Cindy User Guide

http://cindy.sourceforge.net

版本号	修改时间	修改人	备注
1.0	2005-03-12	Roger Chen	Based on Cindy-2.3 RC
2.0	2006-03-02	Roger Chen	Based on Cindy-3.0 a3

Cindy 简介

Cindy 是一个基于 Java NIO 的异步 I/O 框架，支持 TCP、UDP 以及 Pipe，为应用提供了一个统一的模型，并能够方便的切换异步和同步操作。

为什么不使用 Java IO?

Java IO 包采用阻塞模型来处理网络操作。假设应用调用了 read 方法读取来自网络流上的数据，而数据尚未到达本机，则对 read 方法的调用将一直等到数据到达并成功接收后才返回。由于 IO 包的所有操作会一直阻塞当前线程，为了不影响其他事务的处理，在一般情况下应用总是会用一个独立线程或线程池来处理这些 I/O 操作。

Java IO 包阻塞模型的优点就是非常简单易用，如果配合上线程池，效率也非常不错，所以得到了广泛的应用。但这种简单模型也有其固有的缺点：扩展性不足。如果应用只需要进行少量的网络操作，那么开启若干个单独的 I/O 线程无伤大雅；但是如果是实现一个服务端的应用，需要同时处理成千上万个网络连接，采用阻塞模型的话就要同时开启上千个线程。虽然现在强劲的服务器能够负担起这么多线程，但操作系统花在线程调度上的时间也会远远多于用于处理网络操作上的时间。

为什么要使用 Java NIO?

采用 IO 包的阻塞模型，如果数据量不大的话，则线程的大部分时间都会耗费在等待上。对于希缺的服务器资源而言，这是一种极大的浪费。

在 Java 1.4 中引入的 NIO 包里，最引人注目的就是加入了非阻塞 I/O。和 IO 包提供的阻塞模型不同的是，对一个非阻塞的连接进行操作，调用会立即返回，而不是等待。假设应用调用了 read 方法读取来自网络流上的数据，而数据尚未到达本机，则对 read 方法的调用将立即返回，并通知应用目前只能读到 0 个字节。应用可以根据自身的策略来进行处理，比如读取其他网络连接的数据等等，这就使得一个线程管理多个连接成为可能。

NIO 包还引入了 Selector 机制，将一个非阻塞连接注册在 Selector 上，应用就不需去轮询该连接当前是否可以读取数据，而是接收到数据后 Selector 会通知该连接。这样就使得可以用更少的线程数来管理更多的连接。

为什么选择 Cindy，而不直接使用 NIO?

Java NIO 包虽然提供了非阻塞 I/O 模型，但是直接使用 NIO 的非阻塞 I/O 需要成熟的网

络编程经验，处理众多底层的网络异常，以及维护连接状态，判断连接超时等等。对于关注于其业务逻辑的应用而言，这些复杂性都是不必要的。不同 Java 版本的 NIO 实现也会有一些 Bug，Cindy 会巧妙的绕开这些已知的 Bug 并完成相应功能。并且 NIO 本身也在不断发展中，Java 1.4 的 NIO 包中只实现了 TCP/UDP 单播/Pipe，Java 5.0 中引入的 SSLEngine 类使得基于非阻塞的流协议（TCP/Pipe）支持 SSL/TLS 成为可能，在未来的版本中还可能会加入非阻塞多播的实现。Cindy 会关注这些新功能，并纳入到统一的框架中来。

而且 Cindy 虽然目前的实现是基于 NIO，但它会不仅仅局限于 NIO。等到一些基于操作系统本身实现的 AIO（Asynchronous IO）类库成熟后，它也会加入对这些类库的支持，通过操作系统本身实现的 AIO 来提高效率。

如果应用程序只想使用一种高效的模型，而不想关心直接使用 NIO 所带来的这些限制，或希望将来无需更改代码就切换到更高效率的 AIO 实现上，那么 Cindy 会是一个很好的选择。并且使用 Cindy，应用可以在同步和异步之间进行无缝切换，对于大部分操作是异步，可某些特殊操作需要同步的应用而言，这极大的提高了易用性。

Hello world

场景：服务端监听本地的 1234 端口，打印任何收到的消息到控制台上；客户端建立 TCP 连接到本地的 1234 端口，发送完"Hello world!"，然后断开连接。

1. 基于 Java IO 包的客户端示例

基于 Java IO 包的阻塞模型的示例，用于对比，不做额外说明。（异常处理代码略）

```
Socket socket = new Socket("localhost",1234);
OutputStream os = new BufferedOutputStream(socket.getOutputStream());
os.write("Hello world!".getBytes());
os.close();
socket.close();
```

2. 基于 Java IO 包的服务端示例

基于 Java IO 包的阻塞模型的示例，用于对比，不做额外说明。（异常处理代码略）

```
ServerSocket ss = new ServerSocket(1234);
while (true) {
```

```
        final Socket socket = ss.accept();
        new Thread() {
            public void run() {
                BufferedReader br = new BufferedReader(
                    new InputStreamReader(socket.getInputStream()));
                String line = null;
                while ((line = br.readLine()) != null) {
                    System.out.println(line);
                }
                socket.close();
            };
        }.start();
    }
```

3. 基于 Cindy 的同步客户端示例

```
Session session = SessionFactory.createSocketSession();
session.setRemoteAddress(new InetSocketAddress("localhost", 1234));
session.start().complete();
Packet packet = new DefaultPacket(BufferFactory.wrap("Hello world!".getBytes()));
session.flush (packet).complete();
session.close().complete();
```

Step 1: the Session interface

Session 是网络连接的抽象，每个 Session 代表着一个连接。SocketSession 代表着 TCP 连接，DatagramSession 代表着 UDP 连接，PipeSession 代表着 Pipe 连接等等。

在这里通过 SessionFactory.createSocketSession()创建了一个 TCP 连接。

Step 2: set the remote address

TCP 需要先设置好要连接的地址才能开始连接，在这里是连接到本机的 1234 端口。

Step 3: start session

参数设置完成后就可以开始连接了。和 Java IO 中的阻塞同步调用不同的是，由于 Cindy

是一个异步 I/O 框架，在调用 session.start()方法并返回后，此时连接可能还没有建立成功。如果我们想要切换到阻塞方式，比如等连接建立成功后才返回，则可以通过 start 方法返回的 Future 对象来切换。

Session 中所有的异步方法(start/close/flush/send)的返回值均为 Future 对象，该对象代表着所进行的异步操作。我们可以通过三种方式来处理 Future 对象：

- 阻塞：调用 Future.complete()，则该方法会一直等到异步操作完成后才返回；或者可以调用 Future.complete(int timeout)来等待异步操作完成，如果在指定时间内操作未完成，则该方法也会返回。值得注意的是操作完成包括操作成功和操作失败两种状态。
- 轮询：周期性的调用 Future.isCompleted()方法来查询异步操作是否完成
- 回调：通过 Future.addListener()方法加入自定义的 FutureListener，在异步操作结束后，Future 会通知加入的 FutureListener

在该示例中我们采用的是第一种方式，等待连接建立完成后才返回。可以通过 session.isStarted()来判断连接是否建立成功，在这个简单的示例里假设不会出现任何连接错误，所以就不再判断异常情况了，不过在正式的应用中应当注意相关的判断。

Step 4: the Packet/Buffer interface

任何数据要在网络上传输最终都得转换成字节流，Buffer 接口就是字节流的抽象。注意的是这里使用的 Buffer 是 net.sf.cindy.Buffer，而没有采用 java.nio.Buffer，这个设计上的取舍可以参考以后的介绍。

传输除了要有内容外，还得有目的地，这样才能送到正确的地址。Packet 接口就代表着要发送的数据包，包括一个 Buffer 对象和目的地——SocketAddress。对于流协议（如 TCP/Pipe）而言，目的地在建立连接的时候就已经设置好了，所以发送过程中无需再进行设置；对于非连接型的 UDP，则需要在发送时指定每个包的目的地址。

在这里要将“Hello world!”转换为 Buffer 对象，可以通过 BufferFactory 来完成：BufferFactory.wrap(“Hello world!”.getBytes())，将一个字节数组包装成 Buffer。

由于是 TCP 连接，无需指定 Buffer 目的地址，所以可以简单的构造一个只有 Buffer 而没有 SocketAddress 的 Packet 对象：new DefaultPacket(buffer)。（DefaultPacket 是 Packet 接口的默认实现）

Step 5: send packet

Session 的 flush 方法和 start 方法一样都是异步方法，在调用 flush 并返回时，数据可能并没有发送完成，在该示例中仍然采用阻塞方式等待数据发送完成后才返回：session.flush(packet).complete()。

Step 6: close session

发送完成后需要关闭连接，close 方法同样是一个异步方法，在这里等待连接完全关闭后才返回：session.close().complete()。

从这个同步的示例可以看到，虽然 Cindy 是一个异步 I/O 框架，但用它来完成同步的 I/O 操作也是一件非常简单的事情。

4. 基于 Cindy 的异步客户端示例

```
final Session session = SessionFactory.createSocketSession();
session.setRemoteAddress(new InetSocketAddress("localhost", 1234));
Future future = session.start();
future.addListener(new FutureListener() {
    public void futureCompleted(Future future) throws Exception {
        Packet packet = new DefaultPacket(BufferFactory.wrap("Hello
    world!".getBytes()));
        session.flush (packet).complete();
        session.close();
    }
});
```

前面的三行代码和同步的示例没有什么区别，只是在第四行代码中采用了回调的方式来处理 Future 对象，而不是通过阻塞的方式。当连接建立完成后，框架会调用所加入 FutureListener 的 futureCompleted 方法，应用可以在该方法中做相应的事件处理。

在发送 Packet 的过程中，其实也可以采用回调的方式来处理。在这里仍采用同步方法处理，一方面是减少内嵌类的数量，另一方面是示例 Cindy 可以非常容易的进行同步和异步操作的切换。

5. 基于 Cindy 的服务端示例

```
SocketSessionAcceptor acceptor = SessionFactory.createSocketSessionAcceptor();
acceptor.setLocalPort(1234);
Session session = null;
while ((session = acceptor.accept()) != null) {
    session.setSessionHandler(new SessionHandlerAdapter() {
        public void objectReceived(Session session, Object obj) throws Exception {
            System.out.println(obj);
        }
    });
    session.start();
}
```

Step 1: the SocketSessionAcceptor interface

SocketSessionAcceptor 代表着 TCP 的服务端，类似于 IO 包的 ServerSocket。

Step2: set listen port

设置 TCP 服务端的监听端口，在这里是 1234 端口。

Step 3: accept incoming session

非常类似于 IO 的 ServerSocket，通过一个循环来接受连接。

Step 4: set incoming session's SessionHandler and start session

设置连接的 SessionHandler。SessionHandler 接口用于处理 Session 产生的各种事件，在这里我们仅仅关心对象接收事件，当接收到对象后，将该对象打印到控制台上。（注：在这里打印出来的并不是 Hello world!字符串，而是接收到的 Packet）

SessionHandler 的更多介绍请参阅后面章节。

PacketEncoder/PacketDecoder

1. PacketEncoder

在前面的 Hello world 示例中，我们都是通过 session.flush 方法来发送数据，而该方法只

接收 Packet 类型的参数，这就要求我们在发送任何数据前都要先进行转换。比如在前面示例中，我们就得先将”Hello world!”字符串转换成一个代表”Hello world!”的 Packet，然后再进行发送。

这种做法没有什么问题，其唯一的缺陷在于将发送逻辑和序列化逻辑耦合在一起。比如在上面的示例中，发送逻辑是将“Hello world!”发送出去，序列化逻辑是”Hello world”字符串的字节表示。虽然有一定的关联，但的确是两种不同的逻辑，比如我们可以更改序列化逻辑，通过 Serializable 接口来序列化”Hello world”，但这并不影响发送逻辑。

PacketEncoder 的作用就是分离发送逻辑和序列化逻辑。对于应用而言，在发送时它只需要把要发送的对象传递给 Session，至于怎么序列化，则由 Session 所关联的 PacketEncoder 来处理。所以在上面的 Hello world 示例中，发送逻辑可以改为：

```
session.send("Hello world!");
```

序列化逻辑可以通过 PacketEncoder 来设置：

```
session.setPacketEncoder(new PacketEncoder() {
    public Packet encode(Session session, Object obj) throws Exception {
        return new DefaultPacket(BufferFactory.wrap(obj.toString().getBytes()));
    }
});
```

如果要改变序列化逻辑，比如通过 Serializable 接口来序列化，则只需要更改 PacketEncoder，而不需要改动发送代码：

```
session.setPacketEncoder(new SerialEncoder());
```

通过 Cindy 内置的 PacketEncoderChain，应用可通过 Session 发送任意对象，而将序列化逻辑完全交给 PacketEncoder。如下面的伪码所示：

```
PacketEncoderChain chain = new PacketEncoderChain();
chain.addPacketEncoder(new Message1Encoder());
chain.addPacketEncoder(new Message2Encoder());
session.setPacketEncoder(chain);
session.send(new Message1());
session.send(new Message2());
```

2. PacketDecoder

PacketEncoder 是用来处理序列化逻辑的，那么相应的 PacketDecoder 则是用来处理反序列化逻辑的。

发送方通过 session.send 方法可以发送任意对象，Session 关联的 PacketEncoder 会将该对象转换为 Packet 发送出去；接收方收到了 Packet 后，也可以通过其关联的 PacketDecoder 将该 Packet 转换为一个对象，再通知应用。假设发送方用了 SerialEncoder 来发送序列化对象，则接收方就可以使用 SerialDecoder 来进行反序列化，然后直接对对象进行处理。

```
session.setPacketDecoder(new SerialDecoder());
session.setSessionHandler(new SessionHandlerAdapter() {
    public void objectReceived(Session session, Object obj) throws Exception {
        //obj 即为反序列化后得到的对象
    }
}
```

SessionHandler/SessionHandlerAdapter

SessionHandler 接口用于处理 Session 的各种事件。比如当连接建立成功后，会触发 SessionHandler 的 sessionStarted 事件；连接关闭后，会触发 SessionHandler 的 sessionClosed 事件；对象发送成功后会触发 SessionHandler 的 objectSent 事件；对象接收成功后会触发 SessionHandler 的 objectReceived 事件等等。

SessionHandlerAdapter 是 SessionHandler 的空实现。即如果你仅仅对 SessionHandler 中某几个事件感兴趣，就不用全部实现 SessionHandler 中定义的各种方法，而只需要继承自 SessionHandlerAdapter，实现感兴趣的事件即可。

通过 SessionHandler，可以极大的减少内嵌类的数量。如前面的异步 Hello world 示例，如果用 SessionHandler 来改写，则会是：

```
Session session = SessionFactory.createSocketSession();
session.setRemoteAddress(new InetSocketAddress("localhost", 1234));
session.setSessionHandler(new SessionHandlerAdapter() {
    public void sessionStarted(Session session) throws Exception {
        Buffer buffer = BufferFactory.wrap("Hello world!".getBytes());
```

```
                Packet packet = new DefaultPacket(buffer);
                session.send(packet);
            }
            public void objectSent(Session session, Object obj) throws Exception {
                session.close();
            };
        };
        session.start();
```

在上面的代码中，当 sessionStarted 事件触发后，即 Session 成功建立后，会发送"Hello world!"消息；当 objectSent 事件触发后，即消息发送成功，调用 session.close()异步关闭连接。所有的这些处理都是异步的，并且仅仅使用了一个内嵌类。

SessionHandler 中一共定义了如下几个方法：

- void sessionStarted(Session session) throws Exception //连接已建立
- void sessionClosed(Session session) throws Exception //连接已关闭
- void sessionTimeout(Session session) throws Exception //连接超时
- void objectReceived(Session session, Object obj) throws Exception //接收到了对象
- void objectSent(Session session, Object obj) throws Exception //发送了对象
- void exceptionCaught(Session session, Throwable cause) //捕捉到异常

如果通过 session.setSessionTimeout 方法设置了超时时间，则在指定的时间内没有接收或发送任何数据就会触发 sessionTimeout 事件。发生了该事件并不代表着连接被关闭，应用可以选择关闭该空闲连接或者发送某些消息来检测网络连接是否畅通。默认情况下 sessionTimeout 为 0，即从不触发 sessionTimeout 事件。

如果通过 PacketEncoder 发送了任何对象，则 objectSent 事件将被触发（注：通过 flush 方法发送的 Packet 不会触发 objectSent 事件，只有通过 send 方法发送的对象才会触发 objectSent 事件）；通过 PacketDecoder 接收到任何对象，则 objectReceived 事件将被触发，一般情况下应用都通过监听该事件来对接收到的对象做相应处理。

exceptionCaught 事件代表着 session 捕捉到了一个异常，这个异常可能是由于底层网络所导致，也可能是应用在处理 SessionHandler 事件时所抛出来的异常。**请注意，如果应用在处理 exceptionCaught 事件中抛出运行期异常，则该异常不会再度触发 exceptionCaught 事件，否则可能出现死循环。**由于应用无法处理底层网络所引发的异常，而且这些异常的记录

对应用本身并没有太大的帮助，所以在部署稳定后，可以通过指定运行期参数-Dnet.sf.cindy.disableInnerException 来取消对底层网络异常的分发。

SessionFilter/SessionFilterAdapter

SessionFilter 与 SessionHandler 有些类似，均用于处理 Session 的各种事件，不同的则是 SessionFilter 先处理这些事件，并判断是否需要把该事件传递给下一个 SessionFilter。等到所有 SessionFilter 处理完成后，事件才会传递给 SessionHandler 由其来处理。SessionFilterAdapter 是 SessionFilter 的空实现，默认是把事件传递给下一个 SessionFilter。

SessionFilter 中定义了如下几个方法：

- void sessionStarted(SessionFilterChain filterChain) throws Exception;
- void sessionClosed(SessionFilterChain filterChain) throws Exception;
- void sessionTimeout(SessionFilterChain filterChain) throws Exception;
- void objectReceived(SessionFilterChain filterChain, Object obj) throws Exception;
- void objectSent(SessionFilterChain filterChain, Object obj) throws Exception;
- void exceptionCaught(SessionFilterChain filterChain, Throwable cause);
- void packetReceived(SessionFilterChain filterChain, Packet packet) throws Exception;
- void packetSend(SessionFilterChain filterChain, Packet packet) throws Exception;
- void packetSent(SessionFilterChain filterChain, Packet packet) throws Exception;

其中前六种方法和 SessionHandler 的作用一致，后三种方法则是用于处理接收和发送的 Packet 的。

基于 SessionFilter，应用可以做很多扩展而不影响原来的业务处理。比如数据包相关的扩展：加入 SSLFilter，则所发送的数据都会被 SSL 编码后才发送，接收的数据会先被解码成明文才接收；加入 ZipFilter，则可以压缩所发送的数据，接收时再解压缩。比如统计的扩展：加入 StatisticFilter，则可以统计发送和接收的字节数，以及发送速率。比如 ACL 的扩展：加入 AllowListFilter/BlockListFilter，则可以允许指定或限制某些 IP 地址访问；加入 LoginFilter，如果用户没有登录，则不把事件传递给后面处理业务逻辑的 SessionFilter 或 SessionHandler。比如线程处理的扩展：加入 ThreadPoolFilter，可以指定让某个线程池来进行后面事件的处理。比如日志记录的扩展：加入 LogFilter，则可以记录相应的事件信息。

所列举的这些只是一些基于 SessionFilter 的常见应用，应用可以根据自身的业务需要来进行选择。Cindy 所推荐的实践是将不同的业务逻辑分散到不同的 SessionFilter 中，在

SessionHandler 中只处理核心逻辑。

在这里可以示范一个 ZipFilter 的伪码：

```
public class ZipFilter extends SessionFilterAdapter {
    public void packetReceived(SessionFilterChain filterChain, Packet packet) throws Exception {
        Packet unzippedPacket = unzip(packet);    //解压缩
        super.packetReceived(filterChain, unzippedPacket);//把解压缩后的包传递给下一个 Filter
    }

    public void packetSend(SessionFilterChain filterChain, Packet packet) throws Exception {
        Packet zippedPacket = zip(packet);    //压缩
        super.packetSend(filterChain, zippedPacket); //把压缩后的包传递给下一个 Filter
    }
}
```

Buffer/Packet

在前面的示例中我们已经接触到了 Buffer/Packet，Buffer 是字节流的抽象，Packet 是网络包的抽象。

Java NIO 中已经提供了 java.nio.ByteBuffer 类用于表示字节流，为什么不直接使用 java.nio.ByteBuffer？

java.nio.ByteBuffer 虽然是 NIO 中表示字节流的标准类，但是对于高负荷的网络应用而言，其设计上存在着以下缺陷：

- ByteBuffer 并不是一个接口，而是一个抽象类。最为关键的地方是其构造函数为包级私有，这意味着我们无法继承自 ByteBuffer 构造子类
- ByteBuffer 仅仅是其所持有内容的一个外部包装，多个不同的 ByteBuffer 可以共享相同的内容。比如通过 slice、duplicate 等方法构造一个新的 ByteBuffer，该新 ByteBuffer 和原 ByteBuffer 共享的是同一份内容。这就意味着无法构造基于 ByteBuffer 的缓存机制。

比如如下代码：

```
ByteBuffer buffer1 = ByteBuffer.allocate(100);
ByteBuffer buffer2 = buffer1.slice();
System.out.println(buffer1 == buffer2);
System.out.println(buffer1.equals(buffer2));
```

打印出来的都是 false，而实际上两个 ByteBuffer 共享的是同一份数据。在不经意的情况下，可能发生应用把两个 ByteBuffer 返回缓存中，被缓存当成是不同的对象进行处理，可能破坏数据完整性。

为什么 Cindy 使用自定义的 Buffer 接口？

- Buffer 是一个接口，如果对现有的实现类不满意，应用可以方便的加入自己的实现
- 支持一系列的工具方法，比如 indexOf/getString/putString/getUnsignedXXX 等等，加入这些常用方法会给应用带来很大的方便
- 可以非常方便的与 nio 中的 ByteBuffer 做转换，并且效率上不会有太大损失。由于大部分的网络类库都是基于 nio 的 ByteBuffer 来设计的，这样保证了兼容性
- NIO 的 ByteBuffer 无法表示 ByteBuffer 数组，而 Cindy 中提供了工具类把 Buffer 数组包装成一个 Buffer
- Cindy 的 Buffer 是可缓存的，对于高负荷的网络应用而言，这会带来性能上优势

在一般情况下，应用应该直接通过 BufferFactory 类来构造 Buffer 实例。目前 BufferFactory 类中有如下方法：

- Buffer wrap(byte[] array)
- Buffer wrap(byte[] array, int offset, int length)
- Buffer wrap(ByteBuffer buffer)
- Buffer wrap(Buffer[] buffers)
- Buffer allocate(int capacity)
- Buffer allocate(int capacity, boolean direct)

前四种方法都是包装方法，将现有的字节数组、ByteBuffer 以及 Buffer 数组包装成一个单一的 Buffer 对象。第五和第六种方法是构造一个新的 Buffer 对象，新构造 Buffer 对象的内容是不确定的（java.nio.ByteBuffer allocate 得到的内容是全 0），可能来自缓存或直接生成，依赖于对 Buffer 缓存的配置。一般情况下都是通过第五种方法构造新的 Buffer 对象，通过

运行期参数-Dnet.sf.cindy.useDirectBuffer 可以改变默认行为，是构造 Non-Direct Buffer 还是构造 Direct Buffer。

默认情况下生成的 Buffer 是可以被缓存，在调用了 session.send/flush 方法后，Buffer 的内容就会被释放掉，或者手动调用 Buffer.release 也能释放 Buffer 所持有的内容。通过 Buffer.isReleased 方法可以可以判断当前的 Buffer 是否被释放掉。对已经释放掉的 Buffer 进行 get/put 操作都会导致 ReleasedBufferException，但是对 position/limit/capacity 的操作还是有效的。如果应用不希望 Buffer 的内容被释放，可以通过设置 permanent 属性为 true 来使得 Buffer 的内容不被释放。

示例

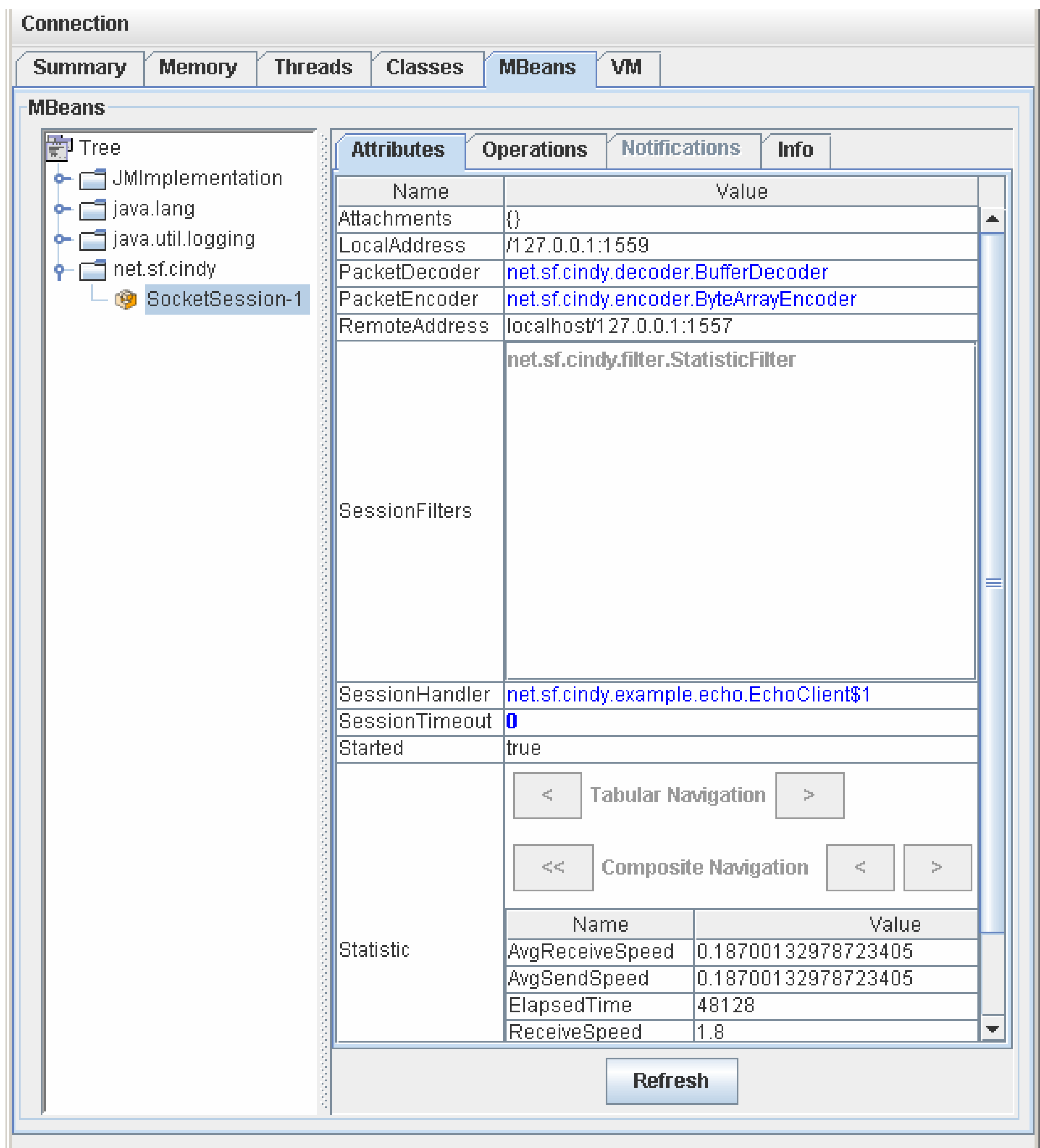
可见 example 目录下源代码

Advanced Topic

1. JMX 支持

通过运行期属性-Dnet.sf.cindy.useJmx 可以开启 JMX 支持(Java 5.0 中内置了 JMX 支持，如果运行在 Java 1.4 版本中，则需要手工下载 JMX API)。

下图是通过 jconsole 进行 JMX 管理的示例：



2. 流量控制

当网络接收的速度大于应用的处理速度时，如果不控制接收速率，则收到的消息会在队列中堆积，应用无法及时处理而造成内存溢出。Cindy 3.0 中加入了流量控制功能，当接收队列中消息超过指定数量时，Cindy 会放慢网络接收速度。该功能对于防止内存溢出以及应用程序调试有很大帮助。

可以通过运行期参数-Dnet.sf.cindy.dispatcherQueueCapacity 来指定队列中最多可以堆积的消息数。默认值是 1000，即队列中消息数超过 1000，网络接收速度就会放缓，等到消息数少于 1000 后，接收速度就会恢复正常。

3. Read packet size

Read packet size 是指接收时每次读包所构造的缓冲区大小。对于 TCP 而言，这个值影响到的仅仅是效率；对于 UDP 则影响到数据正确性。

对于 UDP 应用，假设发送方发送的包所携带数据大小为 1000 字节，接收方的 read packet size 设置为 600 字节，则后面 400 个字节会被丢弃，这样构造出来的逻辑包可能会不正确。

对于 TCP 应用，如果逻辑包是定长的，这个值最好也设为逻辑包的长度。该值太少，则可能导致在过多的包组合操作（比如逻辑包长度为 1000 字节，read packet size 设置为 100，则可能需要把 10 次接收到的 packet 组合成一个 packet 才能构造出一个逻辑包）；该值太大，可能造成内存的浪费（不过由于 Buffer 缓存的存在会缓解这一情况）。

默认的 read packet size 是 8192 字节，可以通过运行期属性-D net.sf.cindy.readPacketSize 来更改。如果对单独的 Session 进行更改，则可以通过 session 的 readPacketSize 属性进行设置。

4. Direct ByteBuffer vs Non-Direct ByteBuffer

java.nio.ByteBuffer 引入了这两种类型的 ByteBuffer，在 Cindy 中也有基于这两种 ByteBuffer 的 Buffer 包装类。那么这两者的区别在什么地方？在什么环境下采用哪种类型的 ByteBuffer 会更有效率？

Non-direct ByteBuffer 内存是分配在堆上的，直接由 Java 虚拟机负责垃圾收集，你可以把它想象成一个字节数组的包装类，如下伪码所示：

```
HeapByteBuffer extends ByteBuffer {
    byte[] content;
    int position, limit, capacity;
    ......
}
```

而 Direct ByteBuffer 是通过 JNI 在 Java 虚拟机外的内存中分配了一块，该内存块并不直接由 Java 虚拟机负责垃圾收集，但是在 Direct ByteBuffer 包装类被回收时，会通过 Java Reference 机制来释放该内存块。如下伪码所示：

```
DirectByteBuffer extends ByteBuffer {
```

```
        long address;
        int position, limit, capacity;
        protected void finalize() throws Throwable{
            //释放内存块，该段代码仅仅用于演示，真正的 Direct ByteBuffer 并不是通过
            finalize 来释放的
            releaseAddress();
            ......
        }
        ......
    }
```

除开以上的这些外，如果我们查找 Java 实现类的代码，就可以了解到这两者之间更深入的区别。比如在 Sun 的 Java 实现中，绝大部分 Channel 类都是通过 sun.nio.ch.IOUtil 这个工具类和外界进行通讯的，如 FileChannel/SocketChannel 等等。简单的用伪码把 write 方法给表达出来（read 方法也差不多，就不多做说明了）：

```
    int write(ByteBuffer src, ......) {
        if (src instanceof DirectBuffer)
            return writeFromNativeBuffer(...);
        ByteBuffer direct = getTemporaryDirectBuffer(src);
        writeFromNativeBuffer(direct,......);
        updatePosition(src);
        releaseTemporaryDirectBuffer(direct);
    }
```

是的，在发送和接收前会把 Non-direct ByteBuffer 转换为 Direct ByteBuffer，然后再进行相关的操作，最后更新原始 ByteBuffer 的 position。这意味着什么？假设我们要从网络中读入一段数据，再把这段数据发送出去的话，采用 Non-direct ByteBuffer 的流程是这样的：

网络 --> 临时的 Direct ByteBuffer --> 应用 Non-direct ByteBuffer --> 临时的 Direct ByteBuffer --> 网络

而采用 Direct ByteBuffer 的流程是这样的：

网络 --> 应用 Direct ByteBuffer --> 网络

可以看到，除开构造和析构临时 Direct ByteBuffer 的时间外，起码还能节约两次内存拷贝的时间。那么是否在任何情况下都采用 Direct Buffer 呢？

答案是否定的。对于大部分应用而言，两次内存拷贝的时间几乎可以忽略不计，而构造和析构 Direct Buffer 的时间却相对较长。在 JVM 的实现当中，某些方法会缓存一部分临时 Direct ByteBuffer，意味着如果采用 Direct ByteBuffer 仅仅能节约掉两次内存拷贝的时间，而无法节约构造和析构的时间。就用 Sun 的实现来说，write(ByteBuffer)和 read(ByteBuffer)方法都会缓存临时 Direct ByteBuffer，而 write(ByteBuffer[])和 read(ByteBuffer[])每次都生成新的临时 Direct ByteBuffer。

根据这些区别，在选择 ByteBuffer 类型上有如下的建议：

- 如果你做中小规模的应用（在这里，应用大小是按照使用 ByteBuffer 的次数和规模来做划分的），并不在乎这些细节问题，请选择 Non-direct ByteBuffer
- 如果采用 Direct ByteBuffer 后性能并没有出现你所期待的变化，请选择 Non-direct ByteBuffer
- 如果没有 Direct ByteBuffer Pool，尽量不要使用 Direct ByteBuffer
- 除非你确定该 ByteBuffer 会长时间存在，并且和外界有频繁交互，可采用 Direct ByteBuffer
- 如果采用 Non-direct ByteBuffer，那么采用非聚集(gather)的 write/read(ByteBuffer)效果反而可能超出聚集的 write/read(ByteBuffer[])，因为聚集的 write/read 的临时 Direct ByteBuffer 是非缓存的

基本上，采用 Non-direct ByteBuffer 总是对的！因为内存拷贝需要的开销对大部分应用而言都可以忽略不计。在 Cindy 中，一般的应用只需要通过 BufferFactory.allocate 方法来得到 Buffer 实例即可，默认设置下采用的是 Non-Direct Buffer；通过 BufferFactory.wrap 方法包装字节数组或 ByteBuffer 得到的 Buffer 类也有很高的效率；只有通过 BufferFactory.wrap(Buffer[])方法目前还处于实验阶段，其效率不一定比生成一个大的 Buffer，然后拷贝现有内容更快。如果应用非常注重效率，要使用该方法上要多加注意。

5. 运行期属性设置

更多的运行期属性设置可见 net.sf.cindy.util.CindyConstants 类中的声明。