Mina2.0快速入门与源码剖析

姓名：邓以克

网名：phinecos(洞庭散人)

MSN: phinecos@msn.com

出处：http://phinecos.cnblogs.com/

目录

1. Mina2.0 快速入门

MinaTimeServer.java

```
package com.vista;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.charset.Charset;

import org.apache.mina.core.service.IoAcceptor;
import org.apache.mina.core.session.IdleStatus;
import org.apache.mina.filter.codec.ProtocolCodecFilter;
import org.apache.mina.filter.codec.textline.TextLineCodecFactory;
import org.apache.mina.filter.logging.LoggingFilter;
import org.apache.mina.transport.socket.nio.NioSocketAcceptor;


public class MinaTimeServer
{
    private static final int PORT = 6488;
    public static void main(String[] args) throws IOException
    {
        //监听即将到来的 TCP 连接
        IoAcceptor acceptor = new NioSocketAcceptor();
        acceptor.getFilterChain().addLast("logger", new LoggingFilter());
        acceptor.getFilterChain().addLast("codec", new ProtocolCodecFilter
( new TextLineCodecFactory( Charset.forName("UTF-8"))));

        acceptor.setHandler(new TimeServerHandler());

        acceptor.getSessionConfig().setReadBufferSize(2048);
        acceptor.getSessionConfig().setIdleTime(IdleStatus.BOTH_IDLE, 10);

        acceptor.bind(new InetSocketAddress(PORT));
        System.out.println("服务器启动");
    }

}
```

TimeServerHandler.java

```
package com.vista;

import java.util.Date;

import org.apache.mina.core.service.IoHandlerAdapter;
import org.apache.mina.core.session.IdleStatus;
import org.apache.mina.core.session.IoSession;

public class TimeServerHandler extends IoHandlerAdapter
{

   public void exceptionCaught(IoSession session, Throwable cause) throws
 Exception
   {
      cause.printStackTrace();
   }
   public void messageReceived(IoSession session, Object message) throws
Exception
   {
      String strMsg = message.toString();
      if(strMsg.trim().equalsIgnoreCase("quit"))
      {
         session.close();
         return;
      }
      Date date = new Date();
      session.write(date.toString());

      System.out.println("Message written…");
   }
   public void sessionIdle(IoSession session, IdleStatus status) throws E
xception
   {

      System.out.println("IDLE…"+session.getIdleCount(status));
   }

}
```

测试:

Client Output **Server Output**

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

参考资料：1，MINA v2.0 Quick Start Guide

2. Mina2.0 框架源码剖析（一）

整个框架最核心的几个包是：org.apache.mina.core.service, org.apache.mina.core.session, org.apache.mina.core.polling 以及 org.apache.mina.transport.socket。

这一篇先来看 org.apache.mina.core.service。第一个要说的接口是 IoService,它是所有 IoAcceptor 和 IoConnector 的基接口.对于一个 IoService,有哪些信息需要我们关注呢？1）底层的元数据信息 TransportMetadata，比如底层的网络服务提供者（NIO,ARP,RXTX 等），2）通过这个服务创建一个新会话时，新会话的默认配置 IoSessionConfig。3）此服务所管理的所有会话。4）与这个服务相关所产生的事件所对应的监听者（IoServiceListener）。5）处理这个服务所管理的所有连接的处理器(IoHandler)。6）每个会话都有一个过滤器链（IoFilterChain），每个过滤器链通过其对应的 IoFilterChainBuilder 来负责构建。7）由于此服务管理了一系列会话,因此可以通过广播的方式向所有会话发送消息,返回结果是一个 WriteFuture 集，后者是一种表示未来预期结果的数据结构。8）服务创建的会话（IoSession）相关的数据通过 IoSessionDataStructureFactory 来提供。9）发送消息时有一个写缓冲队列。10）服务的闲置状态有三种：读端空闲，写端空闲，双端空闲。11）还提供服务的一些统计信息，比如时间，数据量等。

IoService 这个服务是对于服务器端的接受连接和客户端发起连接这两种行为的抽象。

再来从服务器看起，IoAcceptor 是 IoService 的子接口，它用于绑定到指定的 ip 和端口，从而接收来自客户端的连接请求，同时会 fire 相应的客户端连接成功接收/取消/失败等事件给自己的 IoHandle 去处理。当服务器端的 Accpetor 从早先绑定的 ip 和端口上取消绑定时，默认是所有的客户端会话会被关闭，这种情况一般出现在服务器挂掉了，则客户端收到连接关闭的提示。这个接口最重要的两个方法是 bind()和 unbind()，当这两个方法被调用时，服务端的连接接受线程就启动或关闭了。

再来看一看客户端的连接发起者接口 IoConnector，它的功能和 IoAcceptor 基本对应的，它用于尝试连接到服务器指定的 ip 和端口，同时会 fire 相应的客户端连接事件给自己的 IoHandle 去处理。当 connet 方法被调用后用于连接服务器端的线程就启动了,而当所有的连接尝试都结束时线程就停止。尝试连接的超时时间可以自行设置。Connect 方法返回的结果是 ConnectFuture，这和前面说的 WriteFuture 类似，在后面会有一篇专门讲这个模式的应用。

前面的 IoAcceptor 和 IoConnector 就好比是两个负责握手的仆人，而真正代表会话的实际 I/O 操作的接口是 IoProcessor，它对现有的 Reactor 模式架构的 Java NIO 框架继续做了一层封装。它的泛型参数指明了它能处理的会话类型。接口中最重要的几个方法， add 用于将指定会话加入到此 Processor 中， 让它负责处理与此会话相关的所有 I/O 操作。由于写操作会有一个写请求队列，flush 就用于对指定会话的写请求队列进行强制刷数据。remove 方法用于从此 Processor 中移除和关闭指定会话,这样就可以关闭会话相关联的连接并释放所有相关资源。updateTrafficMask 方法用于控制会话的 I/O 行为，比如是否允许读/写。

然后来说说 IoHandle 接口，Mina 中的所有 I/O 事件都是通过这个接口来处理的，这些事件都是上面所说的 I/O Processor 发出来的，要注意的一点是**同一个 I/O Processor 线程是负责处理多个会话的**。包括下面这几个事件的处理：

```
public interface IoHandler
{
    void sessionCreated(IoSession session) throws Exception;//会话创建
    void sessionOpened(IoSession session) throws Exception;//打开会话,与 ses
sionCreated 最大的区别是它是从另一个线程处调用的
    void sessionClosed(IoSession session) throws Exception;//会话结束，当连接
关闭时被调用
    void sessionIdle(IoSession session, IdleStatus status) throws Exceptio
n;//会话空闲
    void exceptionCaught(IoSession session, Throwable cause) throws Except
ion;//异常捕获，Mina 会自动关闭此连接
    void messageReceived(IoSession session, Object message) throws Excepti
on;//接收到消息
    void messageSent(IoSession session, Object message) throws Exception;/
/发送消息
}
```

IoHandlerAdapter 就不说了，简单地对 IoHandler 使用适配器模式封装了下，让具体的 IoHandler 子类从其继承后，从而可以对自身需要哪些事件处理拥有自主权。

来看看 IoServiceListener 接口，它用于监听 IoService 相关的事件。

```
public interface IoServiceListener extends EventListener
{
    void serviceActivated(IoService service) throws Exception;//激活了一个新
service
    void serviceIdle(IoService service, IdleStatus idleStatus) throws Exce
ption; // service 闲置
    void serviceDeactivated(IoService service) throws Exception;//挂起一个 s
ervice
    void sessionCreated(IoSession session) throws Exception;//创建一个新会话
    void sessionDestroyed(IoSession session) throws Exception;//摧毁一个新会
话
}
```

IoServiceListenerSupport 类就是负责将上面的 IoService 和其对应的各个 IoServiceListener 包装到一起进行管理。下面是它的成员变量：

```
    private final IoService service;
    private final List<IoServiceListener> listeners = new CopyOnWriteArray
List<IoServiceListener>();
    private final ConcurrentMap<Long, IoSession> managedSessions = new Con
currentHashMap<Long, IoSession>();//被管理的会话集（其实就是服务所管理的会话集）
```

```
   private final Map<Long, IoSession> readOnlyManagedSessions = Collectio
ns.unmodifiableMap(managedSessions);//上面的会话集的只读版
   private final AtomicBoolean activated = new AtomicBoolean();//被管理的服
务是否处于激活状态
```

激活事件就以会话创建为例来说明：

```
public void fireSessionCreated(IoSession session)
{
       boolean firstSession = false;
       if (session.getService() instanceof IoConnector)
{//若服务类型是 Connector，则说明是客户端的连接服务
           synchronized (managedSessions)
{//锁住当前已经建立的会话集
               firstSession = managedSessions.isEmpty();//看服务所管理的会话集
是否为空集
           }
       }
             if (managedSessions.putIfAbsent(Long.valueOf(session.getId
()), session) != null) { // If already registered, ignore.
           return;
       }
       if (firstSession)
{//第一个连接会话，fire 一个虚拟的服务激活事件
           fireServiceActivated();
       }
       //呼叫过滤器的事件处理
       session.getFilterChain().fireSessionCreated();// 会话创建
       session.getFilterChain().fireSessionOpened();//会话打开
       int managedSessionCount = managedSessions.size();
       //统计管理的会话数目
       if (managedSessionCount > largestManagedSessionCount)
{
           largestManagedSessionCount = managedSessionCount;
       }
       cumulativeManagedSessionCount ++;
       //呼叫监听者的事件处理函数
       for (IoServiceListener l : listeners)
{
           try
 {
               l.sessionCreated(session);
           } catch (Throwable e)
{
               ExceptionMonitor.getInstance().exceptionCaught(e);
```

```
            }
        }
    }
```

这里值得注意的一个地方是断开连接会话，设置了一个监听锁，直到所有连接会话被关闭后才放开这个锁。

```
  private void disconnectSessions()
    {
        if (!(service instanceof IoAcceptor))
        {//确保服务类型是 IoAcceptor
            return;
        }
        if (!((IoAcceptor) service).isCloseOnDeactivation())
        {// IoAcceptor 是否设置为在服务失效时关闭所有连接会话
            return;
        }
        Object lock = new Object();//监听锁
        IoFutureListener<IoFuture> listener = new LockNotifyingListener(lo
ck);
        for (IoSession s : managedSessions.values())
        {
            s.close().addListener(listener);//为每个会话的close动作增加一个监听
者
        }
        try
        {
            synchronized (lock)
            {
                while (!managedSessions.isEmpty())
                {//所管理的会话还没有全部结束，持锁等待
                    lock.wait(500);
                }
            }
        } catch (InterruptedException ie)
        {
            // Ignored
        }
    }
    private static class LockNotifyingListener implements IoFutureListener
<IoFuture>
    {
        private final Object lock;
        public LockNotifyingListener(Object lock)
        {
```

```
            this.lock = lock;
        }
        public void operationComplete(IoFuture future)
{

            synchronized (lock)
            {
                lock.notifyAll();
            }
        }
    }
```

3. Mina2.0 框架源码剖析（二）

上一篇介绍了几个核心的接口，这一篇主要介绍实现这些接口的抽象基类。首先是实现 IoService 接口的 AbstractIoService 类。它包含了一个 Executor 来处理到来的事件。每个 AbstractIoService 都一个 AtomicInteger 类型的 id 号，确保每个 id 的唯一性。

它内部的 Executor 可以选择是从外部传递进构造函数中，也可以在实例内部自行构造，若是后者，则它将是 ThreadPoolExecutor 类的一个实例，即是 Executor 线程池中的一员。代码如下：

```
    if (executor == null)
       {
          this.executor = Executors.newCachedThreadPool();
          createdExecutor = true;
       }
       else
       {
          this.executor = executor;
          createdExecutor = false;
       }
```

其中有一个 IdleStatusChecker 成员，它用来对服务的空闲状态进行检查，在一个服务激活时会将服务纳入到检查名单中，而在服务失效时会将服务从名单中剔除。会单独开一个线程进行具体的空闲检查，这是通过下面这个线程类来负责的：

```
private class NotifyingTaskImpl implements NotifyingTask
{
       private volatile boolean cancelled;//取消检查标志
       private volatile Thread thread;
       public void run()
{
          thread = Thread.currentThread();
          try {
             while (!cancelled)
{
                //每隔1秒检查一次空闲状态
                long currentTime = System.currentTimeMillis();
                notifyServices(currentTime);
                notifySessions(currentTime);
                try
```

```
{
                    Thread.sleep(1000);
                } catch (InterruptedException e)
{
                    // will exit the loop if interrupted from interrupt()
                }
            }
        }
 Finally
 {
            thread = null;
        }
    }
}
```

具体的空闲检查代码如下，超过能容忍的最大空闲时间，就会 fire 出 SessionIdle 事件，上文也说过空闲有三种类型：读端空，写端空，双端空。

```
notifyIdleSession1(s, currentTime,
            s.getConfig().getIdleTimeInMillis(IdleStatus.BOTH_IDLE),Idl
eStatus.BOTH_IDLE,Math.max(s.getLastIoTime(),s.getLastIdleTime(IdleStatu
s.BOTH_IDLE)));
   private static void notifyIdleSession1(
        AbstractIoSession session, long currentTime,
        long idleTime, IdleStatus status, long lastIoTime)
{
    if (idleTime > 0 && lastIoTime != 0
            && currentTime - lastIoTime >= idleTime)
{
        session.getFilterChain().fireSessionIdle(status);
    }
}
```

在释放资源的方法时，首先去获取释放锁 disposalLock 才行，然后具体的释放动作是通过 dispose0完成的，接着取消掉空闲检查线程，此外，若线程是内部创建的线程池中的一员，则通过线程池去关闭线程。

```
  public final void dispose()
{
    IoFuture disposalFuture;
    synchronized (disposalLock)
{//获取释放锁
        disposalFuture = this.disposalFuture;
        if (!disposing) {
            disposing = true;
```

```
                try {
                    this.disposalFuture = disposalFuture = dispose0();//具体
释放动作
                } catch (Exception e) {
                    ExceptionMonitor.getInstance().exceptionCaught(e);
                } finally {
                    if (disposalFuture == null) {
                        disposed = true;
                    }
                }
            }
        }
        idleStatusChecker.getNotifyingTask().cancel();
        if (disposalFuture != null)
{//无中断地等待释放动作完成
            disposalFuture.awaitUninterruptibly();
        }
        if (createdExecutor)
{通过线程池去关闭线程
            ExecutorService e = (ExecutorService) executor;
            e.shutdown();
            while (!e.isTerminated()) {
                try {
                    e.awaitTermination(Integer.MAX_VALUE, TimeUnit.SECONDS);
                } catch (InterruptedException e1) {
                    // Ignore; it should end shortly.
                }
            }
        }
        disposed = true;
    }
```

再来看会话初始化完成后的动作每个 session 都保持有自己的属性映射图，在会话结束初始化时，应该设置这个 AttributeMap。

```
((AbstractIoSession) session).setAttributeMap(session.getService()
                .getSessionDataStructureFactory().getAttributeMap(sessio
n));
```

除此以为，还应该为会话配置写请求队列：

```
((AbstractIoSession) session).setWriteRequestQueue(session
                .getService().getSessionDataStructureFactory()
                .getWriteRequestQueue(session));
```

在初始化时会在会话的属性中加入一项 SESSION_CREATED_FUTURE，这个属性会在连接真正建立后从会话中去除。

```
       if (future != null && future instanceof ConnectFuture)
{
         session.setAttribute(DefaultIoFilterChain.SESSION_CREATED_FUTUR
E,
                 future);
 }
```

用户特定的初始化动作在 finishSessionInitialization0这个方法中自行实现。

4. Mina2.0 框架源码剖析（三）

AbstractIoAcceptor 类继承自 AbstractIoService 基类，并实现了 IoAcceptor 接口，它主要的成员变量是本地绑定地址。

```
    private final List<SocketAddress> defaultLocalAddresses =
        new ArrayList<SocketAddress>();
    private final List<SocketAddress> unmodifiableDefaultLocalAddresses =
        Collections.unmodifiableList(defaultLocalAddresses);
    private final Set<SocketAddress> boundAddresses =
        new HashSet<SocketAddress>();
```

在调用 bind 或 unbind 方法时需要先获取绑定锁 bindLock，具体的绑定操作还是在 bind0这个方法中实现的。一旦绑定成功后，就会向服务监听者发出服务激活的事件(ServiceActivated),同理，解除绑定也是在 unbind0这个方法中具体实现的。一旦解除绑定成功后，就会向服务监听者发出服务激活的事件(ServiceDeActivated)。

AbstractIoConnector 类继承自 AbstractIoService 基类，并实现了 IoConnect 接口,连接超时检查间隔时间默认是50毫秒，超时时间默认为1分钟，用户可以自行配置。此类中重要的方法就是 connect 方法，其中调用了具体的连接逻辑实现 connect0,

```
  protected abstract ConnectFuture connect0(SocketAddress remoteAddress,
          SocketAddress localAddress, IoSessionInitializer<? extends Conn
ectFuture> sessionInitializer);
```

AbstractIoConnector 在 AbstractIoService 的基础上，在会话初始化结束时增加了一个功能，就是加入了一个监听者，当连接请求被取消时立即结束此会话。

```
    protected final void finishSessionInitialization0(
            final IoSession session, IoFuture future) {
        // In case that ConnectFuture.cancel() is invoked before
        // setSession() is invoked, add a listener that closes the
        // connection immediately on cancellation.
        future.addListener(new IoFutureListener<ConnectFuture>() {
            public void operationComplete(ConnectFuture future) {
                if (future.isCanceled()) {
                    session.close();
                }
            }
        });
}
```

下面再来看一个 IoProcessor 接口的基本实现类 SimpleIoProcessorPool，它的泛型参数是 AbstractIoSession 的子类，表示此 Processor 管理的具体会话类型。并且这个类还实现了池化，它会将多个 IoSession 分布到多个 IoProcessor 上去管理。下面是文档中给出的一个示例：

```
// Create a shared pool.
SimpleIoProcessorPool<NioSession> pool =
        new SimpleIoProcessorPool<NioSession>(NioProcessor.class, 16);

// Create two services that share the same pool.
SocketAcceptor acceptor = new NioSocketAcceptor(pool);
SocketConnector connector = new NioSocketConnector(pool);
...
// Release related resources.
connector.dispose();
acceptor.dispose();
pool.dispose();
```

与 Processor 池有关的包括如下这些成员变量：

```
 private static final int DEFAULT_SIZE = Runtime.getRuntime().availablePr
ocessors() + 1;//处理池大小，默认是处理器数+1，便于多核分布处理
   private final IoProcessor<T>[] pool;//IoProcessor 池
   private final AtomicInteger processorDistributor = new AtomicInteger();
```

Processor 池的构造过程,其中有三种构造函数供选择来构造一个 Processor ：

1. 带参数 ExecutorService 的构造函数.
2. 带参数为 Executor 的构造函数.
3. 默认构造函数

```
       pool = new IoProcessor[size];//构建池

       boolean success = false;
       try {
           for (int i = 0; i < pool.length; i ++) {
              IoProcessor<T> processor = null;

//有三种构造函数供选择来构造一个 Processor
 try {
                 try {
                    processor = processorType.getConstructor(ExecutorServ
ice.class).newInstance(executor);
                 } catch (NoSuchMethodException e) {
```

```
                    // Tc the next step…
                }

                if (processor == null) {
                    try {
                        processor = processorType.getConstructor(Executor.
class).newInstance(executor);
                    } catch (NoSuchMethodException e) {

                        // To the next step…
                    }
                }

                if (processor == null) {
                    try {
                        processor = processorType.getConstructor().newIns
tance();
                    } catch (NoSuchMethodException e) {

                        // To the next step…
                    }
                }
            } catch (RuntimeException e) {
                throw e;
            } catch (Exception e) {
                throw new RuntimeIoException(
                        "Failed to create a new instance of " + processorTy
pe.getName(), e);
            }
            pool[i] = processor;
        }

        success = true;
    } finally {
        if (!success) {
            dispose();
        }
    }
```

从 Processor 池中分配一个 processor 的过程，注意**一个 processor 是可以同时管理多个 session 的**。

```
private IoProcessor<T> getProcessor(T session)
{//返回 session 所在的 processor，若没分配，则为之分配一个
```

```
        IoProcessor<T> p = (IoProcessor<T>) session.getAttribute(PROCESSO
R);//看 session 的属性中是否保存对应的 Processor
        if (p == null)
{//还没为此 session 分配 processor
            p = nextProcessor();//从池中取一个 processor
            IoProcessor<T> oldp =
                (IoProcessor<T>) session.setAttributeIfAbsent(PROCESSOR, p);

            if (oldp != null)
{//原来的 processor
                p = oldp;
            }
        }
        return p;
    }

    private IoProcessor<T> nextProcessor()
    {//从池中分配一个 Processor
        checkDisposal();
        return pool[Math.abs(processorDistributor.getAndIncrement()) % poo
l.length];
    }
```

5. Mina2.0 框架源码剖析（四）

前面几篇介绍完了 org.apache.mina.core.service 这个包，现在进入 org.apache.mina.core.session，这个包主要是围绕 IoSession 展开的，包括会话的方方面面。

IoSession 接口与底层的传输层类型无关（也就是不管是 TCP 还是 UDP）,它表示通信双端的连接。它提供用户自定义属性，可以用于在过滤器和处理器之间交换用户自定义协议相关的信息。

每个会话都有一个 Service 为之提供服务，同时有一个 Handler 负责此会话的 I/O 事件处理。最重要的两个方法是 read 和 write，这两个方法都是异步执行，若要真正完成必须在其返回结果上进行等待。关闭会话的方法 close 是异步执行的，也就是应当等待返回的 CloseFuture，此外，还有另一种关闭方式 closeOnFlush,它和 close 的区别是会先 flush 掉写请求队列中的请求数据，再关闭会话，但同样是异步的。会话的读写类型是可配置的，在运行中可设置此端是否可读写。

一个会话主要包含两个方面的数据，属性映射图，写请求队列，在这里作者使用了工厂模式来为新创建的会话提供这些数据结构。

```
public interface IoSessionDataStructureFactory
{
    IoSessionAttributeMap getAttributeMap(IoSession session) throws Except
ion;
    WriteRequestQueue getWriteRequestQueue(IoSession session) throws Excep
tion;
}
```

IoSessionConfig 接口用于表示会话的配置信息，主要包括：读缓冲区大小，会话数据吞吐量，计算吞吐量时间间隔，指定会话端的空闲时间，写请求操作超时时间。在这个接口中有一个方法值得注意

```
void setUseReadOperation(boolean useReadOperation);
```

通过它来设置 IoSession 的 read 方法是否启用，若启用的话，则所有接收到的消息都会存储在内部的一个阻塞队列中，好处在于可以更方便用户对信息的处理，但对于某些应用来说并不管用，而且还会造成内存泄露，因此默认情况下这个选项是不开启的。

IoSessionInitializer 接口定义了一个回调函数，这在 AbstractIoService 这个类中的 finishSessionInitialization 方法中已经见识过它的使用了，用于把用户自定义的会话初始化行为剥离出来。

```
public interface IoSessionInitializer<T extends IoFuture>
{
    void initializeSession(IoSession session, T future);
```

```
}
```

IoSessionRecycler 接口为一个无连接的传输服务提供回收现有会话的服务，主要的方法是：

```
IoSession recycle(SocketAddress localAddress, SocketAddress remoteAddres
s);
```

一个会话的读写能力控制通过 TrafficMask 类来描述，主要是 SelectionKey.OP_READ 和 SelectionKey.OP_WRITE 结合。此类使用单例模式实现，还提供了与，或，非，异或等位操作来动态控制会话读写能力。

Mina 中的 I/O 事件类型如下：

```
public enum IoEventType {
    SESSION_CREATED,//会话创建
    SESSION_OPENED,//会话打开
    SESSION_CLOSED,//会话关闭
    MESSAGE_RECEIVED,//接收到消息
    MESSAGE_SENT,//发送消息
    SESSION_IDLE,//空闲
    EXCEPTION_CAUGHT,//异常捕获
    WRITE,
    CLOSE,
    SET_TRAFFIC_MASK,//设置读写能力
}
```

IoEvent 类实现了 Runnable 接口，表示一个 I/O 事件或一个 I/O 请求，包括事件类型，所属的会话，事件参数值。最重要的方法就是 fire，根据事件类型向会话的过滤器链上的众多监听者发出事件到来的信号。

```
    public void fire() {
        switch (getType()) {
        case MESSAGE_RECEIVED:
            getSession().getFilterChain().fireMessageReceived(getParameter
());
            break;
        case MESSAGE_SENT:
            getSession().getFilterChain().fireMessageSent((WriteRequest) ge
tParameter());
            break;
        case WRITE:
            getSession().getFilterChain().fireFilterWrite((WriteRequest) ge
tParameter());
            break;
        case SET_TRAFFIC_MASK:
            getSession().getFilterChain().fireFilterSetTrafficMask((Traffi
cMask) getParameter());
```

```
            break;
        case CLOSE:
            getSession().getFilterChain().fireFilterClose();
            break;
        case EXCEPTION_CAUGHT:
            getSession().getFilterChain().fireExceptionCaught((Throwable) g
etParameter());
            break;
        case SESSION_IDLE:
            getSession().getFilterChain().fireSessionIdle((IdleStatus) getP
arameter());
            break;
        case SESSION_OPENED:
            getSession().getFilterChain().fireSessionOpened();
            break;
        case SESSION_CREATED:
            getSession().getFilterChain().fireSessionCreated();
            break;
        case SESSION_CLOSED:
            getSession().getFilterChain().fireSessionClosed();
            break;
        default:
            throw new IllegalArgumentException("Unknown event type: " + getT
ype());
        }
    }
```

Mina 的会话中，有三种类型的闲置状态：1）READER_IDLE ，这表示从远端没有数据到来，读端空闲。2）WRITER_IDLE ，这表示写端没有在写数据。3）BOTH_IDLE，读端和写端都空闲。为了节约会话资源，可以让用户设置当空闲超过一定时间后关闭此会话，因为此会话可能在某一端出问题了，从而导致另一端空闲超过太长时间。这可以通过使用 IoSessionConfig.setIdleTime(IdleStatus,int)来完成，空闲时间阀值在会话配（IoSessionConfig）中设置。

前面介绍过 IoSessionDataStructureFactor 接口为会话提供所需要的数据结构，DefaultIoSessionDataStructureFactory 是其一个默认实现类。它提供的写请求队列内部是一个初始大小为16的循环队列，并且在插入队列尾部和从队列头部取数据时都必须满足互斥同步。

```
 private static class DefaultWriteRequestQueue implements WriteRequestQue
ue {
        private final Queue<WriteRequest> q = new CircularQueue<WriteReques
t>(16);
        public void dispose(IoSession session) {
        }
        public void clear(IoSession session) {
```

```
            q.clear();
        }
        public synchronized boolean isEmpty(IoSession session) {
            return q.isEmpty();
        }
        public synchronized void offer(IoSession session, WriteRequest writ
eRequest) {
            q.offer(writeRequest);
        }
        public synchronized WriteRequest poll(IoSession session) {
            return q.poll();
        }
    }
```

6. Mina2.0 框架源码剖析（五）

前面介绍过 IoSessionRecycler 是负责回收不再使用的会话的接口，ExpiringSessionRecycler 是其一个实现类，用于回收超时失效的会话。

```
private ExpiringMap<Object, IoSession> sessionMap;//待处理的会话集
private ExpiringMap<Object, IoSession>.Expirer mapExpirer;//负责具体的回收工作
```

sessionMap 的键是由本地地址和远端地址共同组成的，值是这两个地址对应的会话。

Expirer 类实现了 Runnable 接口，这个线程负责监控 ExpiringMap，并把 ExpiringMap 中超过阀值的元素从 ExpiringMap 中移除。这个线程调用了 setDaemon(true)，因此是作为守护线程在后台运行。具体的处理过程如下：

```
  private void processExpires()
      {
          long timeNow = System.currentTimeMillis();//当前时间
          for (ExpiringObject o : delegate.values())
          {
              if (timeToLiveMillis <= 0)
              {
                  continue;
              }
              long timeIdle = timeNow - o.getLastAccessTime();//时间差
              if (timeIdle >= timeToLiveMillis)
              {//超时
                  delegate.remove(o.getKey());
                  for (ExpirationListener<V> listener : expirationListeners)
                  {//呼叫监听者
                      listener.expired(o.getValue());
                  }
              }
          }
      }
```

启动/关闭超时检查线程都需要进行封锁机制,这里使用的是读写锁:

```
    private final ReadWriteLock stateLock = new ReentrantReadWriteLock();
```

```
    public void startExpiring()
    {
        stateLock.writeLock().lock();
        try
        {
            if (!running)
            {
                running = true;
                expirerThread.start();
            }
        }
        finally
        {
            stateLock.writeLock().unlock();
        }
    }
    public void stopExpiring()
    {
        stateLock.writeLock().lock();
        try
        {
            if (running)
            {
                running = false;
                expirerThread.interrupt();
            }
        }
        finally
        {
            stateLock.writeLock().unlock();
        }
    }
```

会话超时监听者：

```
private class DefaultExpirationListener implements
        ExpirationListener<IoSession> {
    public void expired(IoSession expiredSession) {
        expiredSession.close();//关闭超时的会话
    }
}
```

7. Mina2.0 框架源码剖析（六）

上文的内容还有一些没有结尾，这篇补上。在 ExpiringMap 类中，使用了一个私有内部类 ExpiringObject 来表示待检查超时的对象，它包括三个域，键，值，上次访问时间，以及用于上次访问时间这个域的读写锁：

```
        private K key;
        private V value;
        private long lastAccessTime;
        private final ReadWriteLock lastAccessTimeLock = new ReentrantReadW
riteLock();
```

而 ExpiringMap 中包括了下述几个变量：

```
  private final ConcurrentHashMap<K, ExpiringObject> delegate;//超时代理集
合，保存待检查对象
    private final CopyOnWriteArrayList<ExpirationListener<V>> expirationLi
steners;//超时监听者
    private final Expirer expirer;//超时检查线程
```

现在再来看看 IoSession 的一个抽象实现类 AbstractIoSession。这是它的几个重要的成员变量：

```
    private IoSessionAttributeMap attributes;//会话属性映射图
    private WriteRequestQueue writeRequestQueue;//写请求队列
    private WriteRequest currentWriteRequest;//当前写请求
```

当要结束当前会话时，会发送一个一个写请求 CLOSE_REQUEST。而 closeFuture 这个 CloseFuture 会在连接关闭时状态被设置为”closed”,它的监听器是 SCHEDULED_COUNTER_RESETTER。

close 和 closeOnFlush 都是异步的关闭操作，区别是前者立即关闭连接，而后者是在写请求队列中放入一个 CLOSE_REQUEST，并将其即时刷新出去，若要真正等待关闭完成，需要调用方在返回的 CloseFuture 等待

```
public final CloseFuture close() {
        synchronized (lock) {
            if (isClosing()) {
                return closeFuture;
            } else {
                closing = true;
            }
        }
```

```
        getFilterChain().fireFilterClose();//fire出关闭事件
        return closeFuture;
    }

    public final CloseFuture closeOnFlush() {
        getWriteRequestQueue().offer(this, CLOSE_REQUEST);
        getProcessor().flush(this);
        return closeFuture;
    }
```

下面来看看读数据的过程：

```
public final CloseFuture close() {
        synchronized (lock) {
            if (isClosing()) {
                return closeFuture;
            } else {
                closing = true;
            }
        }
        getFilterChain().fireFilterClose();//fire出关闭事件
        return closeFuture;
    }

    public final CloseFuture closeOnFlush() {
        getWriteRequestQueue().offer(this, CLOSE_REQUEST);
        getProcessor().flush(this);
        return closeFuture;
    }


    private Queue<ReadFuture> getReadyReadFutures() {//返回可被读数据队列
        Queue<ReadFuture> readyReadFutures =
            (Queue<ReadFuture>) getAttribute(READY_READ_FUTURES_KEY);//从会
话映射表中取出可被读数据队列
        if (readyReadFutures == null) {//第一次读数据
            readyReadFutures = new CircularQueue<ReadFuture>();//构造一个新读
数据队列
            Queue<ReadFuture> oldReadyReadFutures =
                (Queue<ReadFuture>) setAttributeIfAbsent(
                        READY_READ_FUTURES_KEY, readyReadFutures);
            if (oldReadyReadFutures != null) {
                readyReadFutures = oldReadyReadFutures;
            }
        }
```

```
        return readyReadFutures;
    }

    public final ReadFuture read() {//读数据
        if (!getConfig().isUseReadOperation()) {//会话配置不允许读数据(这是默认情况)
            throw new IllegalStateException("useReadOperation is not enabled.");
        }
        Queue<ReadFuture> readyReadFutures = getReadyReadFutures();//获取已经可被读数据队列
        ReadFuture future;
        synchronized (readyReadFutures) {//锁住读数据队列
            future = readyReadFutures.poll();//取队头数据
            if (future != null) {
                if (future.isClosed()) {//关联的会话已经关闭了，让读者知道此情况
                    readyReadFutures.offer(future);
                }
            } else {
                future = new DefaultReadFuture(this);
                getWaitingReadFutures().offer(future); //将此数据插入等待被读取数据的队列，这个代码和上面的getReadyReadFutures类似，只是键值不同而已

            }
        }
        return future;
    }
```

再来看写数据到指定远端地址的过程，可以写三种类型数据：IoBuffer，整个文件或文件的部分区域，这会通过传递写请求给过滤器链条来完成数据向目的远端的传输。

```
    public final WriteFuture write(Object message, SocketAddress remoteAddress) {
        FileChannel openedFileChannel = null;
        try
        {
            if (message instanceof IoBuffer&& !((IoBuffer) message).hasRemaining())
            {// 空消息
                throw new IllegalArgumentException(
                "message is empty. Forgot to call flip()?");
            }
            else if (message instanceof FileChannel)
            {//要发送的是文件的某一区域
```

```
                FileChannel fileChannel = (FileChannel) message;
                message = new DefaultFileRegion(fileChannel, 0, fileChannel.
size());
            }
            else if (message instanceof File)
            {//要发送的是文件,打开文件通道
                File file = (File) message;
                openedFileChannel = new FileInputStream(file).getChannel();
                message = new DefaultFileRegion(openedFileChannel, 0, opened
FileChannel.size());
            }
        }
        catch (IOException e)
        {
            ExceptionMonitor.getInstance().exceptionCaught(e);
            return DefaultWriteFuture.newNotWrittenFuture(this, e);
        }
        WriteFuture future = new DefaultWriteFuture(this);
        getFilterChain().fireFilterWrite(
                new DefaultWriteRequest(message, future, remoteAddress)); //
构造写请求,通过过滤器链发送出去，写请求中指明了要发送的消息，目的地址，以及返回的结果

//如果打开了一个文件通道（发送的文件的部分区域或全部），就必须在写请求完成时关闭文件通道
        if (openedFileChannel != null) {
            final FileChannel finalChannel = openedFileChannel;
            future.addListener(new IoFutureListener<WriteFuture>() {
                public void operationComplete(WriteFuture future) {
                    try {
                        finalChannel.close();//关闭文件通道
                    } catch (IOException e) {
                        ExceptionMonitor.getInstance().exceptionCaught(e);
                    }
                }
            });
        }
        return future;//写请求成功完成
    }
```

最后，来看看一个 WriteRequestQueue 的实现，唯一加入的一个功能就是若在队头发现是请求关闭，则会去关闭会话。

```
 private class CloseRequestAwareWriteRequestQueue implements WriteRequest
Queue {
        private final WriteRequestQueue q;//内部实际的写请求队列
        public CloseRequestAwareWriteRequestQueue(WriteRequestQueue q) {
```

```
            this.q = q;
        }
        public synchronized WriteRequest poll(IoSession session) {
            WriteRequest answer = q.poll(session);
            if (answer == CLOSE_REQUEST) {
                AbstractIoSession.this.close();
                dispose(session);
                answer = null;
            }
            return answer;
        }
        public void offer(IoSession session, WriteRequest e) {
            q.offer(session, e);
        }
        public boolean isEmpty(IoSession session) {
            return q.isEmpty(session);
        }
        public void clear(IoSession session) {
            q.clear(session);
        }
        public void dispose(IoSession session) {
            q.dispose(session);
        }
    }
```

8. Mina2.0 框架源码剖析（七）

前面介绍完了 org.apache.mina.core.session 这个包，现在开始进入 org.apache.mina.core. polling 包。这个包里包含了实现基于轮询策略（比如 NIO 的 select 调用或其他类型的 I/O 轮询系统调用（如 epoll,poll,kqueue 等）的基类。

先来看 AbstractPollingIoAcceptor 这个抽象基类，它继承自 AbstractIoAcceptor，两个泛型参数分别是所处理的会话和服务器端 socket 连接。底层的 sockets 会被不断检测，并当有任何一个 socket 需要被处理时就会被唤醒去处理。这个类封装了服务器端 socket 的 bind,accept 和 dispose 等动作，其成员变量 Executor 负责接受来自客户端的连接请求，另一个 AbstractPollingIoProcessor 用于处理客户端的 I/O 操作请求，如读写和关闭连接。

其最重要的几个成员变量是：

```
  private final Queue<AcceptorOperationFuture> registerQueue = new Concurr
entLinkedQueue<AcceptorOperationFuture>();//注册队列
   private final Queue<AcceptorOperationFuture> cancelQueue = new Concurr
entLinkedQueue<AcceptorOperationFuture>();//取消注册队列
   private final Map<SocketAddress, H> boundHandles = Collections
           .synchronizedMap(new HashMap<SocketAddress, H>());//本地地址到服
务器 socket 的映射表
```

先来看看当服务端调用 bind 后的处理过程：

```
protected final Set<SocketAddress> bind0(
           List<? extends SocketAddress> localAddresses) throws Exception
{
       AcceptorOperationFuture request = new AcceptorOperationFuture(loca
lAddresses);//注册请求
       registerQueue.add(request);//加入注册队列中，等待 worker 处理
       //创建一个 Worker 实例，开始工作
       startupWorker();
       wakeup();
       request.awaitUninterruptibly();
       // 更新本地绑定地址
       Set<SocketAddress> newLocalAddresses = new HashSet<SocketAddress>
();
       for (H handle : boundHandles.values()) {
           newLocalAddresses.add(localAddress(handle));
       }
       return newLocalAddresses;
   }
```

真正的负责接收客户端请求的工作都是 Worker 线程完成的,

```
private class Worker implements Runnable {
       public void run() {
          int nHandles = 0;
          while (selectable) {
             try {
                // Detect if we have some keys ready tc be processed
                boolean selected = select();//检测是否有 SelectionKey 已经可以被处理了
                nHandles += registerHandles();//注册服务器 sockets 句柄，这样做的目的是将 Selector 的状态置于 OP_ACCEPT，并绑定到所监听的端口上，表明接受了可以接收的来自客户端的连接请求，
                if (selected) {
                   processHandles(selectedHandles());//处理可以被处理的 SelectionKey 状态为 OP_ACCEPT 的服务器 socket 句柄集(即真正处理来自客户端的连接请求)
                }
                nHandles -= unregisterHandles();//检查是否有取消连接的客户端请求
                if (nHandles == 0) {
                   synchronized (lock) {
                      if (registerQueue.isEmpty()
                            && cancelQueue.isEmpty()) {//完成工作
                         worker = null;
                         break;
                      }
                   }
                }
             } catch (Throwable e) {
                ExceptionMonitor.getInstance().exceptionCaught(e);
                try {
                   Thread.sleep(1000);//线程休眠一秒
                } catch (InterruptedException e1) {
                   ExceptionMonitor.getInstance().exceptionCaught(e1);
                }
             }
          }
          if (selectable && isDisposing()) {//释放资源
             selectable = false;
             try {
                if (createdProcessor) {
                   processor.dispose();
                }
             } finally {
```

```
                try {
                    synchronized (disposalLock) {
                        if (isDisposing()) {
                            destroy();
                        }
                    }
                } catch (Exception e) {
                    ExceptionMonitor.getInstance().exceptionCaught(e);
                } finally {
                    disposalFuture.setDone();
                }
            }
        }
    }
private int registerHandles() {//注册服务器 sockets 句柄
    for (;;) {
        AcceptorOperationFuture future = registerQueue.poll();
        Map<SocketAddress, H> newHandles = new HashMap<SocketAddress, H
>();
        List<SocketAddress> localAddresses = future.getLocalAddresses();

        try {
            for (SocketAddress a : localAddresses) {
                H handle = open(a);//打开指定地址，返回服务器 socket 句柄
                newHandles.put(localAddress(handle), handle);//加入地址—
服务器 socket 映射表中
            }
            boundHandles.putAll(newHandles);//更新本地绑定地址集
            // and notify.
            future.setDone();//完成注册过程
            return newHandles.size();
        } catch (Exception e) {
            future.setException(e);
        } finally {
            // Roll back if failed to bind all addresses.
            if (future.getException() != null) {
                for (H handle : newHandles.values()) {
                    try {
                        close(handle);//关闭服务器 socket 句柄
                    } catch (Exception e) {
                        ExceptionMonitor.getInstance().exceptionCaught(e);

                    }
                }
```

```
                wakeup();
            }
        }
    }
    private void processHandles(Iterator<H> handles) throws Exception
{//处理来自客户端的连接请求
        while (handles.hasNext()) {
            H handle = handles.next();
            handles.remove();
            T session = accept(processor, handle);//为一个服务器 socket 句柄
handle 真正接收来自客户端的请求，在给定的所关联的 processor 上返回会话 session
            if (session == null) {
                break;
            }
            finishSessionInitialization(session, null, null);//结束会话初
始化
            // add the session tc the SocketIoProcessor
            session.getProcessor().add(session);
        }
    }
}
```

这个类中有个地方值得注意，就是 wakeup 方法，它是用来中断 select 方法的，当注册队列或取消注册队列发生变化时需要调用它，可以参看本类的一个子类 NioSocketAcceptor 的实现：

```
protected boolean select() throws Exception {
    return selector.select() > 0;
}
protected void wakeup() {
    selector.wakeup();
}
```

我们可以查阅 jdk 文档，它对 Selector 的 select 方法有如下解释：选择一组键，其相应的通道已为 I/O 操作准备就绪。 此方法执行**处于阻塞模式的选择操作**。仅在至少选择一个通道、调用此选择器的 wakeup 方法、当前的线程已中断，或者给定的超时期满（以先到者为准）后此方法才返回。

参考资料

1，《Java NIO 非阻塞服务器示例》

9. Mina2.0 框架源码剖析（八）

这篇来看看 AbstractPollingIoConnector 抽象类，它用于用于实现客户端连接的轮询策略。处理逻辑基本上和上一篇文章说的 AbstractPollingIoAcceptor 类似，它继承自 AbstractIoConnector，两个泛型参数分别是所处理的会话和客户端 socket 连接。底层的 sockets 会被不断检测，并当有任何一个 socket 需要被处理时就会被唤醒去处理。这个类封装了客户端 socket 的 bind,connect 和 dispose 等动作，其成员变量 Executor 用于发起连接请求，另一个 AbstractPollingIoProcessor 用于处理已经连接客户端的 I/O 操作请求，如读写和关闭连接。

其最重要的几个成员变量是：

```
private final Queue<ConnectionRequest> connectQueue = new ConcurrentLinke
dQueue<ConnectionRequest>();//连接队列
private final Queue<ConnectionRequest> cancelQueue = new ConcurrentLinked
Queue<ConnectionRequest>();// 取消连接队列
```

先来看看当服务端调用 connect 后的处理过程：

```
   protected final ConnectFuture connect0(
           SocketAddress remoteAddress, SocketAddress localAddress,
           IoSessionInitializer<? extends ConnectFuture> sessionInitialize
r) {
       H handle = null;
       boolean success = false;
       try {
           handle = newHandle(localAddress);
           if (connect(handle, remoteAddress)) {//若已经连接服务器成功
               ConnectFuture future = new DefaultConnectFuture();
               T session = newSession(processor, handle);//创建新会话
               finishSessionInitialization(session, future, sessionInitial
izer);//结束会话初始化
               session.getProcessor().add(session);//将剩下的处理交给 IoProce
```

```
ssor
                success = true;
                return future;
            }
            success = true;
        } catch (Exception e) {
            return DefaultConnectFuture.newFailedFuture(e);
        } finally {
            if (!success && handle != null) {
                try {
                    close(handle);
                } catch (Exception e) {
                    ExceptionMonitor.getInstance().exceptionCaught(e);
                }
            }
        }
        ConnectionRequest request = new ConnectionRequest(handle, sessionIn
itializer);
        connectQueue.add(request);//连接请求加入连接队列中
        startupWorker();//开启工作线程处理连接请求
        wakeup();//中断 select 操作
        return request;
    }
```

真正的负责处理客户端请求的工作都是 Worker 线程完成的,

```
private class Worker implements Runnable {
        public void run() {
            int nHandles = 0;
            while (selectable) {
                try {
                     int timeout = (int)Math.min(getConnectTimeoutMillis(),
1000L);//等待超时时间
                    boolean selected = select(timeout);//在超时时限内查看是否有
可以被处理的选择键（状态
                    nHandles += registerNew();//取出连接队列队头的连接请求，将其注
册一个用于连接的新的客户端 socket， 并把它加入连接轮询池中
                    if (selected) {
                        nHandles-= processSessions(selectedHandles());//处理连
接请求
                    }
                    processTimedOutSessions(allHandles());//处理超时连接请求
                    nHandles -= cancelKeys();
                    if (nHandles == 0) {
                        synchronized (lock) {
```

```
                        if (connectQueue.isEmpty()) {
                            worker = null;
                            break;
                        }
                    }
                }
            } catch (Throwable e) {
                ExceptionMonitor.getInstance().exceptionCaught(e);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e1) {
                    ExceptionMonitor.getInstance().exceptionCaught(e1);
                }
            }
        }
        if (selectable && isDisposing()) {
            selectable = false;
            try {
                if (createdProcessor) {
                    processor.dispose();
                }
            } finally {
                try {
                    synchronized (disposalLock) {
                        if (isDisposing()) {
                            destroy();
                        }
                    }
                } catch (Exception e) {
                    ExceptionMonitor.getInstance().exceptionCaught(e);
                } finally {
                    disposalFuture.setDone();
                }
            }
        }
    }
}
private int registerNew() {
    int nHandles = 0;
    for (; ;) {
        ConnectionRequest req = connectQueue.poll();//取连接队列队头请求
        if (req == null) {
            break;
        }
```

```
            H handle = req.handle;
            try {
                register(handle, req);//注册一个用于连接的新的客户端 socket， 并把
它加入连接轮询池中
                nHandles ++;
            } catch (Exception e) {
                req.setException(e);
                try {
                    close(handle);
                } catch (Exception e2) {
                    ExceptionMonitor.getInstance().exceptionCaught(e2);
                }
            }
        }
        return nHandles;
    }
private int processSessions(Iterator<H> handlers) {//处理连接请求
        int nHandles = 0;
        while (handlers.hasNext()) {
            H handle = handlers.next();
            handlers.remove();
            ConnectionRequest entry = connectionRequest(handle);
            boolean success = false;
            try {
                if (finishConnect(handle)) {//连接请求成功完成，创建一个新会话
                    T session = newSession(processor, handle);
                    finishSessionInitialization(session, entry, entry.getSes
sionInitializer());//结束会话初始化
                    session.getProcessor().add(session);//将剩下的工作交给 IoPr
ocessor 去处理
                    nHandles ++;
                }
                success = true;
            } catch (Throwable e) {
                entry.setException(e);
            } finally {
                if (!success) {//若连接失败，则将此连接请求放到取消连接队列中
                    cancelQueue.offer(entry);
                }
            }
        }
        return nHandles;
    }
private void processTimedOutSessions(Iterator<H> handles) {//处理超时的连接
```

```
请求
        long currentTime = System.currentTimeMillis();//当前时间

        while (handles.hasNext()) {
            H handle = handles.next();
            ConnectionRequest entry = connectionRequest(handle);
            if (currentTime >= entry.deadline) {//当前时间已经超出了连接请求的底
限
                entry.setException(
                        new ConnectException("Connection timed out."));
                cancelQueue.offer(entry);//将此连接请求放入取消连接队列中
            }
        }
    }
private int cancelKeys() {//把取消队列中的连接请求给 cancel 掉
        int nHandles = 0;
        for (; ;) {
            ConnectionRequest req = cancelQueue.poll();
            if (req == null) {
                break;
            }
            H handle = req.handle;
            try {
                close(handle);//关闭对应的客户端 socket
            } catch (Exception e) {
                ExceptionMonitor.getInstance().exceptionCaught(e);
            } finally {
                nHandles ++;
            }
        }
        return nHandles;
    }
```