的版本 Mina 配合使用(1.0.x, 1.1.x 和 当前发布的版本)。

一个简单的例子

让我们使用一个简单的例子来展示一下 Mina 的状态机是如何工作的。下面的图片展示了一个录音机的状态机。其中的椭圆是状态，箭头表示事务。每个事务都有一个事件的名字来标记该事务。



初始化时，录音机的状态是空的。当磁带放如录音机的时候，加载的事件被触发，录音机进入到加载状态。在加载的状态下，退出的事件会使录音机进入到空的状态，播放的事件会使加载的状态进入到播放状态。等等.....我想你可以推断后后面的结果:)

现在让我们写一些代码。外部(录音机中使用该代码的地方)只能看到录音机的接口：

----------START----------

```
public interface TapeDeck {
    void load(String nameOfTape);
    void eject();
    void start();
    void pause();
    void stop();
}
```

----------END----------

下面我们开始编写真正执行的代码，这些代码在一个事务被触发时，会在状态机中执行。首先我们定义一个状态。这些状态都使用字符串常量来定义，并且使用@state 标记来声明。

----------START----------

```
public class TapeDeckHandler {
```

```
    @State public static final String EMPTY   = "Empty";
    @State public static final String LOADED  = "Loaded";
    @State public static final String PLAYING = "Playing";
    @State public static final String PAUSED  = "Paused";
}
```

----------END----------

现在我们已经定义了录音机中的所有状态，我们可以根据每个事务来创建相应的代码。每个事务都和一个 TapeDeckHandler 的方法对应。每个事务的方法都使用@Transtration 标签来声明，这个标签定义了事件的 ID，该 ID 会触发事务的执行。事务开始时的状态使用 start，事务结束使用 next，事务正在运行使用 on。

----------START----------

```
public class TapeDeckHandler {
    @State public static final String EMPTY = "Empty";
    @State public static final String LOADED = "Loaded";
    @State public static final String PLAYING = "Playing";
    @State public static final String PAUSED = "Paused";

    @Transition(on = "load", in = EMPTY, next = LOADED)
    public void loadTape(String nameOfTape) {
        System.out.println("Tape '" + nameOfTape + "' loaded");
    }

    @Transitions({
        @Transition(on = "play", in = LOADED, next = PLAYING),
        @Transition(on = "play", in = PAUSED, next = PLAYING)
    })
    public void playTape() {
        System.out.println("Playing tape");
    }

    @Transition(on = "pause", in = PLAYING, next = PAUSED)
    public void pauseTape() {
        System.out.println("Tape paused");
    }

    @Transition(on = "stop", in = PLAYING, next = LOADED)
    public void stopTape() {
        System.out.println("Tape stopped");
    }

    @Transition(on = "eject", in = LOADED, next = EMPTY)
    public void ejectTape() {
        System.out.println("Tape ejected");
    }
```

```
}
----------END----------
```

请注意，TapeDeckHandler 类没有实现 TapeDeck ，呵呵，这是故意的。

现在让我们亲密接触一下这个代码。在 loadTape 方法上的@Transition 标签：

```
----------START----------
@Transition(on = "load", in = EMPTY, next = LOADED)
public void loadTape(String nameOfTape) {}
----------END----------
```

指定了这个状态后，当录音机处于空状态时，磁带装载事件启动后会触发 loadTape 方法，并且录音机状态将会变换到 Loaded 状态。 @Transition 标签中关于 pauseTape,stopTape,ejectTape 的方法就不需要在多介绍了。关于 playTape 的标签和其他的标签看起来不太一样。从上面的图中我们可以知道，当录音机的状态在 Loaded 或者 Paused 时，play 事件都会播放磁带。当多个事务同时条用同一个方法时，@Transition 标签需要按下面的方法使用：

```
----------START----------
@Transitions({
    @Transition(on = "play", in = LOADED, next = PLAYING),
    @Transition(on = "play", in = PAUSED, next = PLAYING)
})
public void playTape(){}
----------END----------
```

@Transition 标签清晰的列出了声明的方法被多个事务调用的情况。

```
##################################################
```

要点：更多关于@Transition 标签的参数

（1）如果你省略了 on 参数，系统会将该值默认为“*”，这样任何事件都可以触发该方法。

（2）如果你省略了 next 参数，系统会将默认值改为“_self_”，这个是和当前的状态相关的，
如果你要实现一个循环的事务，你所需要做的就是省略状态机中的 next 参数。

（3）weight 参数用于定义事务的查询顺序，一般的状态的事务是根据 weight 的值
按升序排列的，weight 默认的是 0.

```
##################################################
```

现在最后一步就是使用声明类创建一个状态机的对象，并且使用这个状态机的实例创建一个代理对象，该代理对象实现了 TapeDeck 接口:

```
----------START----------
public static void main(String[] args) {
    //  创建录音机事件的句柄
    TapeDeckHandler handler = new TapeDeckHandler();
    // 创建录音机的状态机
    StateMachine sm =
StateMachineFactory.getInstance(Transition.class).create(TapeDeckHandler.EMPTY,
handler);
    // 使用上面的状态机，通过一个代理创建一个 TapeDeck 的实例
    TapeDeck deck = new StateMachineProxyBuilder().create(TapeDeck.class, sm);
```

```
    // 加载磁带
    deck.load("The Knife - Silent Shout");
    // 播放
    deck.play();
    // 暂停
    deck.pause();
    // 播放
    deck.play();
    // 停止
    deck.stop();
    // 退出
    deck.eject();
}
```

----------END----------

这一行

----------START----------

```
TapeDeckHandler handler = new TapeDeckHandler();
StateMachine sm = StateMachineFactory.getInstance(Transition.class).create(TapeDeckHandler.EMPTY, handler);
```

----------END----------

使用 TapeDeckHandler 创建一个状态机的实例。StateMachineFactory.getInstance(...) 调用的方法中使用的 Transition.class 是通知工厂我们使用@Transition 标签创建一个状态机。我们指定了状态机开始时状态是空的。一个状态机是一个基本的指示图。状态对象和图中的节点对应，事务对象和箭头指向的方向对应。我们在 TapeDeckHandler 中使用的 每一个 @Transition 标签都和一个事务的实例对应。

##

要点： 那么， @Transition 和 Transition 有什么不同吗？

@Transition 是你用来标记当事务在状态之间变化时应该使用那个方法。在后台处理中，

Mina 的状态机会为 MethodTransition 中每一个事务标签创建一个事务的实例。MethodTransition

实现了 Transition 接口。作为一个 Mina 状态机的使用者，你不用直接使用 Transition 或者

MethodTransition 类型的对象。

##

录音机 TapeDeck 的实例是通过调用 StateMachineProxyBuilder 来创建的:

----------START----------

```
TapeDeck deck = new StateMachineProxyBuilder().create(TapeDeck.class, sm);
```

----------END----------

StateMachineProxyBuilder.create()使用的接口需要由代理的对象来实现，状态机的实例将接收由代理产生的事件所触发的方法。

当代码执行时，输出的结果如下：

----------START----------

```
Tape 'The Knife - Silent Shout' loaded
Playing tape
```

Tape paused
Playing tape
Tape stopped
Tape ejected
----------END----------
##
要点： 这和 Mina 有什么关系？
或许你已经注意到，在这个例子中没有对 Mina 进行任何配置。但是不要着急。
稍后我们将会看到如何为 Mina 的 IoHandler 接口创建一个状态机。
##

它是怎样工作的？

让我们走马观花的看看当代理调用一个方法的时候发生了什么。

查看一个 StateContext(状态的上下文)对象
状态上下文之所以重要是因为它保存了当前的状态。代理调用一个方法时，状态上下文会通知 StateContextLookup 实例去方法的参数中获取一个状态的上下文。一般情况下，StateContextLookup 的实现将会循环方法中的参数，并查找一个指定类型的对象，并且使用这个对象反转出一个上下文对象。如果没有声明一个状态上下文，StateContextLookup 将会创一个，并将其存放到对象中。

当代理 Mina 的 IoHandler 接口时，我们将使用 IoSessoinStateContextLookup 实例，该实例用来查询一个 IoSession 中的方法参数。它将会使用 IoSession 的属性值为每一个 Mina 的 session 来存放一个独立的状态上下文的实例。这中方式下，同样的状态机可以让所有的 Mina 的会话使用，而不会使每个会话彼此产生影响。

##
要点： 在上面的例子中，当我们使用 StateMachineProxyBuilder 创建一个代理时，我们一直没有我们一直没有配置 StateContextLookup 使用哪种实现。如果没有配置，系统会使用 SingletonStateContextLookup 。SingletonStateContextLookup 总是不理会方法中传递给它的参数，它一直返回一个相同的状态上下文。很明显，这中方式在多个客户端并发的情况下使用同一个同一个状态机是没有意义的。这种情况下的配置会在后面的关于 IoHandler 的代理配置时进行说明。
##
将方法请求反转成一个事件对象所有在代理对象上的方法请求都会有代理对象转换成事件对象。一个事件有一个 ID 或者 0 个或多个参数。事件的 ID 和方法的名字相当，事件的参数和方法的参数相当。调用方法 deck.load("The Knife - Silent Shout") 相当于事件{id = "load", arguments = ["The Knife - Silent Shout"]}.事件对象中包含一个状态上下文的引用，该状态上下文是当前查找到的。

触发状态机

一旦事件对象被创建，代理会调用 StateMachine.handle(Event). 方法。StateMachine.handle(Event)遍历事务对象中当前的状态，来查找能够接收当前事件的事务的实例。这个过程会在事务的实例找到后停止。这个查询的顺序是由事务的重量值来决定的

（重量值一般在@Transition 标签中指定）。

执行事务

最后一部就是在 Transition 中调用匹配事件对象的 Transition.execute(Event)方法。当事件已经执行，这个状态机将更新当前的状态，更新后的值是你在事务中定义的后面的状态。
##
要点： 事务是一个接口。每次你使用@Transition 标签时，MethodTransition 对象将会被创建。
##

MethodTransition(方法事务)

MethodTransition 非常重要，它还需要一些补充说明。如果事件 ID 和@Transition 标签中的 on 参数匹配，事件的参数和@Transition 中的参数匹配，那么 MethodTransition 和这个事件匹配。

所以如果事件看起来像{id = "foo", arguments = [a, b, c]}，那么下面的方法：

```
----------START----------
@Transition(on = "foo")
public void someMethod(One one, Two two, Three three) { ... }
----------END----------
```

只和这个事件匹配((a instanceof One && b instanceof Two && c instanceof Three) == true).。当匹配时，这个方法将会被与其匹配的事件使用绑定的参数调用。
##
要点： Integer, Double, Float, 等也和他们的基本类型 int, double, float, 等匹配。
##

因此，上面的状态是一个子集，需要和下面的方法匹配：

```
----------START----------
@Transition(on = "foo")
public void someMethod(Two two) { ... }
----------END----------
```

上面的方法和((a instanceof Two || b instanceof Two || c instanceof Two) == true)是等价的。在这种情况下，第一个被匹配的事件的参数将会和该方法绑定，在它被调用的时候。

一个方法如果没有参数，在其事件的 ID 匹配时，仍然会被调用：

```
----------START----------
@Transition(on = "foo")
public void someMethod() { ... }
----------END----------
```

这样做让事件的处理变得有点复杂，开始的两个方法的参数和事件的类及状态的上下文接口相匹配。这意味着：

```
----------START----------
@Transition(on = "foo")
public void someMethod(Event event, StateContext context, One one, Two two, Three three) { ... }
```

```
@Transition(on = "foo")
public void someMethod(Event event, One one, Two two, Three three) { ... }
@Transition(on = "foo")
public void someMethod(StateContext context, One one, Two two, Three three) { ... }
```

----------END----------

上面的方法和事件{id = "foo", arguments = [a, b, c]} if ((a instanceof One && b instanceof Two&& c instanceof Three) == true) 是匹配的。当前的事件对象和事件的方法绑定，当前的状态上下文和该方法被调用时的上下文绑定。

在此之前一个事件的参数的集合将会被使用。当然，一个指定的状态上下文的实现将会被指定，以用来替代通用的上下文接口。

----------START----------

```
@Transition(on = "foo")
public void someMethod(MyStateContext context, Two two) { ... }
```

----------END----------

##

要点：方法中参数的顺序很重要。若方法需要访问当前的事件，它必须被配置为第一个方法参数。当事件为第一个参数的时候，状态上下问不能配置为第二个参数，它也不能配置为第一个方法的参数。事件的参数也要按正确的顺序进行匹配。方法的事务不会在查找匹配事件方法的时候重新排序。

##

到现在如果你已经掌握了上面的内容，恭喜你！我知道上面的内容会有点难以消化。希望下面的例子 能让你对上面的内容有更清晰的了解。注意这个事件 Event {id = "messageReceived", arguments = [ArrayList a = [...], Integer b = 1024]}。下面的方法将和这个事件是等价的：

----------START----------

```
// All method arguments matches all event arguments directly
@Transition(on = "messageReceived")
public void messageReceived(ArrayList l, Integer i) { ... }

// Matches since ((a instanceof List && b instanceof Number) == true)
@Transition(on = "messageReceived")
public void messageReceived(List l, Number n) { ... }

// Matches since ((b instanceof Number) == true)
@Transition(on = "messageReceived")
public void messageReceived(Number n) { ... }

// Methods with no arguments always matches
@Transition(on = "messageReceived")
public void messageReceived() { ... }

// Methods only interested in the current Event or StateContext always matches
@Transition(on = "messageReceived")
```

```
public void messageReceived(StateContext context) { ... }

// Matches since ((a instanceof Collection) == true)
@Transition(on = "messageReceived")
public void messageReceived(Event event, Collection c) { ... }
```
----------END----------
但是下面的方法不会和这个事件相匹配：
----------START----------
```
// Incorrect ordering
@Transition(on = "messageReceived")
public void messageReceived(Integer i, List l) { ... }

// ((a instanceof LinkedList) == false)
@Transition(on = "messageReceived")
public void messageReceived(LinkedList l, Number n) { ... }

// Event must be first argument
@Transition(on = "messageReceived")
public void messageReceived(ArrayList l, Event event) { ... }

// StateContext must be second argument if Event is used
@Transition(on = "messageReceived")
public void messageReceived(Event event, ArrayList l, StateContext context) { ... }

// Event must come before StateContext
@Transition(on = "messageReceived")
public void messageReceived(StateContext context, Event event) { ... }
```
----------END----------

状态继承

状态的实例将会有一个父类的状态。如果StateMachine.handle(Event)的方法不能找到一个事务和当前的事件在当前的状态中匹配，它将会寻找父类中的装。如果仍然没有找到，那么事务将会自动寻找父类的父类，知道找到为止。

这个特性很有用，当你想为所有的状态添加一些通用的代码时，不需要为每一个状态的方法来声明事务。这里你可以创建一个类的继承体系，使用下面的方法即可：
----------START----------
```
@State      public static final String A = "A";
@State(A) public static final String B = "A->B";
@State(A) public static final String C = "A->C";
@State(B) public static final String D = "A->B->D";
@State(C) public static final String E = "A->C->E";
```
----------END----------
使用状态继承来处理错误信息

让我们回到录音机的例子。如果录音机里没有磁带，当你调用 deck.play()方法时将会怎样？让我们试试：
示例代码：

```
----------------------START----------------------
public static void main(String[] args) {
    ...
    deck.load("The Knife - Silent Shout");
    deck.play();
    deck.pause();
    deck.play();
    deck.stop();
    deck.eject();
    deck.play();
}
----------------------END----------------------
```

运行结果：

```
----------------------START----------------------
...
Tape stopped
Tape ejected
Exception in thread "main" o.a.m.sm.event.UnhandledEventException:
Unhandled event: org.apache.mina.statemachine.event.Event@15eb0a9[id=play,...]
    at org.apache.mina.statemachine.StateMachine.handle(StateMachine.java:285)
    at
org.apache.mina.statemachine.StateMachine.processEvents(StateMachine.java:142)
    ...
----------------------END----------------------
```

哦，我们得到了一个无法处理的异常 UnhandledEvent Exception，这是因为在录音机的空状态时，没有事务来处理播放的状态。我们将添加一个指定的事务来处理所有不能匹配的事件。

```
----------------------START----------------------
@Transitions({
    @Transition(on = "*", in = EMPTY, weight = 100),
    @Transition(on = "*", in = LOADED, weight = 100),
    @Transition(on = "*", in = PLAYING, weight = 100),
    @Transition(on = "*", in = PAUSED, weight = 100)
})
public void error(Event event) {
    System.out.println("Cannot '" + event.getId() + "' at this time");
}
----------------------END----------------------
```

现在当你运行上面的 main()方法时，你将不会再得到一个异常，输出如下：

```
----------------------START----------------------
```

...
Tape stopped
Tape ejected
Cannot 'play' at this time.
----------END----------
现在这些看起来运行的都很好，是吗？但是如果们有 30 个状态而不是 4 个，那该怎么办？那么我们需要在上面的错误方法处理中配置 30 个事务的声明。这样不好。让我们用状态继承来解决:
----------START----------

```
public static class TapeDeckHandler {
    @State public static final String ROOT = "Root";
    @State(ROOT) public static final String EMPTY = "Empty";
    @State(ROOT) public static final String LOADED = "Loaded";
    @State(ROOT) public static final String PLAYING = "Playing";
    @State(ROOT) public static final String PAUSED = "Paused";
      ...
    @Transition(on = "*", in = ROOT)
    public void error(Event event) {
        System.out.println("Cannot '" + event.getId() + "' at this time");
    }
}
```

----------END----------
这个运行的结果和上面的是一样的，但是它比要每个方法都配置声明要简单的多。

Mina 的状态机和 IoHandler 配合使用

现在我们将上面的录音机程序改造成一个 TCP 服务器，并扩展一些方法。服务器将接收一些命令类似于： load <tape>, play, stop 等等。服务器响应的信息将会是+ <message> 或者是- <message>。协议是基于 Mina 自身提供的一个文本协议，所有的命令和响应编码都是基于 UTF-8。这里有一个简单的会话示例：
----------START----------

```
telnet localhost 12345
S: + Greetings from your tape deck!
C: list
S: + (1: "The Knife - Silent Shout", 2: "Kings of convenience - Riot on an empty street")
C: load 1
S: + "The Knife - Silent Shout" loaded
C: play
S: + Playing "The Knife - Silent Shout"
C: pause
S: + "The Knife - Silent Shout" paused
C: play
S: + Playing "The Knife - Silent Shout"
C: info
S: + Tape deck is playing. Current tape: "The Knife - Silent Shout"
```

C: eject
S: - Cannot eject while playing
C: stop
S: + "The Knife - Silent Shout" stopped
C: eject
S: + "The Knife - Silent Shout" ejected
C: quit
S: + Bye! Please come back!
----------END----------
该程序完整的代码在 org.apache.mina.example.tapedeck 包中，这个可以通过检出 Mina 源码的 SVN 库中的 mina-example 来得到。代码使用 Mina 的 ProtocolCodecFilter 来编解码传输的二进数据对象。这里只是为每个状态对服务器的请求实现了一个简单的编解码器。在此不在对 Mina 中编解码的实现做过多的讲解。

现在我们看一下这个服务器是如何工作的。这里面一个重要的类就是实现了录音机程序的 TapeDeckServer 类。这里我们要做的第一件事情就是去定义这些状态：
----------START----------
```
@State public static final String ROOT = "Root";
@State(ROOT) public static final String EMPTY = "Empty";
@State(ROOT) public static final String LOADED = "Loaded";
@State(ROOT) public static final String PLAYING = "Playing";
@State(ROOT) public static final String PAUSED = "Paused";
```
----------END----------
在这里没有什么新增的内容。然而，但是处理这些事件的方法看起来将会不一样。让我们看看 playTape 的方法。
----------START----------
```
@IoHandlerTransitions({
    @IoHandlerTransition(on = MESSAGE_RECEIVED, in = LOADED, next = PLAYING),
    @IoHandlerTransition(on = MESSAGE_RECEIVED, in = PAUSED, next = PLAYING)
})
public void playTape(TapeDeckContext context, IoSession session, PlayCommand cmd) {
    session.write("+ Playing \"" + context.tapeName + "\"");
}
```
----------END----------
这里没有使用通用的 @Transition 和 @Transitions 的事务声明，而是使用了 Mina 指定的 @IoHandlerTransition 和 @IoHandlerTransitions 声明。当为 Mina 的 IoHandler 创建一个状态机时，它会选择让你使用 Java enum （枚举）类型来替代我们上面使用的字符串类型。这个在 Mina 的 IoFilter 中也是一样的。

我们现在使用 MESSAGE_RECEIVED 来替代 "play" 来作为事件的名字 (on 是 @IoHandlerTransition 的一个属性)。这个
常量是在 org.apache.mina.statemachine.event.IoHandlerEvents 中定义的，它的值是 "messageReceived"，这个和 Mina 的 IoHandler 中的 messageReceived()方法是一致的。谢谢 Java 5 中的静态导入，我们在使用该变量的时候就不用再通过类的名字来调用该常量，

我们只需要按下面的方法导入该类：

----------START----------

```
import static org.apache.mina.statemachine.event.IoHandlerEvents.*;
```

----------END----------

这样状态内容就被导入了。

另外一个要改变的内容是我们自定了一个 StateContext 状态上下文的实现 -- TapeDeckContext。这个类主要是用于返回当前录音机的状态的名字。

----------START----------

```
static class TapeDeckContext extends AbstractStateContext {
    public String tapeName;
}
```

----------END----------

##

要点：为什么不把状态的名字保存到 IoSession 中？

我们可以将录音机状态的名字保存到 IoSession 中，但是使用一个自定义的 StateContext 来保存这个状态将会使这个类型更加安全。

##

最后需要注意的事情是 playTape()方法使用了 PlayCommand 命令来作为它的最后的一个参数。最后一个参数和 IoHandler's messageReceived(IoSession session, Object message)方法匹配。这意味着只有在客户端发送的信息被编码成 playCommand 命令时，该方法才会被调用。

在录音机开始进行播放前，它要做的事情就是要装载磁带。当装载的命令从客户端发送过来时，服务器提供的磁带的数字代号将会从磁带列表中将可用的磁带的名字取出：

----------START----------

```
@IoHandlerTransition(on = MESSAGE_RECEIVED, in = EMPTY, next = LOADED)
public void loadTape(TapeDeckContext context, IoSession session, LoadCommand cmd) {
    if (cmd.getTapeNumber() < 1 || cmd.getTapeNumber() > tapes.length) {
        session.write("- Unknown tape number: " + cmd.getTapeNumber());
        StateControl.breakAndGotoNext(EMPTY);
    } else {
        context.tapeName = tapes[cmd.getTapeNumber() - 1];
        session.write("+ \"" + context.tapeName + "\" loaded");
    }
}
```

----------END----------

这段代码使用了 StateControl 状态控制器来重写了下一个状态。如果用户指定了一个非法的数字，我们将不会将加载状态删除，而是使用一个空状态来代替。代码如下所示：

----------START----------

```
StateControl.breakAndGotoNext(EMPTY);
```

----------END----------

状态控制器将会在后面的章节中详细的讲述。

connect()方法将会在 Mina 开启一个会话并调用 sessionOpened()方法时触发。

----------START----------

```
@IoHandlerTransition(on = SESSION_OPENED, in = EMPTY)
public void connect(IoSession session) {
    session.write("+ Greetings from your tape deck!");
}
----------------------END----------------------
```

它所做的工作就是向客户端发送欢迎的信息。状态机将会保持空的状态。
pauseTape(), stopTape() 和 ejectTape() 方法和 playTape()很相似。这里不再进行过多的讲述。listTapes(), info() 和 quit() 方法也很容易理，也不再进行过多的讲解。请注意后面的三个方法是在根状态下使用的。这意 味着 listTapes(), info() 和 quit() 可以在任何状态中使用。

现在让我们看一下错误处理。error()将会在客户端发送一个非法的操作时触发：

```
----------------------START----------------------
@IoHandlerTransition(on = MESSAGE_RECEIVED, in = ROOT, weight = 10)
public void error(Event event, StateContext context, IoSession session, Command cmd) {
    session.write("- Cannot " + cmd.getName() + " while "
            + context.getCurrentState().getId().toLowerCase());
}
----------------------END----------------------
```

error()已经被指定了一个高于 listTapes(), info() 和 quit() 的重量值来阻止客户端调用上面的方法。注意 error()方法是怎样使用状态上下文来保存当前状态的 ID 的。字符串常量值由@State annotation (Empty, Loaded etc) 声明。这个将会由 Mina 的状态机当成状态的 ID 来使用。

commandSyntaxError()方法将会在 ProtocolDecoder 抛出 CommandSyntaxException 异常时被调用。它将会简单的输出
客户端发送的信息不能解码为一个状态命令。

exceptionCaught() 方法将会在任何异常发生时调用，除了CommandSyntaxException 异常(这个异常有一个较高的重量值)。
它将会立刻关闭会话。

最后一个 @IoHandlerTransition 的方法是 unhandledEvent() ，它将会在 @IoHandlerTransition 中的方法没有事件匹配时调用。
我们需要这个方法是因为我们没有@IoHandlerTransition 的方法来处理所有可能的事件 (例如：我们没有处理 messageSent(Event)
方法)。没有这个方法，Mina 的状态机将会在执行一个事件的时候抛出一个异常。

最后一点我们要看的是那个类创建了 IoHandler 的代理，main()方法也在其中：

```
----------------------START----------------------
private static IoHandler createIoHandler() {
    StateMachine sm =
StateMachineFactory.getInstance(IoHandlerTransition.class).create(EMPTY, new
```

```
TapeDeckServer());

        return new StateMachineProxyBuilder().setStateContextLookup(
                new IoSessionStateContextLookup(new StateContextFactory() {
                    public StateContext create() {
                        return new TapeDeckContext();
                    }
                })).create(IoHandler.class, sm);
}

// This code will work with MINA 1.0/1.1:
public static void main(String[] args) throws Exception {
        SocketAcceptor acceptor = new SocketAcceptor();
        SocketAcceptorConfig config = new SocketAcceptorConfig();
        config.setReuseAddress(true);
        ProtocolCodecFilter pcf = new ProtocolCodecFilter(
                new TextLineEncoder(), new CommandDecoder());
        config.getFilterChain().addLast("codec", pcf);
        acceptor.bind(new InetSocketAddress(12345), createIoHandler(), config);
}

// This code will work with MINA trunk:
public static void main(String[] args) throws Exception {
        SocketAcceptor acceptor = new NioSocketAcceptor();
        acceptor.setReuseAddress(true);
        ProtocolCodecFilter pcf = new ProtocolCodecFilter(
                new TextLineEncoder(), new CommandDecoder());
        acceptor.getFilterChain().addLast("codec", pcf);
        acceptor.setHandler(createIoHandler());
        acceptor.setLocalAddress(new InetSocketAddress(PORT));
        acceptor.bind();
}
----------------------END----------------------
```

createIoHandler() 方法创建了一个状态机，这个和我们之前所做的相似除了我们指定一个 IoHandlerTransition.clas 类来代替 Transition.class 在 StateMachineFactory.getInstance(...) 方法中。这是我们在使用 @IoHandlerTransition 声明的时候必须要做的。当然这时我们使用了一个 IoSessionStateContextLookup 和一个自定义的 StateContextFactory 类，这个在我们创建一个 IoHandler 代理时被使用到了。如果我们没有使用 IoSessionStateContextLookup ，那么所有的客户端将会使用同一个状态机，这是我们不希望看到的。

main()方法创建了 SocketAcceptor 实例，并且绑定了一个 ProtocolCodecFilter ，它用于编解码命令对象。最后它绑定了 12345 端口和 IoHandler 的实例。这个 IoHandler 实例是由 createIoHandler()方法创建的。

深入理解 Apache Mina ---- Mina 的几个类

最近一直在看 Mina 的源码，用了 Mina 这么长时间，说实话，现在才开始对 Mina 有了一些深刻的理解，关于 Mina 的基本知识的介绍，这里就不多说了，网上已经有很多不错的文章都对 Mina 做了较深刻的剖析，现在就是想从 Mina 的最根本的地方来对 Mina 做一些深层次上的探讨。

还是先从 Mina 的入口程序来说，每当要启动一个 Mina 的程序（包括服务器和客户端）时候，这里只是对服务器重点做一些讲解，至于说 Mina 的客户端的应用，这里只是简单的涉及一点，不会对其做很深入的探讨。但是 Mina 的服务器和客户端在很大的程度上都是一样，所以这里就“挂一漏万”的简单讲解一下。

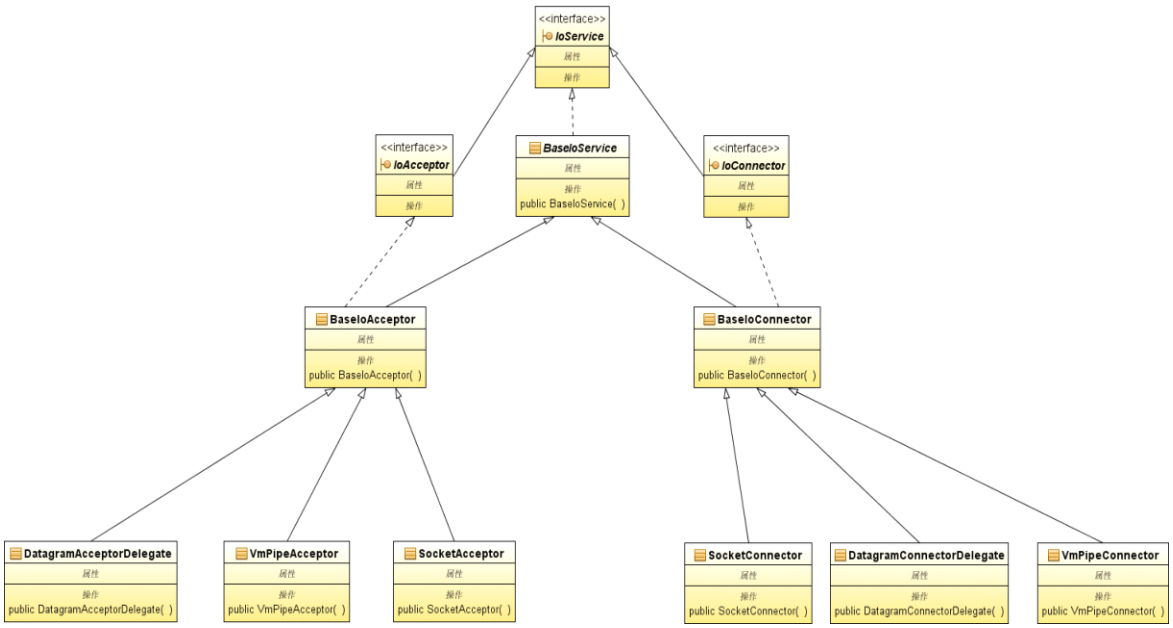
在此之前我一直想找一种“串糖葫芦”的方式来讲解一下 Mina，可是一直没有时间来看 Mina 的源码，真的是无从下手，虽然网上的很多关于 Mina 的一些文章，讲解的非常透彻了，但是可能对于初学者来说，显得有些深奥，在这里特别的提一下洞庭散人对 Mina 源码的透彻的分析，如果你对 Mina 已经有了一定的了解或者是正在学习 Mina 的源码，建议你去看看他的博客，里面有很多东西讲的是相当到位的。在这里就不在多举例子了。写这篇文档主要是想对刚接触 Mina 的人讲解一些 Mina 的基本知识，由浅入深，一步一步的学习 Mina 思想的精髓，我接触 Mina 的时间也比较长了，几乎天天在和它打交道，每当你发现一个新奇的用法的时候，你真的会被 Mina 所折服，我这里不是对 Mina 的吹捧，记得我曾经和同事开玩笑说，“等真正的懂得了 Mina，你就知道什么叫 Java 了”，所以，我现在想急切的把现在所知道和了解的所有关于 Mina 的一些东西都想在这篇文章里面写出来，如果有写的不到位的地方还请各位同学多多指正，下面就开始对 Mina 做一个完整的介绍。

一、先说说 Mina 的几个类和接口

（1）IoService
（2）BaseIoService
（3）BaseIoAcceptor
（4）IoAcceptor
（5）IoConnector

这几个类和接口是整个服务器或客户端程序（IoConnector）的入口程序，其中就 Mina 的整体上来说，IoService 是所有 IO 通信的入口程序，下面的几个接口和类都是继承或者实现了 IoService 接口。

下面先给出 Mina（入口程序）的整体架构图：



Mina 的整体架构图

在这里先提出几个个问题：

（1）为什么有了一个 IoService 还要再有一个 BaseIoService？

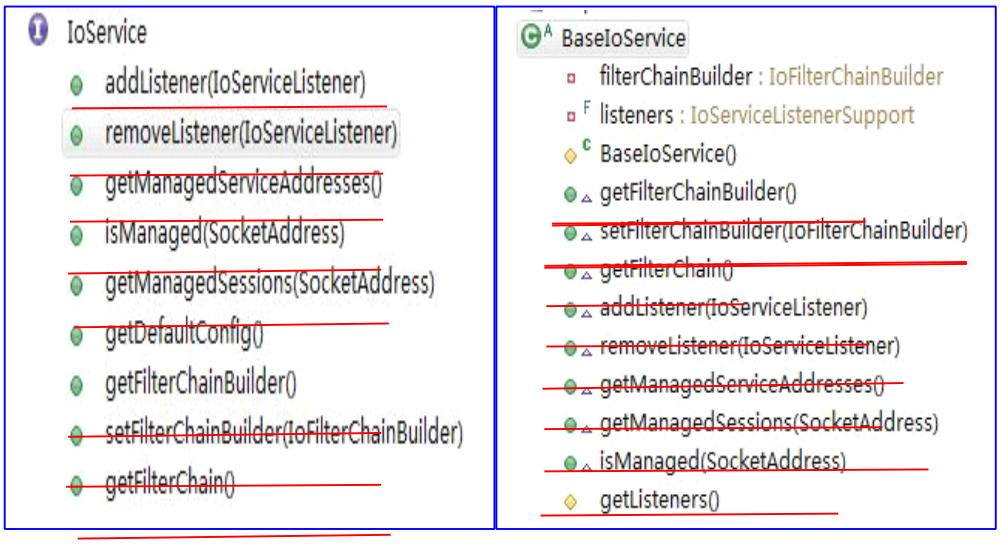
（2）BaseIoService 和 IoAcceptor(IoConnector)有什么区别？

（3）BaseIoAcceptor(BaseIoConnector)为什么不去直接实现 IoService，而是又添加了 IoAcceptor(IoConnector)?

带着这几个问题我们来解读一下 Mina 的源码：

首先，解答第一个问题，为什么有了一个 IoService 还要再有一个 BaseIoService？IoService 和 BaseIoService 最明显的区别就是 IoService 是一个接口，而 BaseIoService 是一个抽象类。BaseIoService 实现了 IoService 中的部分方法。

这里先把 IoService 接口中要实现的方法和 BaseIoService 中实现的方法列举如下：



通过对 IoService 和 BaseIoService 的比较可以发现，除了 getDefaultConfig()这个方法没有在 BaseIoService 中实现之外，其他的方法都已经在 BaseIoService 实现了。这里就有一个问题，为什么 BaseIoService 只是实现了 IoService 的部分方法，而没有全部实现 IoService 的方法呢？通常都知道，接口中的方法是必须要由实现类来实现的，这点是毋庸置疑的。你可以写一个空方法，里面没有任何的逻辑处理，但是你的实现类中却不能没有该方法。但是在 Mina 中作为实现类的 BaseIoService 却没有 IoService 指定的方法 getDefaultConfig()，难道 Mina 真的有独到之处？不是！仔细看看 BaseIoService 你就会知道，BaseIoService 是一个抽象类，抽象类就是用来被继承的，它提供了一些其子类公用的一些方法，当抽象类实现一个接口时，抽象类可以有选择性的实现其所有子类都需要的实现的一些方法，对于接口中指定法方法，抽象类可以选择全部实现或者部分实现。在Mina 中如果没有 BaseIoService 这个抽象类，而是由 BaseIoAcceptor 和 BaseIoConnector 直接去实现 BaseIoService 接口，那么必然会导致这个两个实现类中都要重写相应的方法，这样就脱离了面向对象设计的本质，没有达到复用的目的。在 BaseIoAcceptor/BaseIoConnector 和 BaseIoService 之间添加一个 BaseIoService 就是为了达到代码复用的目的。在这个问题上主要是要记住两点：

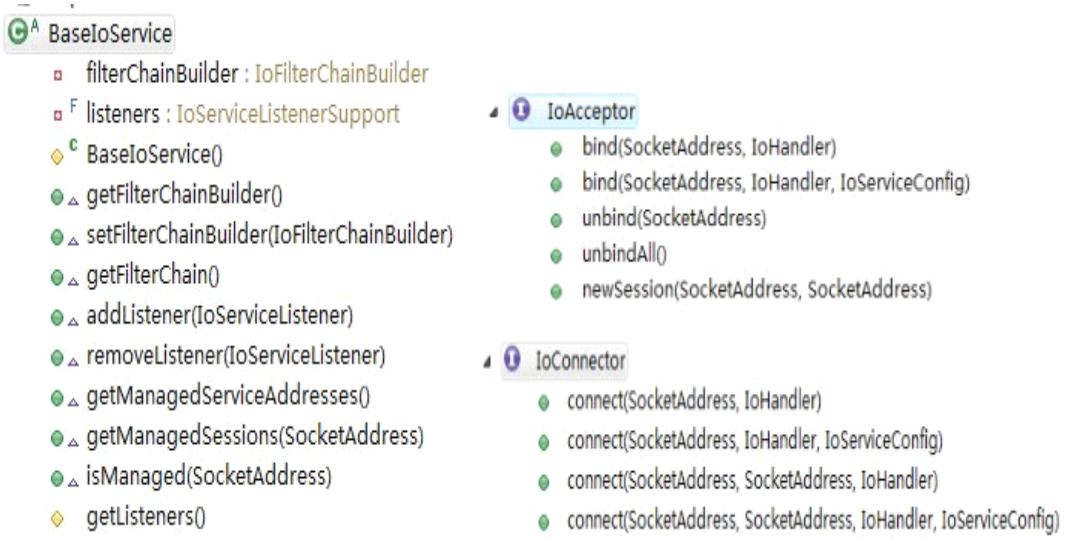
1）抽象类在实现接口的时候可以部分或者全部实现接口中的方法。但是当抽象类只是实现了接口中的部分方法的时候，抽象类的子类必须要实现抽象类中未实现的接口的方法。在此处，IoService 的 getDefaultConfig() 方法在 BaseIoService(BaseIoAcceptor 是 BaseIoService 的子类，但它也是一个抽象类，所以它也没有实现 getDefaultConfig()) ，getDefaultConfig() 是由 BaseIoAcceptor 的子类们来实现的(如 SocketAcceptor，这是一个具体实现类)。所以接口的所有方法必须被具体的实现类实现和抽象类在实现接口的时候可以部分或者全部实现接口中的方法是不矛盾的。

2）注意代码的重用。在面向对象的编程语言中都提供了抽象类和接口，抽象类和接口最大的区别就是抽象类提供了方法的具体实现，供其子类来调用；而接口只是提供了对方法的声明，其方法的实现要由其具体实现类来做。在 Java 中一个子类只能有一个父类，但是却能实现多个接口。个人认为接口和抽象类各有特色，接口的使用比较灵活，不同的接口可以让其子类扮演不同的角色，侧重于类的复用，在很大程度上解决了代码复用的问题；抽象类更侧重的是方法的复用，某种意义上讲，抽象类的使用对于程序来说使用起来更加轻松，但是是使用抽象类还是接口要根据具体的情况而定。对于接口和抽象类的具体的用法请参考闫宏的《Java 与模式》中相关部分的讲解。

之所以在这里罗列这么些问题，目的不仅仅是为了讲解 Mina 的原理，而是想从一个高的角度来看待的这个经典的开源项目，通过对 Mina 的学习和理解，能够真正的懂得什么是一个项目，什么是面向对象编程，更本质的东西是怎么灵活运用 Java 来达到上面的两个目的。这个才是最重要的，哪怕是你在看完本文后对 Mina 的理解还是有点模糊，但是你至少要知道在编写一个程序的时候怎样从面向对象的角度上去思考一个问题，而不是在用着面向对象的语言写着结构化的程序。这些东西都是自己做开发这么长时间的一些心得，在这里总结出来，目的主要是用于交流和学习，不是在卖弄，只是想让更多的初学者少走一些弯路，懂得学习的方法。

还是回到对 Mina 的刚提出的那几个问题上来，现在，第一个问题已经解决了，为什么有了一个 IoService 还要再有一个 BaseIoService？答案就是为了代码的复用。

其次，下面开始讨论第二个问题，BaseIoService 和 IoAcceptor(IoConnector)有什么区别？在讨论这个问题之前，还是先给出这两个类(接口)提供的方法，如下图：



在讨论第一个问题的时候我们已经看过了 BaseIoService 的方法了，但是没有对这些方法的功能做些梳理，现在就对这些方法做些简单的介绍：

getFilterChainBuilder() 和 setFilterChainBuilder(): 这两个方法主要是对一个服务的 IoFilter 的操作，关于 IoFilter 的详细介绍会在后面给出，现在你可以将其理解为是一个处理业务逻辑的模块，例如：黑名单的处理、数据的转换、日志信息的处理等等都可以在这个IoFilter 中实现，它的工作原理和Servlet 中的过滤器很相似。

addListener() 和 removeListener(): 这两个方法通过名字看就可以理解了，就是给当前的服务添加和删除一个监听器，这个监听器主要是用于对当前连接到服务的 IoSession 进行管理，这个也会在后面做详细的讲解。

getManagerServiceAddress() 和 getManagerSessions(): 这两个方法的功能比较相似，一个是获取当前服务所管理的远程地址，一个是获取当前服务所管理的会话 IoSession ，IoSession 对 SocketAddress 做了一个完整的封装，你也可以先将这两个方法的功能理解为是一回事，具体的区别会在后面给出。

isManaged(): 检测某个 SocketAddress 是否处于被管理的状态。

getListeners(): 获取当前服务的监听器。

看了上面对 BaseIoService 功能的介绍，现在我们可以理解 BaseIoService 提供的方法主要是用于对当前服务的管理。那么要管理一个服务，前提条件是这个服务必须存在，存在的前提是什么，就是要启动一个服务，或者是连接到一个远程主机上，这两个任务分别是 IoAcceptor 和 IoConnector 来完成的，此处要注意的是这两个对象都是接口，没有具体的实现，具体的实现会由下面介绍的它们相关的子类(SocketAcceptor 等)来实现。这样 IoAcceptor/IoConnector 的功能我们就可以总结出来了，就是启动和停止一个服务。

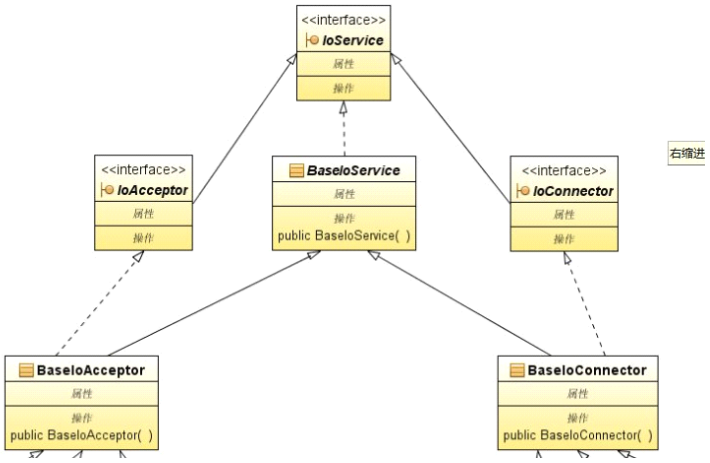
对于一个完整的服务来说，既要有启动这样的前提条件，还要有对服务的管理和对服务响应

的逻辑处理，这两个缺一不可，回到第二个问题，BaseIoService 和 IoAcceptor(IoConnector)有什么区别？区别就在于它们实现的功能不一样，但都是为了一个完整的服务来打基础的，两者缺一都不能称为一个完整的服务。这三个都是 IoService 子类(子接口)，IoService 只是提供了一些服务应该具有多基本的方法，BaseIoService 提供了 IoService 部分方法的具体实现，而 IoAcceptor(IoConnector)是对特定服务要具备的操作的做了一些扩展，这样一个服务完整的模型正在逐渐向我们清晰的展现出来。

再次，讨论一下第三个问题。BaseIoAcceptor(BaseIoConnector)为什么不去直接实现 IoService，而是又添加了 IoAcceptor(IoConnector)?这个问题其实在上面已经有所涉及，为了达到对象复用的目的，所以 Mina 的设计者给出了一个 BaseIoService，IoAcceptor(IoConnector)是实现一个特定服务必须要提供的一些方法。更具体一点，IoAcceptor(IoConnector)是为了一个特定的服务(服务器/客户端)而设计的，而 IoService 只是提供了一个服务应该具备的一些基本的方法。所以在 Mina 中给出了一个针对具体服务的一个接口 IoAcceptor(IoConnector)，这样 BaseIoAcceptor(BaseIoConnector)就提供了一个服务所必备的一些条件。因为它即实现了 IoAcceptor(IoConnector)接口又继承了抽象类 BaseIoService，这样就实现了 IoService 中的所有方法，并且也添加了特定服务应该具有的方法(即 IoAcceptor(IoConnector)中的方法)。以上就是第三个问题的答案。

二、Mina 中提供的几个特定的服务

从上面的讨论中我们已经知道了 Mina 上层的类和接口的一些功能。即图中所示的已经在上面解释清楚了。



Mina 的上层结构图(抽象层)

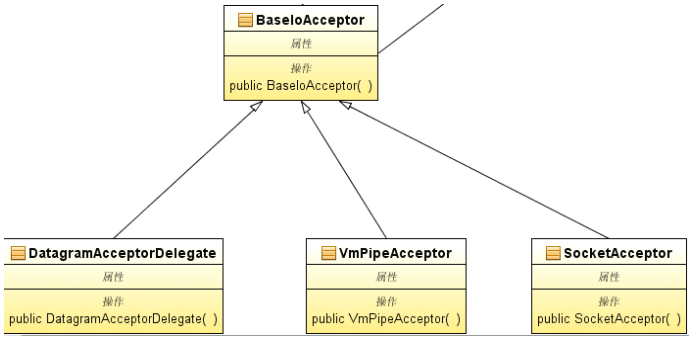
在此我们可以把 Mina 的上层结构简单的定义为 Mina 的“抽象层”，既然有了抽象层，肯定就会有其具体实现，抽象中最重要的两个类是 BaseIoAcceptor 和 BaseIoConnector，它们分别是用于服务器和客户端的一个入口程序。

首先，说一下 BaseIoAcceptor 中的三个具体实现类：

(1) DatagramAcceptorDelegate：数据报 UDP 通信的服务器入口程序。该类使用 UDP 协议进行通信，UDP 协议主要是用在视频、远程服务的监听(如心跳程序)中等数据传输要求不是很高的地方。

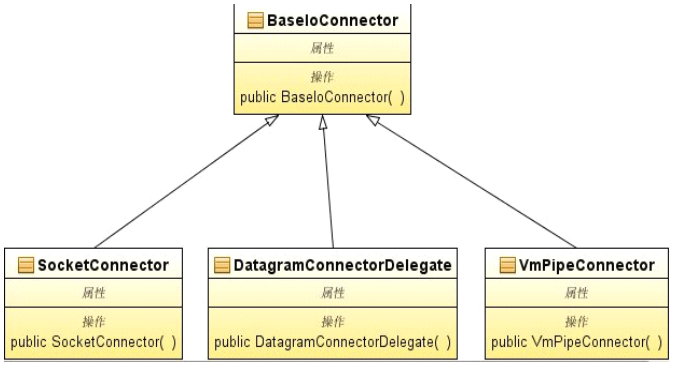
(2) VmPipeAcceptor：虚拟通道(VM)通信的服务器入口程序。虚拟管道协议主要用于无线通信方面。

(3) SocketAcceptor：TCP/IP 通信的服务器入口程序。这个是比较常用的协议，该协议主要数据传输要求较高的地方，比如实时系统、游戏服务器等。



BaseIoAcceptor 及其子类

与 BaseIoAcceptor 相对应的就是 BaseIoConnector，该类主要用于客户端程序。其具体的子类就不再赘述，这里只给出 BaseIoConnector 及其子类的结构图。



BaseIoConnector 及其子类

关于 SocketAcceptor、IoFilter、IoProcessor、IoHandler 等会有专门的文章来讨论。这里就不在对这些组件类做详细的说明了。

深入理解 Apache Mina ---- 与 IoFilter 相关的几个类

从名字上看知道 IoFilter 应该是一个过滤器，不错，它确实是一个过滤器，它和 Servlet 中的过滤器类似，主要用于拦截和过滤 I/O 操作中的各种信息。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（这个在本文中关于 LoggingFilter 的讲述中会提到）
（2）测量系统性能
（3）信息验证
（4）过载控制
（5）信息的转换 (例如：编码和解码，这个会在关于 ProtocolCodecFilter 的讲述中会提到)
（6）和其他更多的信息

还是上一篇文档一样，先提出几个问题，然后沿着这几个问题的思路一个一个的对 IoFilter 进行讲解。

（1）什么时候需要用到 IoFilter，如果在自己的应用中不添加过滤器可以吗？
（2）如果在 IoService 中添加多个过滤器可以吗？若可以，如何进行添加，这多个过滤器是如何工作的？
（3）Mina 中提供了协议编、解码器，IoFilter 也可以实现 IO 数据的编解码功能，在实际的使用中如何选择？

在开始对上面的问题进行讨论前，为了对 IoFilter 提供的方法有一个具体的了解，先对 Mina 自身提供的一个最简单的过滤器进行一些讲解----LoggingFilter（源码在附件中，配有中文翻译）。

首先还是看一下 LoggingFilter 中提供的几个方法。列举如下（方法中的参数就不再给出，完整方法的实现请参考附件中 LoggingFilter 的源码）：

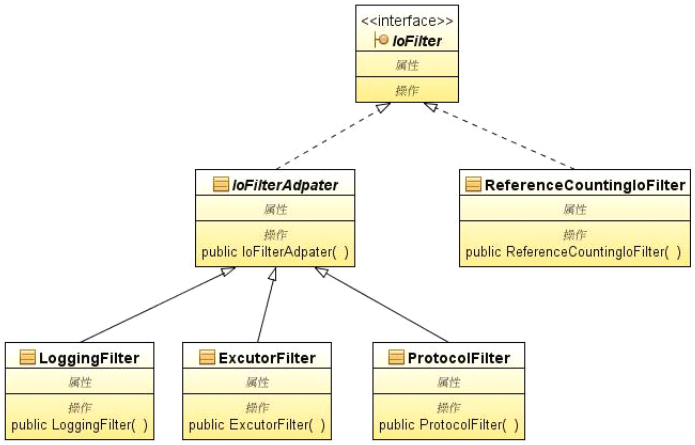
（1）sessionCreated()
（2）sessionOpened()
（3）sessionClosed()
（4）sessionIdle()
（5）exceptionCaught()
（6）messageReceived()
（7）messageSent()
（8）filterWrite()
（9）filterClose()

这几个方法都由相应会话（或者说是连接的状态，读、写、空闲、连接的开闭等）的状态的改变来触发的。当一个会话开启时，LoggingFilter 捕获到会话开启的事件，会触发 sessionCreated()方法，记录该会话开启的日志信息。同样当一个会话发送数据时，Logging 捕获到会话发送消息的事件会记录消息发送的日志信息。这里只是给出 messageReceived() 的完成方法的实现，其他方法的完整实现请参考附件中 LoggingFilter 的源码。

```
/**
 * 记录会话接收信息时的信息，然后将该信息传递到过滤器链中的下一个过滤器
 * */
public void messageReceived(NextFilter nextFilter, IoSession session,
        Object message) {
    if (SessionLog.isInfoEnabled(session)) {
        SessionLog.info(session, "RECEIVED: " + message);
    }
    nextFilter.messageReceived(session, message);
}
```

LoggingFilter 继承与 IoFilterAdpater，IoFilterAdpater 是 IoFilter 的一个实现类，该类只是提供了 IoFilter 方法的简单实现----将传递到各方法中的消息转发到下一个过滤器中。你可以根据自己的需求继承 IoFilterAdpater，并重写相关的方法。LoggingFilter 就是重写了上面提到的几个方法，用于记录当前的会话各种操作的日志信息。

通过上面的例子，我们可以大体的了解了 IoFilter 的基本功能：根据当前会话状态，来捕获和处理当前会话中所传递的消息。IoFilter 的 UML 图如下：



从上面的类图我们可以清晰的看到 IoFilter 是一个接口，它有两个具体的实现类：

IoFilterAdpater ：该类提供了 IoFilter 所有方法的方法体，但是没有任何逻辑处理，你可以根据你具体的需求继承该类，并重写相关的方法。IoFilterAdpater 是在过滤器中使用的较多的一个类。

ReferenceCountingIoFilter：该类封装 IoFilter 的实例，它使用监视使用该 IoFilter 的对象的数量，当没有任何对象使用该 IoFilter 时，该类会销毁该 IoFilter 。

IoFilterAdpater 有三个子类，它们的作用分别如下：

LoggingFilter：日志工具，该类处理记录 IoFilter 每个状态触发时的日志信息外不对数据做任何处理。它实现了 IoFilter 接口的所有方法。你可以通过阅读该类的源码学习如何实现你自己的 IoFilter 。

ExcuterFilter：这个 Mina 自身提供的一个线程池，在 Mina 中你可以使用这个类配置你自己的线程池，由于创建和销毁一个线程，需要耗费很多资源，特别是在高性能的程序中这点尤其重要，因此在你的程序中配置一个线程池是很重要的。它有助于你提高你的应用程序的性能。关于配置 Mina 的线程池在后续的文档中会给出详细的配置方法。

ProtocolFilter：该类是 Mina 提供的一个协议编解码器，在socket 通信中最重要的就是协议的编码和解码工作，Mina 提供了几个默认的编解码器的实现，在下面的例子中使用了 ObjectSerializationCodecFactory，这是 Mina 提供的一个 Java 对象的序列化和反序列化方法。使用这个

编解码器，你可以在你的 Java 客户端和服务器之间传递任何类型的 Java 对象。但是对于不同的平台之间的数据传递需要自己定义编解码器，关于这点的介绍会在后续的文档中给出。

为了更加清楚的理解这个过滤器的作用我们先来看一个简单的例子，这个例子的功能就是服务器在客户端连接到服务器时创建一个会话，然后向客户端发送一个字符串(完整的源码在附件中，这里只给出程序的简要内容)：

ServerMain:

```
public class ServerMain {

    public static void main(String[] args) throws IOException {
        SocketAddress address = new InetSocketAddress("localhost", 4321);
        IoAcceptor acceptor = new SocketAcceptor();
        IoServiceConfig config = acceptor.getDefaultConfig();

        // 配置数据的编解码器
        config.getFilterChain().addLast("codec",
                new ProtocolCodecFilter(new ObjectSerializationCodecFactory()));

        // 绑定服务器端口
        acceptor.bind(address, new ServerHandler());
        System.out.println("  服务器开始在 8000 端口监听 .....");
    }
}
```

ServerHandler:

```
public class ServerHandler extends IoHandlerAdapter {

    // 创建会话
    public void sessionOpened(IoSession session) throws Exception {
        System.out.println("  服务器创建了会话  ");
        session.write("  服务器创建会话时发送的信息 。");
    }

    // 发送信息
    public void messageSent(IoSession session, Object message) throws Exception {
    }

    // 接收信息
    public void messageReceived(IoSession session, Object message)
            throws Exception {
    }
}
```

ClientMain:

```
public class ClientMain {

    public static void main(String[] args) {

        SocketAddress address = new InetSocketAddress("localhost", 4321);
        IoConnector connector = new SocketConnector();
        IoServiceConfig config = connector.getDefaultConfig();

        // 配置数据的编解码器
        config.getFilterChain().addLast("codec",
                new ProtocolCodecFilter(new ObjectSerializationCodecFactory()));

        config.getFilterChain().addLast("logger", new LoggingFilter());

        // 连接到服务器
        connector.connect(address, new ClientHandler());
        System.out.println(" 已经连接到了服务器 " + address);
    }
}
```

ClientHandler:

```
public class ClientHandler extends IoHandlerAdapter {

    // 发送信息
    public void messageSent(IoSession session, Object message) throws Exception {
    }

    // 接收信息
    public void messageReceived(IoSession session, Object message)
            throws Exception {
        System.out.println("  客户端接收到的服务器的信息是  " + message);
    }
}
```

其中 ServerMain 和 ClientMain 分别是服务器和客户端的主程序， ServerHandler 和 ClientHandler 是服务器和客户端的数据处理句柄，关于 IoHandler 会在后面的文档中做详细的讲解，这里只是简单说明一下，IoHandler 主要是对数据进行逻辑操作，也可以理解为程序的业务逻辑层。其中：

```
// 配置数据的编解码器
        config.getFilterChain().addLast("codec",
                new ProtocolCodecFilter(new ObjectSerializationCodecFactory()));
```

这行代码的功能是将网络传输中的数据在发送时编码成二进制数据，解码时将二进制数据还

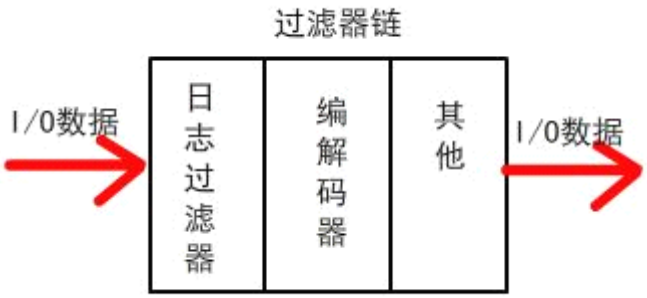
原成一个对象或者是基本类型的数据。

运行这个程序会得到如下结果：

```
已经连接到了服务器 localhost/127.0.0.1:4321
2009-7-9 23:36:46 org.apache.mina.util.SessionLog info
信息：[localhost/127.0.0.1:4321] CREATED
2009-7-9 23:36:46 org.apache.mina.util.SessionLog info
信息：[localhost/127.0.0.1:4321] OPENED
2009-7-9 23:36:46 org.apache.mina.util.SessionLog info
信息：[localhost/127.0.0.1:4321] RECEIVED:  服务器创建会话时发送的信息 。
客户端接收到的服务器的信息是  服务器创建会话时发送的信息 。
```

其中的红字部分是 LoggingFilter 打印出的事件信息。黑体部分是程序中 System.out 的输出。在 ClientMain 中的这两行代码是向过滤器链中添加 IoFilter：

```
// 配置数据的编解码器
config.getFilterChain().addLast("codec",
        new ProtocolCodecFilter(new ObjectSerializationCodecFactory()));
config.getFilterChain().addLast("logger", new LoggingFilter());//添加日志工具
```



上图表示了数据在本程序中通过过滤器链的过程，日志过滤器是根据会话(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，编解码器是根据会话的接收和发送数据来触发事件的，从这里我们也可以了解通过过滤器我们可以专门针对会话的某个或某几个状态来专门处理相关的事件，如异常事件，我们可以专门定义一个 Exception 的 IoFilter 来处理 Mina 在通信中所发生的异常信息。

还有一个比较有意思的问题是，假如我们将上面过滤器的顺序该成下面的样子：

```
config.getFilterChain().addLast("logger", new LoggingFilter());//添加日志工具
// 配置数据的编解码器
config.getFilterChain().addLast("codec",
        new ProtocolCodecFilter(new ObjectSerializationCodecFactory()));
```

程序的执行结果如下：

```
已经连接到了服务器 localhost/127.0.0.1:4321
2009-7-10 0:30:12 org.apache.mina.util.SessionLog info
信息：[localhost/127.0.0.1:4321] CREATED
2009-7-10 0:30:12 org.apache.mina.util.SessionLog info
```

```
信息：[localhost/127.0.0.1:4321] OPENED
2009-7-10 0:30:12 org.apache.mina.util.SessionLog info
信息：[localhost/127.0.0.1:4321] RECEIVED: DirectBuffer[pos=0 lim=56 cap=1024: 00 00 00 34 AC ED 00 05 74 00 2D 20
20 E6 9C 8D E5 8A A1 E5 99 A8 E5 88 9B E5 BB BA E4 BC 9A E8 AF 9D E6 97 B6 E5 8F 91 E9 80 81 E7 9A 84 E4 BF A1 E6
81 AF 20 E3 80 82]
  客户端接收到的服务器的信息是      服务器创建会话时发送的信息 。
```

很明显的是在顺序变化了之后，日志中多了接收到的二进制数据，这是因为在上面数据已经有解码器将数据还原成了 Java 对象，所以我们就看不到二进制数据了，而在顺序变换后，由于先执行的是打印信息，此时的数据还没有还原成 java 对象，所以接收到的数据是二进制的。

在上面的例子中我们清楚了整个 IoFilter 或者是 IoFilter 的工作流程，那么 IoFilter 在 Mina 中的作用如何？所有的数据在发送到 Mina 程序中时，数据都是先通过 IoFilter，经过处理后再转发到业务层。这里 IoFilter 就起到了一个承上启下的作用。

到这里我们就可以回答本文开始提到的问题了：

（1）什么时候需要用到 IoFilter，如果在自己的应用中不添加过滤器可以吗？

在你自己的程序中可以添加过滤器，也可以不添加，但是在数据发送之前，所发送的数据必须转换成二进制数据，这个可以有 IoFilter 完成，也可以由 ProtocolCodecFilter 完成(关于这个问题会在后面的文章中详细讲述)，否则 Mina 会抛出 Write requests must be transformed to class org.apache.mina.common.ByteBuffer: 异常。这是因为网络中传输的数据只能是二进制数据。因此无论添加不添加过滤器，都必须将要发送的数据转换成二进制数据。

（2）如果在 IoService 中添加多个过滤器可以吗？若可以，如何进行添加，这多个过滤器是如何工作的？

在 IoService 中可以添加多个过滤器，这个在上面的程序中已经给处理，添加的方式也很简单，通过程序一目了然。

（3）Mina 中提供了协议编、解码器，IoFilter 也可以实现 IO 数据的编解码功能，在实际的使用中如何选择？

Mina 的编解码器在 Mina 的使用中是最关键的一个问题，特别是在不同语言之间进行通信的时候，比如 Java 和 C/C++等，由于 Mina 自身没有提供这些编解码器，所以需要自己来实现。Mina 提供了一个 Decoder/Encoder，你可以实现两个类来完成不同平之间的通信。关于这个问题会在后面的文档给出具体的实习方法。

至此，关于 IoFilter 的作用就讲述完了，希望对你能有所帮助。:)

深入理解 Apache Mina ---- 与 IoHandler 相关的几个类

在上一篇文档中我们已经了解了 IoFilter 的用法和其在 Mina 中的作用，作为 Mina 数据传输过程中比较重要的组件，IoFilter 起到了承上启下的作用----接收数据，编/解码，将数据传递到逻辑层，当数据传递地到逻辑层时，IoFilter 的使命就完成了，那么逻辑层的数据由谁来处理呢？如何处理的？这就是本文要讲述的内容----IoHandler。

在介绍 IoFilter 的时候，文中首先是从 IoFilter 的结构和其在 Mina 中的作用谈起的，最后添加了一个使用 IoFilter 的例子，之前我将其传给几个同学看时，感觉这种方式比较晦涩，应该将例子提到前面，由于时间的关系我不能在对 IoFilter 的介绍做过多的修改，所以在本篇文档中我就先以一个例子开头，介绍 IoHandler，希望这种讲述方式能对你理解 Mina 有更多的帮助。

好了，言归正传，我们的例子还是以上篇文档中的 IoFilter 的例子为基础，在此基础上着重突出 IoHandler 的作用。

ServerMain:

```
public class ServerMain {
    public static void main(String[] args) throws IOException {
        SocketAddress address = new InetSocketAddress("localhost", 4321);
        IoAcceptor acceptor = new SocketAcceptor();
        IoServiceConfig config = acceptor.getDefaultConfig();

        // 配置数据的编解码器
        config.getFilterChain().addLast("codec",
                new ProtocolCodecFilter(new ObjectSerializationCodecFactory()));
        config.getFilterChain().addLast("logger", new LoggingFilter());

        // 绑定服务器端口
        acceptor.bind(address, new ServerHandler());
        System.out.println("  服务器开始在  8000  端口监听  ……");
    }
}
```

ServerHandler:

```
public class ServerHandler extends IoHandlerAdapter {
    // 创建会话
    public void sessionOpened(IoSession session) throws Exception {
        System.out.println("  服务器创建了会话  ");
        session.write("  服务器创建会话时发送的信息  。");
    }

    // 发送信息
```

```
    public void messageSent(IoSession session, Object message) throws Exception {
    }

    // 接收信息
    public void messageReceived(IoSession session, Object message)
            throws Exception {
    }
}
```

ClientMain:

```
public class ClientMain {

    public static void main(String[] args) {
        SocketAddress address = new InetSocketAddress("localhost", 4321);
        IoConnector connector = new SocketConnector();
        IoServiceConfig config = connector.getDefaultConfig();

        // 配置数据的编解码器
        config.getFilterChain().addLast("codec",
                new ProtocolCodecFilter(new ObjectSerializationCodecFactory()));
        config.getFilterChain().addLast("logger", new LoggingFilter());

        // 连接到服务器
        connector.connect(address, new ClientHandler());
        System.out.println(" 已经连接到了服务器 " + address);
    }
}
```

ClientHandler:

```
public class ClientHandler extends IoHandlerAdapter {

    // 发送信息
    public void messageSent(IoSession session, Object message) throws Exception {
    }

    // 接收信息
    public void messageReceived(IoSession session, Object message)
            throws Exception {
        System.out.println("  客户端接收到的服务器的信息是  " + message);
    }
}
```

上面给出里这个例子中的主要的代码，当先后启动服务器和客户端后，服务器会在客户端连接到服务器后发送一个字符串的消息。这个消息的发送就是在 IoHandler 中发送的。

IoHandler 在 Mina 中属于业务层，这里的 IoHandler 更相是 J2EE 中的 Servlet 的作用，在 IoHandler 中你可以不用考虑底层数据的封装和转换，前提是你已经在 IoFilter 中已经完成了数据的转换。这里需要提到的一个是，所谓数据的转换，是指将二进制数据转换成 Java 中的可用对象或者是基本类型的数据。由于网络传输中传输的都是二进制数据，这就需要有一个专门的数据转换层，就 Mina 中的编解码器来实现这个功能。如果使用 RMI，对于这个问题应该不陌生，二进制和对象之间的转化过程其实就是对象的序列化和反序列化的过程。关于 Mina 中实现对象的序列化和反序列化会在后续的文档中详细介绍，在此不在赘述。

既然 IoHandler 是逻辑层，我们就用 IoHandler 实现一个简单的逻辑实现。先听一个小故事：一个淘气的小孩要去 KFC 买汉堡，由于 KFC 生意比较好，人比较多，服务员忙不过来，于是 KFC 专门设立了一个自动售汉堡的机器，这个机器只是一个简单的数据收发装置，（由于汉堡的价格时常变化，所以价格会实时更新，因此该机器需要和 KFC 的汉堡的价格服务器相连）小朋友买汉堡时只要向机器中投入硬币，机器就会查询服务器，看价格是否符合，若符合，则送给小朋友一个汉堡，若不符合则提示小朋友钱不够，买不到这个汉堡。

上面是我自己虚构的一个小故事，我们先不管现实中的 KFC 是如何运作的，我们就当故事是真的了，那么现在这个小的项目分配给了我们，我们需要抽象出它的需求：
（1）客户端要向服务器发送数据，查询价格，根据价格是否合理给出相应的显示
（2）服务器接收客户度的价格查询请求，根据服务器中存储的价格信息，返回响应的结果。
根据上面的需求，我们使用 Mina 来实现上面的服务器和客户端，程序的代码如下(完整代码在附件中)：

KFCFoodPriceHandler(服务器句柄):

```
public class KFCFoodPriceHandler extends IoHandlerAdapter {
	// 创建会话
	public void sessionOpened(IoSession session) throws Exception {
//		System.out.println("  服务器创建了会话  ");
	}

	// 接收信息
	public void messageReceived(IoSession session, Object message)
			throws Exception {
		HashMap<String, Object> map = (HashMap<String, Object>) message;
		String buythings = (String) map.get("购买");
//		System.out.println("  服务器接收到的信息  " + buythings);
		if (buythings.equals("汉堡")) {
			HashMap<String, Object> map2 = new HashMap<String, Object>();
			map2.put("食品", "汉堡");
			map2.put("价格", 4);
			session.write(map2);
		} else if (buythings.equals("鸡翅")) {
			HashMap<String, Object> map2 = new HashMap<String, Object>();
			map2.put("食品", "鸡翅");
			map2.put("价格", 5);
```

```
                session.write(map2);
            } else {
                session.write(" 该种物品已经出售完毕，谢谢惠顾！ ");
            }
        }
}
```

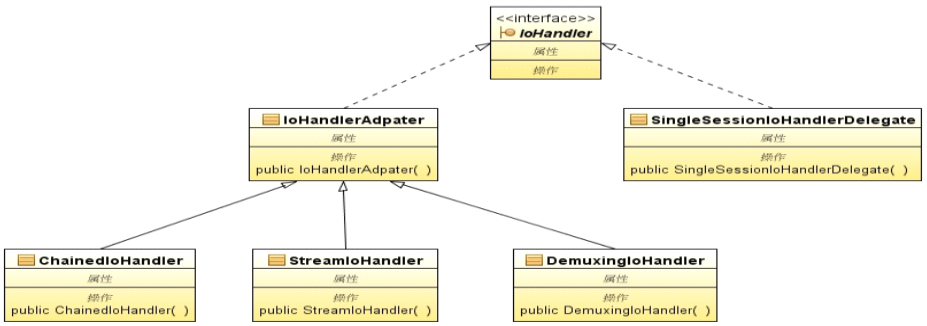
KFCSellerHandler (客户端句柄):

```
public class KFCSellerHandler extends IoHandlerAdapter {

    private Integer childInputMoney_Ham = 4;
    private Integer childInputMoney_Chick = 4;
    // 创建会话
    public void sessionOpened(IoSession session) throws Exception {

        HashMap<String, Object> map = new HashMap<String, Object>();
        map.put("购买","汉堡");
        session.write(map);
    }

    // 接收信息
    public void messageReceived(IoSession session, Object message)
            throws Exception {
//      System.out.println("  客户端接收到的服务器的信息是  "
//              + (HashMap<String, Object>) message);
        HashMap<String, Object> priceInfor = (HashMap<String, Object>) message;
//      System.out.println("==========" + priceInfor.get("食品"));
        String foodName = (String) priceInfor.get("食品");
        if (foodName.equals("汉堡")) {
            Integer foodPrice = (Integer) priceInfor.get("价格");
            if (foodPrice.equals(childInputMoney_Ham)) {
                System.out.println("  您好，请收好你的汉堡，欢迎下次光临！ ");
            } else {
                System.out.println("  对不起，你投如的钱币数量不够，钱已经如数归还，请收好！ ");
            }
        } else if (foodName.equals("鸡翅")) {
            Integer foodPrice = (Integer) priceInfor.get("价格");
            if (foodPrice.equals(childInputMoney_Chick)) {
                System.out.println("  您好，请收好你的汉堡，欢迎下次光临！ ");
            } else {
                System.out.println("  对不起，你投如的钱币数量不够，钱已经如数归还，请收好！ ");
            }
        }
```

```
    }
}
```

通过上面的程序我们可以看出 Mina 的中业务逻辑处理都可以在 IoHandler 中，而不需要考虑对象的序列化和反序列化问题。关于 IoHandler 的简单用法就说这么多。下面再看看与 IoHandler 相关的几个中要的类。

按照惯例，还是先给出 IoHandler 及其相关类的类图:



从上面的类图我们可以清晰的看到 IoHandler 是一个接口，它有两个子类：

IoHandlerAdpater：它只是提供了 IoHandler 中定义的方法体，没有任何的逻辑处理，你可以根据你自己的需求重写该类中的相关方法。这个类在实际的开发中使用的是较多的。我们上面写的例子都是继承于这个类来实现的。

SingleSessionIoHandlerDelegate：这是一个服务器和客户端只有一个会话时使用的类，在该类的方法中没有提供 session 的参数，该类在实际的开发中使用的较少，如果需要对该类进行更深入的了解，请参考Mina 1.1.7 的 API 文档。

在 Mina 提供的 IoHandler 的具体实现中，大部分的实现类都是继承与 IoHandlerApater，IoHandlerAdpater 在 Mina 1.1.7 中的子类有三个：

ChainedIoHandler：这个类主要是用于处理 IoHandler 的messageReceived 事件，它和 IoHandlerChain 配合使用。当在业务逻辑中有多个 IoHandler 需要处理时，你可以将你的每个 IoHandler 添加到 IoHandlerChain 中，这个和过滤器链比较相似，关于 IoFilter 和 IoHandlerChain 的具体用法和区别会在后续的文档中给出。

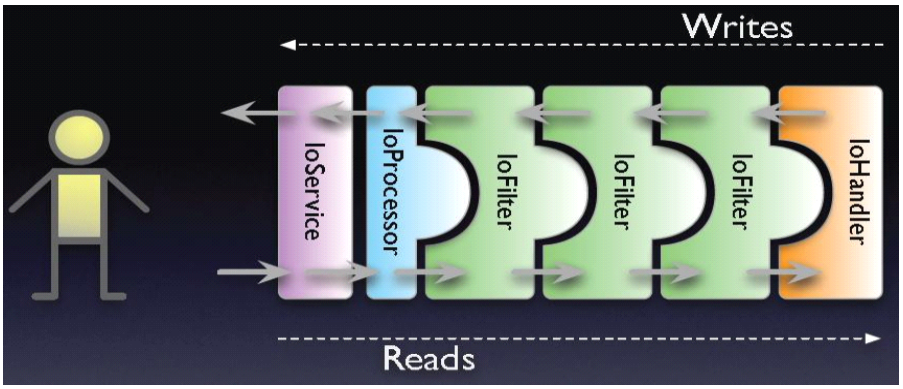
StreamIoHandler：该类也是用于处理 IoHandler 的 messageReceived 事件，它主要用于文件传输的系统中，比如 FTP 服务器中，如果需要对该类进行更深入的了解，请参考 Mina 1.1.7 的 API 文档。

DemuxingIoHandler：该类主要是用于处理多个 IoHandler 的 messageReceived ，由于在 TCP/IP 协议的数据传输中会出现数据的截断现象(由于 socket 传输的数据包的长度是固定的，当数据包大于该长度，数据包就会被截断)，所以提供这个类主要是保证 IoHandler 所处理的数据包的完整性，这个和编解码器中的 CumulativeProtocolDecoder 类似，关于这两个类的具体介绍会在后续的文档中给出。

深入理解 Apache Mina ---- IoFilter 和 IoHandler 的区别和联系

在《与 IoFilter 相关的几个类》和《与 IoHandler 相关的几个类》两篇文档中我们了解了 IoFilter 和 IoHandler 的基本用法，以及其相关类的作用和用途。在本文中主要探讨 IoFilter 和 IoHandler 的主要区别和联系。

在上面的两篇文档中都提到了 IoFilter 和 IoHandler 都是对服务器或客户端(IoAcceptor/IoConnector)接收到的数据进行处理。在 Mina 的官方文档《The high-performance protocol construction toolkit》给出了 IoFilter 和 IoHandler 在 Mina 数据传输中的执行顺序，如下图：



上图显示了 IoService 进行数据读写时，各主要组件的执行顺序：
（1）IoService 读取数据时个组件的执行顺序是：IoProcessor-->IoFilter-->IoHandler。
（2）IoService 发送数据时的执行数顺序：IoHandler-->IoFilter-->IoProcessor。

IoProcessor 是一个处理线程，它的主要作用是根据当前连接的状态的变化(创建会话、开启会话、接收数据、发送数据、发生异常等等)，来将数据或事件通知到 IoFilter，当 IoFilter 的相应的方法接收到该状态的变化信息是会对接收到的数据进行处理，处理完毕后会将该事件转发到 IoHandler 中，有 IoHandler 完成最终的处理。在这里 IoProcessor 的主要功能是创建资源(创建/分配线程给 IoFilter)和数据转发（转发到 IoFilter），IoFilter 对数据进行基本的分类（如编解码），IoHandler 则负责具体的逻辑实现。也就是说 IoFilter 对接收到的数据包的具体内容不做处理，而是有 IoHandler 来对所接收到的数据包进行处理，根据数据包的内容向客户端返回响应的信息。

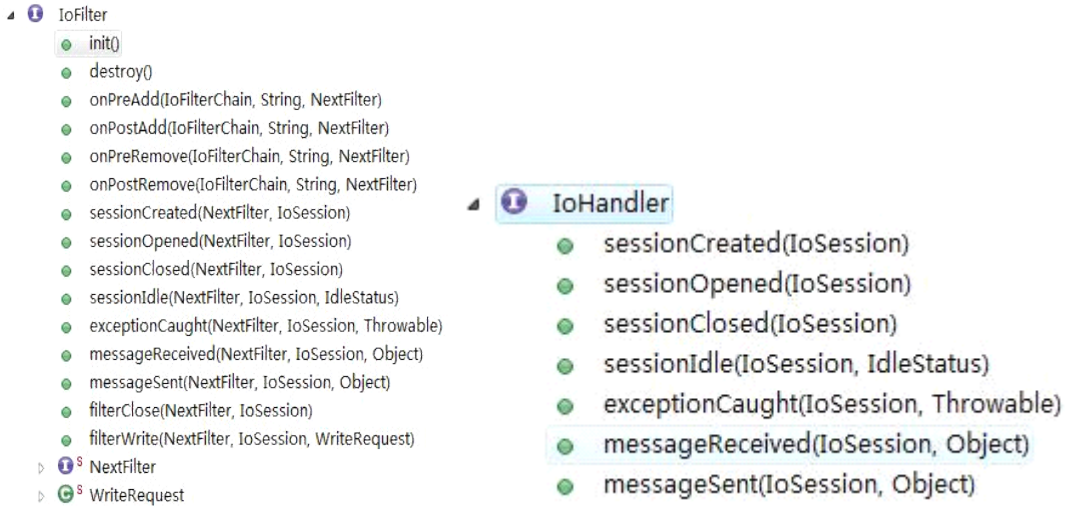
我们以《与 IoHandler 相关的几个类》中 KFC 售货机的例子来做一个具体的解释，在该例子中，客户端需要想服务器发送查询价格的请求，服务器根据接收到的请求查询物品的价格，然后将该物品的价格返回到客户端。客户端在向服务器发送数据前会有 IoFilterr 将发送的信息序列化为二进制数据，然后有 IoProcess 发送出去，简化如下：

IoHandler 发送客户端数据-->IoFilter 进行序列化-->IoProcessor

上面是数据的发送过程，当服务器接收到客户端的的请求数据后，先有 IoProcessor 将该数据转发到 IoFilter，IoFilter 将对象进行反序列化，反序列化的结果完成后将数据转发到

IoHandler 中，过程简化如下：
IoProcessor 接收客户度端的数据-->IoFilter 进行反序列化-->IoHandler 根据请求查询价格这样一个完整的数据请求的过程就完成了。

上面简单介绍了 IoFilter 和 IoHandler 在 Mina 中的作用，前者是数据的转换层，后者是业务层。但是两者在很多地方都有相似之处，为了将两者的区别做更详细的讨论，先给出两者的结构图：



图中的 IoFilter 比 IoHandler 中多出的一个最重要的方法就是 filterWriter()，该方法会在程序调用 session.write()的时候触发，该方法的重要之处就在于它表明了 IoFilter 和 IoHandler 的重要区别，即进行 IoFilter 是数据的收发层，也可以说是一个数据的收发器，而 IoHandler 则是逻辑层，并不负责数据的收发，如果把 IoProcessor 说成是底层的数据收发层，则 IoFilter 则是一个上层的数据收发层。关于 IoFilter 中 on*()的方法的使用和作用请参考帮助文档，这里不再给出具体的解释。

到此我们就可以明白了 IoFilter 是一个数据收发和转化的装置，而 IoHandler 则是一个单一的业务处理装置，你的所有业务逻辑都应该写在这个类中。

如果没有在 IoService 中配置 IoFilter，那么在 IoHandler 中接收到的数据是一个 ByteBuffer，你需要在你的 IoHandler(业务层)中完成数据的转化，但是这样就破坏了 Mina 中各个组件层的关系，这样你的程序结构就不在清晰，因此建议在使用 Mina 时将数据的转化(即二进制与对象之间的转换放在 IoFilter 层来处理)。在 Mina 中必须要配置 IoHandler，因为 Mina 中提供的 IoService 中的 bind 方法必须要有一个 IoHandler，因此 IoHandler 不能省略。

到这里对于 IoFilter 和 IoHandler 的内容已经讲述完毕，下面的内容是对我在开发中遇到的一些问题的一些总结，顺便也给自己以前的问题写出答案：
(1)IoHandler 和 IoHandlerCommand 的区别和联系。
IoHandler 和 IoHandlerCommand 是两个接口，在开发中经常遇到的他们两个实现类分别是 IoHandlerAdpater 和 IoHandlerChain ， IoHandlerAdpater 的子类 ChainedIoHandler 和 IoHandlerChain 结合使用可以实现多个逻辑功能， IoHandlerChain （代表 IoHandlerCommand ）是业务逻辑的处理单元，而 ChainedIoHandler （代表 Iohandler ）则是处理这些逻辑单元的组件。因此它们的区别是： IoHandler 是刀俎，

而 IoHandlerCommand 则是鱼肉。他们的一般用法如下：

```
IoHandlerChain chain = new IoHandlerChain();// 创建逻辑处理组件
chain.addLast("first", new FistCommand);// 添加逻辑组件单元一
chain.addLast("second", new SecondCommand);// 逻辑组件单元二
ChainedIoHandler chained = new ChainedIoHandler(chain);// 创建逻辑组件执行模块
chained.messageReceived(session, message);// 当 messageReceived 触发该事件
```

(2)IoFilter 和 IoHandler 可以同时使用吗？

IoFilter 和 IoHandler 由于分工不同，因此他们需要同时使用，但是这不是绝对的，在 Mina 的 IoService 中可以不配置 IoFilter ，但是必须配置 IoHandler 。但是，这不是提倡的方式，因为这破坏的 mina 的分层结构，因此建议在使用 Mina 的时候同时使用 IoFilter 和 IoHandler 。

(3)IoFilter 和 IoHandlerCommand/IoHandler 的区别和联系。

这个问题的答案请参考问题（1）和（2）给出的解释。

(4)IoHandlerAdpater 和 IoFilterAdpater 的区别和联系。

IoHandlerAdpater 和 IoFilterAdpater 一个是业务逻辑层的监听器，一个数据传输层的监听器，他们的区别就是 IoHandler 和 IoFilter 的区别，这个在上面已经讨论清楚了，不在详细说明。

(5)IoFilterChainBuilder 和 ChainedIoHandler 的区别和联系。

深入理解 Apache Mina ---- 配置 Mina 的线程模型

在 Mina 的使用中，线程池的配置一个比较关键的环节，同时它也是 Mina 性能提高的一个有效的方法，在 Mina 的 2.0 以上版本中已经不再需要对 Mina 线程池的配置了，本系列文章都是基于当前的稳定版本 Mina 1.1.7 版来进行讲述的，Mina 的 2.0 以上版本现在还都是 M(millestone,即里程碑)版的，在 1.5 版本上 2.0M 版为稳定版本，但是在 1.5+以上则为非稳定版本，所以，为了更好的进行讨论和学习，还是基于 Mina 1.1.7 版本进行讨论，如果使用 Mina 2.0 进行开发要注意 JDK 的版本问题，当然如果有能力的话也可以自行修改和编译 Mina 的 2.0 版本，这里对此就不再多说，使用 2.0 版本的同学可以不用理会本文的内容。上面的内容都是基于 Apache Mina 提供的文档讲述，如有需要，请自行查找相关资料，在此不再赘述。

下面开始对 Mina 的线程模型的配置、使用、及 ExcutorFilter 的基本原理进行简单的讲解。

（一）配置 Mina 的三种工作线程

在 Mina 的 NIO 模式中有三种 I/O 工作线程（这三种线程模型只在 NIO Socket 中有效，在 NIO 数据包和虚拟管道中没有，也不需要配置）：

(1)Acceptor thread

该线程的作用是接收客户端的连接，并将客户端的连接导入到 I/O processor 线程模型中。所谓的 I/O processor 线程模型就是 Mina 的 I/O processor thread。Acceptor thread 在调用了 Acceptor.bind()方法后启动。每个 Acceptor 只能创建一个 Acceptor thread，该线程模型不能配置，它由 Mina 自身提供。

(2)Connector thread

该线程模型是客户端的连接线程模型，它的作用和 Acceptor thread 类似，它将客户端与服务器的连接导入到 I/O processor 线程模型中。同样地，该线程模型也是由 Mina 的客户端自动创建，该线程模型也不能进行配置。

(3)I/O processor thread

该线程模型的主要作用就行接收和发送数据，所有的 IO 操作在服务器与客户端的连接建立后，所有的数据的接收和发送都是有该线程模型来负责的，知道客户端与服务器的连接关闭，该线程模型才停止工作。该线程模型可以由程序员根据需要进行配置。该线程模型默认的线程的数量为 cpu 的核数+1 。若你的 cpu 为双核的，则你的 I/O processor 线程的最大数量为 3，同理若你的若你的 cpu 为四核的，那么你的 I/O processor 线程的最大数量为 5。

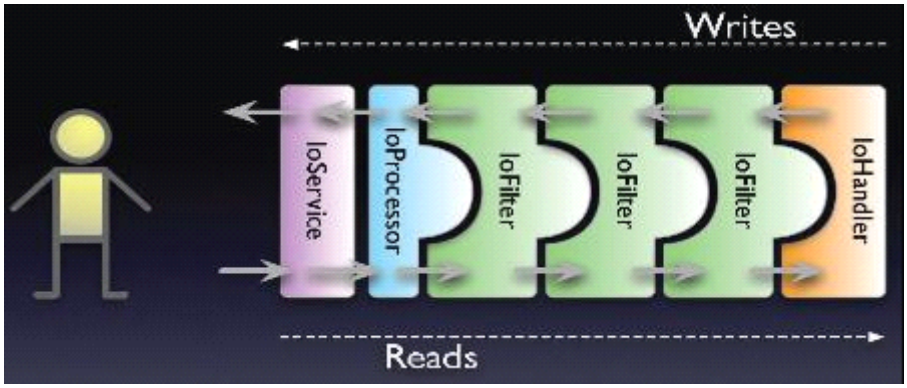
由上面的内容我们可以知道在 Mina 中可以配置的线程数量只有 I/O processor，对于每个 IoService 再创建其实例的时候可以配置该 IoService 的 I/O processor 的线程数量。在 SokcetConnector 和 SocketAccpetor 中 I/O Processor 的数量是由 CPU 的核数+1 来决定的。

他们的配置方式如下：

```
/***
 * 配置SocketAcceptor 监听器的I/O Processor的线程的数量，
 * 此处的I/O Processor的线程数量由CPU的核数决定，但Acceptor
 * 的线程数量只有一个，也就是接收客户端连接的线程数只有一个，
```

```
     * Acceptor的线程数量不能配置。
     **/
    SocketAcceptor acceptor = new SocketAcceptor(Runtime.getRuntime()
            .availableProcessors() + 1, Executors.newCachedThreadPool());

    /***
     * 配置SocketConnector监听器的I/O Processor 的线程的数量,
     * 此处的I/O Processor的线程数量由CPU的核数决定，但SocketConnector
     * 的线程数量只有一个， 也就是接收客户端连接的线程数只有一个，
     * SocketConnector 的线程数量不能配置。
     **/
    SocketConnector connector = new SocketConnector(Runtime.getRuntime()
            .availableProcessors() + 1, Executors.newCachedThreadPool());
```

在上面的配置比较难以理解的地方就是 Runtime.getRuntime().availableProcessors() + 1，它的意思就是由 JVM 根据系统的情况(即 CPU 的核数)来决定 IO Processor 的线程的数量。虽然这个线程的数量是在 SocketAcceptor /SocketConnector 的构造器中进行的，但是对于 SocketAcceptor /SocketConnector 自身的线程没有影响，SocketAcceptor /SocketConnector 的线程数量仍然为 1。因为 SocketAcceptor /SocketConnector 本身就封装了 IO Processor，SocketAcceptor /SocketConnector 只是由一个单独的线程来负责接收外部连接/向外部请求建立连接，当连接建立后，SocketAcceptor /SocketConnector 会把数据收发的任务转交 I/O Processor 的线程。这个在本系列文章的《IoFilter 和 IoHandler 的区别和联系》中的图示中可以看。



图中清晰的显示了 IO Processor 就是位于 IoService 和 IoFilter 之间，IoService 负责和外部建立连接，而 IoFilter 则负责处理接收到的数据，IoProcessor 则负责数据的收发工作。

关于配置 IO Processor 的线程数量还有一种比较"笨"的办法，那就一个一个试，你可以根据你的 PC 的硬件情况从 1 开始，每次加 1，然后得出 IO Processor 的最佳的线程的数量。但是这种方式个人建议最好不要用了，上面的方法足矣。配置方法如下：

```
//从 1--N 开始尝试，N 的最大数量为 CPU核数+1
SocketAcceptor acceptor = new SocketAcceptor(N, Executors.newCachedThreadPool());
```

（二）为 Mina 的 IoFilterChain 添加线程池

在 Mina 的 API 中提供了一个 ExecutorFilter，该线程池实现了 IoFilter 接口，它可以作为一

个 IoFilter 添加到 IoFilterChain 中，它的作用就是将 I/O Processor 中的事件通过其自身封装的一个线程池来转发到下一个过滤器中。在没有添加该线程模型时，I/O Processor 的事件是通过方法来触发的，然后转发给 IoHandler。在没有添加该线程池的时候，所有的事件都是在单线程模式下运行的，也就是说有的事件和处理(IO Processor，IoHandler，IoFilter)都是运行在同一个线程上，这个线程就是 IO Processor 的线程，但是这个线程的数量受到 CPU 核数的影响，因此系统的性能也直接受 CPU 核数的影响。

比较复杂的应用一般都会用到该线程池，你可以根据你的需求在 IoFilterchain 中你可以添加任意数量的线程池，这些线程池可以组合成一个事件驱动(SEDA)的处理模型。对于一般的应用来说不是线程的数量越多越好，线程的数量越多可能会加剧 CPU 切换线程所耗费的时间，反而会影响系统的性能，因此，线程的数量需要根据实际的需要由小到大，逐步添加，知道找到适合你系统的最佳线程的数量。ExcutorFilter 的配置过程如下：

```
SocketAcceptor acceptor = ...;
DefaultIoFilterChainBuilder filterChainBuilder = acceptor.getDefaultConfig().getFilterChain();
filterChainBuilder.addLast("threadPool", new ExecutorFilter(Executors.newCachedThreadPool());
```

在配置该线程池的时候需要注意的一个问题是，当你使用自定的 ProtocolCodecFactory 时候一定要将线程池配置在该过滤器之后，如下所示：

```
DefaultIoFilterChainBuilder filterChainBuilder = acceptor.getDefaultConfig().getFilterChain();
// 和 CPU 绑定的操作配置在过滤器的前面
filterChainBuilder.addLast("codec", new ProtocolCodecFactory(...));
// 添加线程池
filterChainBuilder.addLast("threadPool", new ExecutorFilter(Executors.newCachedThreadPool());
```

因为你自己实现的 ProtocolCodecFactory 直接读取和转换的是二进制数据，这些数据都是由和 CPU 绑定的 I/O Processor 来读取和发送的，因此为了不影响系统的性能，也应该将数据的编解码操作绑定到 I/O Processor 线程中，因为在 Java 中创建和线程切换都是比较耗资源的，因此建议将 ProtocolCodecFactory 配置在 ExecutorFilter 的前面。关于 ProtocolCodecFactory 详细讲述会在后续的文档中给出，此处就不多说了。

最后给出一个服务器线程模型完整配置的例子，该例子和 KFCClient 一起配置使用，详细代码在附件中，此处只给出代码的主要部分：

```
SocketAddress address = new InetSocketAddress("localhost", 4321);
/***
 * 配置SocketAcceptor 监听器的I/O Processor的线程的数量，此处的I/O
 * Processor的线程数量由CPU的核数决定，但 Acceptor 的线程数量只有一个，也就
是接收客户端连接的线程数只有一个，
 * Acceptor的线程数量不能配置。
 **/
IoAcceptor acceptor = new SocketAcceptor(Runtime.getRuntime()
        .availableProcessors() + 1, Executors.newCachedThreadPool());

acceptor.getDefaultConfig().setThreadModel(ThreadModel.MANUAL);
// 配置数据的编解码器
acceptor.getDefaultConfig().getFilterChain().addLast("codec",
        new ProtocolCodecFilter(new ObjectSerializationCodecFactory()));
```

```
            // 此处为你自己实现的编解码器
            // config.getFilterChain().addLast("codec", new
            // ProtocolCodecFactory(...));

            // 为IoFilterChain添加线程池
            acceptor.getDefaultConfig().getFilterChain().addLast("threadPool",
                    new ExecutorFilter(Executors.newCachedThreadPool()));
            acceptor.getDefaultConfig().getFilterChain().addLast("logger",
                    new LoggingFilter());
            // 绑定服务器端口
            acceptor.bind(address, new KFCFoodPriceHandler());
            System.out.println("  服务器开始在  8000  端口监听  ......");
            // ========================================//
            // 此处为客户端的I/O Processor线程数的配置，你可以模仿 //
            // IoAcceptor 配置来实现 //
            // ========================================//
            /***
             * 配置SocketConnector监听器的I/O Processor 的线程的数量，此处的I/O
             * Processor 的线程数量由CPU的核数决定，但SocketConnector
             * 的线程数量只有一个，也就是接收客户端连接的线程数只有一个，
SocketConnector的线程数量不能配置。
             **/
            // SocketConnector connector = new SocketConnector(Runtime.getRuntime()
            // .availableProcessors() + 1, Executors.newCachedThreadPool());
        }
```

深入理解 Apache Mina ---- Java Nio ByteBuffer 与 Mina ByteBuffer 的区别

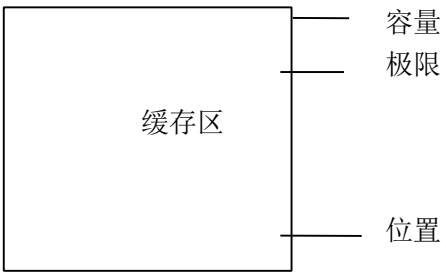
为了对后续关于 Mina 的 ProtocolFilter(编解码器)的编写有一个更好的理解，本文讲述一下关于 Mina ByteBuffer 和 Java Nio ByteBuffer 的区别。关于 Java Nio ByteBuffer 和 Mina ByteBuffer 及其子类的类图在附件中都已经给出了。因为 Mina 的 ByteBuffer 在 Mina 2.0 以上的版本中都改称 IoBuffer。为了使后文关于 ByteBuffer 的名字不致混淆，Mina ByteBuffer 都统称 IoBuffer，Java Nio ByteBuffer 统称 ByteBuffer。关于 IoBuffer 中的对 ByteBuffer 扩展及一些重要的方法都在 IoBuffer 的类图中用红色方框标出。详细的信息请参考附件中。

在开始对 IoBuffer 的讨论前，先简单的讲述一下 ByteBuffer 的用法。IoBuffer 是对 ByteBuffer 的一个封装。IoBuffer 中的很多方法都是对 ByteBuffer 的直接继承。只是对 ByteBuffer 添加了一些扩展了更加实用的方法。

(1)ByteBuffer 简介
ByteBuffer 继承于 Buffer 类，ByteBuffer 中存放的是字节，如果要将它们转换成字符串则需要使用 Charset，Charset 是字符编码。它提供了把字节流转换成字符串(解码)和将字符串转换成字节流(编码)的方法。这个和后面讲述的 Mina 的编解码的工作原理类似。对 ByteBuffer 的访问可以使用 read()，write()等方法。

ByteBuffer 有一下三个重要的属性：
1) 容量(capacity)：表示该缓存区可以存放多少数据。
2) 极限(limit)：表示读写缓存的位置，不能对超过位置进行数据的读或写操作。
3) 位置(position)：表示下一个缓存区的读写单元。每读写一次缓存区，位置都会变化。位置是一个非负整数。

ByteBuffer 的这三个属性相当于三个标记位，来表示程序可以读写的区域：



上图简单的表示了容量、极限、位置在缓存区中的位置。其中极限只能标记容量以内的位置，即极限值的大小不能超过容量。同样位置是用来标记程序对缓存区进行读或写操作的开始位置。程序只能在极限以内的范围进行读写，即读写操作不能超过极限的范围，所以位置值的大小也不能超过极限。三者的大小关系为：容量>极限>位置>=0。

上面说到 ByteBuffer 的三个属性只是缓存区的标记位置。那么如何改变这些标记的位置呢？ByteBuffer 提供了一下三种方法来改变上面的属性值。

1)clear():极限设置为容量，位置设为 0。
2)Flip():极限设为位置，位置设为 0。
3)Rewind():不改变极限，位置设为 0。

```
// JDK没有提供ByteBuffer的公开构造方法只能通过该
// 方法来创建一个缓存区。
ByteBuffer buffer = ByteBuffer.allocate(1024);
// ========测试缓存读写一个字符串=======//
 String userName ="chinaestone";
 char[] charArray = userName.toCharArray();

 System.out.println("这是往缓存中存放的 字符串");
 for(int i=0;i<charArray.length;i++){
        System.out.println(charArray[i]);
        buffer.putChar(charArray[i]);
 }

 buffer.limit(buffer.position());
 buffer.position(0);

 System.out.println();
 System.out.println("这是缓存中取出来的 字符串");
 while(buffer.hasRemaining()){
        System.out.println(buffer.getChar());
 }
```

上面只是一个简单的演示程序，功能是实现对字符串的读写，比较"笨"，呵呵。关于如何向 ByteBuffer 读写字符串会在 IoBuffer 中详细讲解。

(2)IoBuffer 简介
IoBuffer 是对 ByteBuffer 的扩展，并不是和 ByteBuffer 毫无关系的。对 Mina 或者 Socket 应用来说，ByteBuffer 提供的方法存在一下不足：
1)它没有提供足够可用的 put 和 set 方法，例如：fill、get/putString、 get/putAsciiInt()等。
2)很难将可变长度的数据放入 ByteBuffer。
基于以上的缺点，Mina 提供了 IoBuffer 来补充了 ByteBuffer 的不足之处。
Let's drink code,来看看 Mina 的 IoBuffer 是如何读写字符串的。

```
// 获取一个容量为1024字节的ByteBuffer
ByteBuffer buffer = ByteBuffer.allocate(1024);
// 设置系统字符集为utf-8
Charset ch =Charset.forName("utf-8");
// 获取utf-8的编码器
CharsetEncoder encoder = ch.newEncoder();
// 获取utf-8的解码器
CharsetDecoder decoder = ch.newDecoder();
```

```
System.out.println(buffer.remaining());
// 进行编码的字符串
String cs = "中國壹石頭";
// 将字符串编码后放入缓存
buffer.putString(cs,encoder);
System.out.println(buffer.remaining());
// 将缓存的位置设为位置
buffer.limit(buffer.position());
// 将缓存的位置设为0
buffer.position(0);

// 读取缓存中的字符串
String str = buffer.getString(decoder);
// 打印输出缓存中的信息
System.out.println(str);
System.out.println(buffer.remaining());
```

注意此处用到了 Charset，它的作用在上面已经说道，它主要用来进行编解码的，因此对字符串进行编码和解码时注意要使用相同的编码。

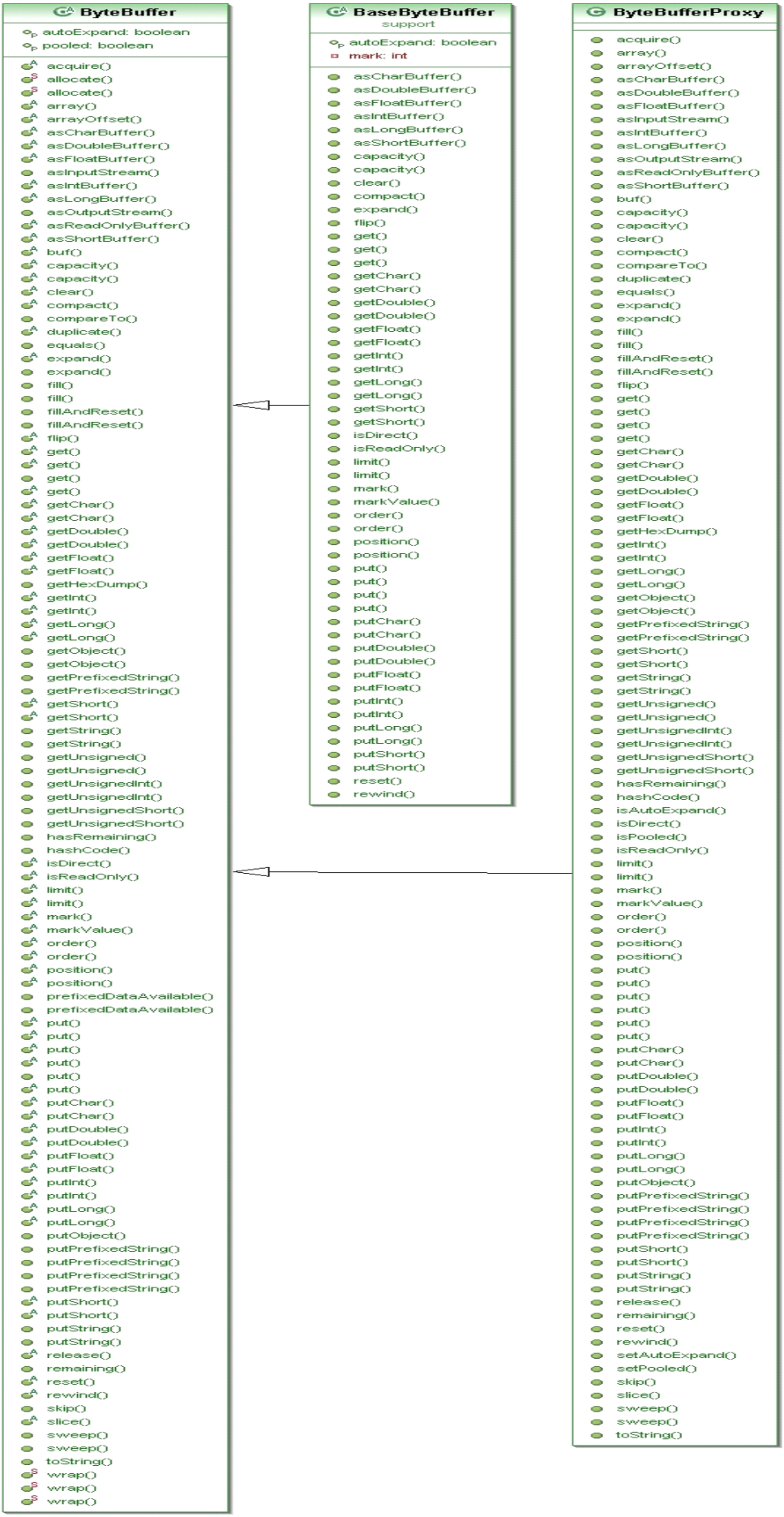
(3)IoBuffer 的子类

为了更好的使用 IoBuffer 进行开发，IoBuffer 提供了两个子类 BaseByteBuffer 和 ByteBufferProxy。BaseByteBuffer 实现了 IoBuffer 中定义的绝大多数方法。如果你在实际开发中要扩展适合于自己的方法时可以继承该类，因为它可以使你的实现更加简单。ByteBufferProxy 中封装了一个 IoBuffer，所有对 ByteBuffer 的操作都可以通过该类提供的方法来实现。

本文只是简单的介绍了 IoBuffer 和 ByteBuffer 的基本知识，如果需要了解 IoBuffer 更多的信息请参考 Mina 的帮助文档和 Mina 的源码。


ByteBuffer
java.nio
bigEndian: boolean
hb: byte[] [1..*]
isReadOnly: boolean
nativeByteOrder: boolean
offset: int
ByteBuffer()
ByteBuffer()
_get()
_put()
allocate()
allocateDirect()
array()
arrayOffset()
asCharBuffer()
asDoubleBuffer()
asFloatBuffer()
asIntBuffer()
asLongBuffer()
asReadOnlyBuffer()
asShortBuffer()
compact()
compareTo()
compareTo()
duplicate()
equals()
get()
get()
get()
get()
getChar()
getChar()
getDouble()
getDouble()
getFloat()
getFloat()
getInt()
getInt()
getLong()
getLong()
getShort()
getShort()
hasArray()
hashCode()
isDirect()
order()
order()
put()
put()
put()
put()
put()
putChar()
putChar()
putDouble()
putDouble()
putFloat()
putFloat()
putInt()
putInt()
putLong()
putLong()
putShort()
putShort()
slice()
toString()
wrap()
wrap()


ByteBuffer
autoExpand: boolean
pooled: boolean
acquire()
allocate()
allocate()
array()
arrayOffset()
asCharBuffer()
asDoubleBuffer()
asFloatBuffer()
asInputStream()
asIntBuffer()
asLongBuffer()
asOutputStream()
asReadOnlyBuffer()
asShortBuffer()
buf()
capacity()
capacity()
clear()
compact()
compareTo()
duplicate()
equals()
expand()
expand()
fill()
fill()
fillAndReset()
fillAndReset()
flip()
get()
get()
get()
get()
getChar()
getChar()
getDouble()
getDouble()
getFloat()
getFloat()
getHexDump()
getInt()
getInt()
getLong()
getLong()
getObject()
getObject()
getPrefixedString()
getPrefixedString()
getShort()
getShort()
getString()
getString()
getUnsigned()
getUnsigned()
getUnsignedInt()
getUnsignedInt()
getUnsignedShort()
getUnsignedShort()
hasRemaining()
hashCode()
isDirect()
isReadOnly()
limit()
limit()
mark()
markValue()
order()
order()
position()
position()
prefixedDataAvailable()
prefixedDataAvailable()
put()
put()
put()
put()
put()
put()
putChar()
putChar()
putDouble()
putDouble()
putFloat()
putFloat()
putInt()
putInt()
putLong()
putLong()
putObject()
putPrefixedString()
putPrefixedString()
putPrefixedString()
putPrefixedString()
putShort()
putShort()
putString()
putString()
release()
remaining()
reset()
rewind()
skip()
slice()
sweep()
sweep()
toString()
wrap()
wrap()
wrap()


ByteBuffer
autoExpand: boolean
pooled: boolean
acquire()
allocate()
allocate()
array()
arrayOffset()
asCharBuffer()
asDoubleBuffer()
asFloatBuffer()
asInputStream()
asIntBuffer()
asLongBuffer()
asOutputStream()
asReadOnlyBuffer()
asShortBuffer()
buf()
capacity()
capacity()
clear()
compact()
compareTo()
duplicate()
equals()
expand()
expand()
fill()
fill()
fillAndReset()
fillAndReset()
flip()
get()
get()
get()
get()
getChar()
getChar()
getDouble()
getDouble()
getFloat()
getFloat()
getHexDump()
getInt()
getInt()
getLong()
getLong()
getObject()
getObject()
getPrefixedString()
getPrefixedString()
getShort()
getShort()
getString()
getString()
getUnsigned()
getUnsigned()
getUnsignedInt()
getUnsignedInt()
getUnsignedShort()
getUnsignedShort()
hasRemaining()
hashCode()
isDirect()
isReadOnly()
limit()
limit()
mark()
markValue()
order()
order()
position()
position()
prefixedDataAvailable()
prefixedDataAvailable()
put()
put()
put()
put()
put()
putChar()
putChar()
putDouble()
putDouble()
putFloat()
putFloat()
putInt()
putInt()
putLong()
putLong()
putObject()
putPrefixedString()
putPrefixedString()
putPrefixedString()
putPrefixedString()
putShort()
putShort()
putString()
putString()
release()
remaining()
reset()
rewind()
skip()
slice()
sweep()
sweep()
toString()
wrap()
wrap()
wrap()
BaseByteBuffer
support
autoExpand: boolean
mark: int
asCharBuffer()
asDoubleBuffer()
asFloatBuffer()
asIntBuffer()
asLongBuffer()
asShortBuffer()
capacity()
capacity()
clear()
compact()
expand()
flip()
get()
get()
get()
getChar()
getChar()
getDouble()
getDouble()
getFloat()
getFloat()
getInt()
getInt()
getLong()
getLong()
getShort()
getShort()
isDirect()
isReadOnly()
limit()
limit()
mark()
markValue()
order()
order()
position()
position()
put()
put()
put()
put()
putChar()
putChar()
putDouble()
putDouble()
putFloat()
putFloat()
putInt()
putInt()
putLong()
putLong()
putShort()
putShort()
reset()
rewind()
ByteBufferProxy
acquire()
array()
arrayOffset()
asCharBuffer()
asDoubleBuffer()
asFloatBuffer()
asInputStream()
asIntBuffer()
asLongBuffer()
asOutputStream()
asReadOnlyBuffer()
asShortBuffer()
buf()
capacity()
capacity()
clear()
compact()
compareTo()
duplicate()
equals()
expand()
expand()
fill()
fill()
fillAndReset()
fillAndReset()
flip()
get()
get()
get()
get()
getChar()
getChar()
getDouble()
getDouble()
getFloat()
getFloat()
getHexDump()
getInt()
getInt()
getLong()
getLong()
getObject()
getObject()
getPrefixedString()
getPrefixedString()
getShort()
getShort()
getString()
getString()
getUnsigned()
getUnsigned()
getUnsignedInt()
getUnsignedInt()
getUnsignedShort()
getUnsignedShort()
hasRemaining()
hashCode()
isAutoExpand()
isDirect()
isPooled()
isReadOnly()
limit()
limit()
mark()
markValue()
order()
order()
position()
position()
put()
put()
put()
put()
put()
put()
putChar()
putChar()
putDouble()
putDouble()
putFloat()
putFloat()
putInt()
putInt()
putLong()
putLong()
putObject()
putPrefixedString()
putPrefixedString()
putPrefixedString()
putPrefixedString()
putShort()
putShort()
putString()
putString()
release()
remaining()
reset()
rewind()
setAutoExpand()
setPooled()
skip()
slice()
sweep()
sweep()
toString()