Netty Guide

Netty(V3.2.2)

2010.11.04

Email	cjswjddbs@naver.com
Name	천정윤(JeongYun Cheon)
Twitter	Yuu_HeaVen
Blog	http://withoutwing.tistory.com

위 문서는 http://kkamdung.tistory.com/40에 구한 The_Netty_Project_3.1_User_Guide.doc번역본과 Netty API 그외 기타등등의 삽질로 작성한 문서입니다.
지금에 와서는 조금더 깔끔하게 정리할 수 있겠지만..
귀찮아서 하지못하겠습니다..

하지만 한글로된 문서가 적기때문에 유용할거라 생각하고 블로그를 통해 공개합니다.

참고로 위문서는 틀린곳이 엄청많을 수 있으니 그냥 참고용으로만 사용하자...(정말이다...)

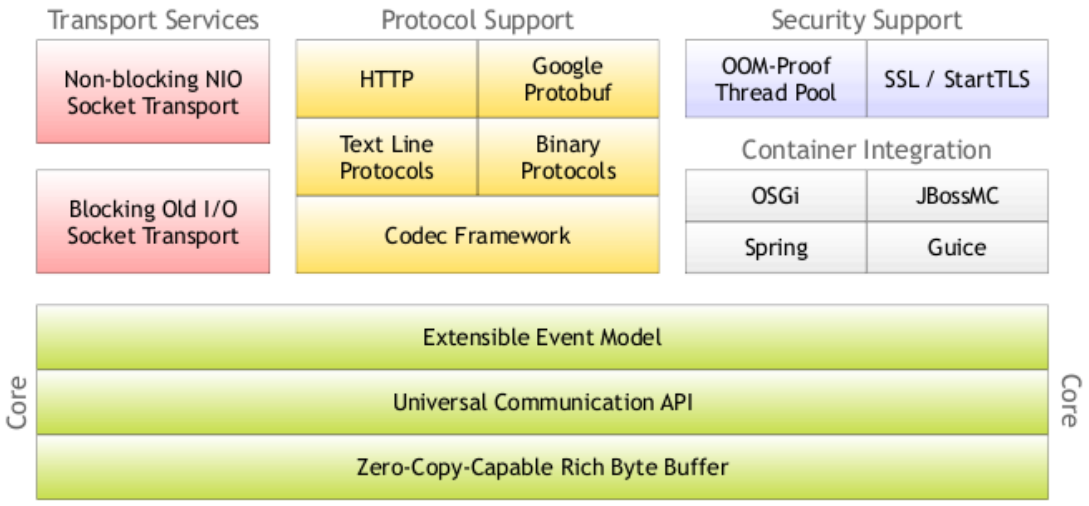
Table of Contents

1 개요

1.1 Netty

-. 자바 네트워크 어플리케이션 프레임워크
-. 이벤트 기반이며 비동기 방식이다.
-. Interceptor Chain 패턴기반이다.
-. Old, New I/O 모두 지원
-. NIO구현 벤더별 최적화
-. 과부하 및 Dos 방어 메커니즘을 포함하고있다.
-. 제한된 환경지원(Androiud, Applet, WebStart)
-. 다양한 컨테이너 지원(JbossMc, OSGi, Guice, Spring)

1.2 Architecture



1.2.1 Rich Buffer Data Structure

Netty는 NIO ByteBuffer 대신 자신만의 버퍼 API를 사용하여 바이트를 표현한다. 이러한 접근 방법은 ByteBuffer를 사용하는 것보다 훨씬 더 큰 장점을 가지고 있다. Netty의 새로운 버퍼 타입인 ChannelBuffer는 ByteBuffer의 문제점들을 해결하고 네트워크 어플리케이션 개발자의 요구를 충족시키기 위해 처음부터 설계되었다. 기능들을 열거해보면 다음과 같다.

- 필요할 경우 자신만의 버퍼 타입을 정의할 수 있다.
- 내장된 복합 버퍼 타입에 의해 투명한 제로 복사(**zero copy**)가 수행된다.
- StringBuffer와 같이 필요시 용량이 증가하는 동적 버퍼 타입이 제공된다.
- flip() 메소드를 더 이상 호출할 필요가 없다.
- 종종 ByteBuffer보다 더 빠르다.(java nio)

1.2.2 Universal Asynchronous I/O API

설상 가상으로, 자바 New IO (NIO) API는 기존 블록킹 IO (OIO) API와 호환되지 않으며, NIO 2 (AIO) 역시 그러할 것이다. 이러한 모든 API들은 설계 및 성능 측면에서 서로 다르기 때문에, 여러분은 종종 구현을 시작하기도 전에 어플리케이션이 의존할 API를 결정해야만 한다.

Netty는 **Channel**이라 불리는 일관된 비 동기 I/O 인터페이스를 가지고 있다. 이는 점 대 점 통신에 필요한 모든 오퍼레이션들을 추상화 한다. 즉, 여러분이 Netty 전송 타입으로 어플리케이션을 작성했다면, 이 어플리케이션은 다른 Netty 전송 타입 상에서 실행될 수 있다. Netty는 일관된 API를 통해 필수적인 전송 타입을 제공한다.

- NIO 기반 TCP/IP 전송 타입 (org.jboss.netty.channel.socket.nio)
- OIO 기반 TCP/IP 전송 타입 (org.jboss.netty.channel.socket.oio)
- OIO 기반 UDP/IP 전송 타입.
- 로컬 전송 타입 (org.jboss.netty.channel.local)
  -. 한 가상머신에서의 두파트의 가상통신이 가능하다.

1.2.3 Event Model based on the Interceptor Chain Pattern

-. Extensible Event Model (확장가능한 이벤트 모델)

잘 정의되고 확장성 좋은 이벤트 모델은 이벤트 구동 어플리케이션에 있어 필수적이다. Netty는 I/O에 초점을 맞춘 잘 정의된 이벤트 모델을 가지고 있다. 또한 여러분은 기존 코드에 영향을 주지 않고 여러분만의 이벤트 타입을 구현할 수 있다. 왜냐하면 각각의 이벤트 타입은 엄격한 타입 계층 구조에 의해 서로 구분되기 때문이다. 이는 다른 프레임워크에 대해 또 다른 차별 요소이다. 많은 NIO 프레임워크는 이벤트 모델을 가지고 있지 않거나 매우 제한적인 개념을 가지고 있다. 그렇기에 여러분이 새로운 사용자 정의 이벤트 타입을 추가하려고 시도할 때, 이들은 기존 코드에 영향을 주거나 아예 확장을 허용하지 않는다.

ChannelEvent는 **ChannelPipeline**에 있는 **ChannelHandler**들에 의해 처리된다. 파이프라인은 **Intercepting Filter** 패턴의 진화된 형태를 구현한다. 이것은 이벤트 처리 방법과 파이프라인에 있는 핸들러들이 상호 작용하는 법에 대해 사용자로 하여금 완전히 제어할 수 있게 해준다. 예를 들면, 데이터가 소켓으로부터 읽혀질 때 무엇을 수행할

지 정의할 수 있다.

1.2.4 Codec framework

훌륭한 네트워크 어플리케이션 프레임워크는 유지 보수가 뛰어난 사용자 코덱을 만들어주는, 확장성 좋고 재 사용 가능하며, 단위 테스트가 가능한 다중 계층 코덱 프레임워크를 제공해야 한다.
Netty는 여러분이 단순하든 복잡하든, 바이너리이든 아니든 간에 상관 없이 여러분이 프로토콜 코덱을 작성할 때 마주치게 될 대부분의 이슈를 해결할 수 있도록, Netty의 핵심 기능을 기반으로 다수의 기본적인 코덱과 고급 코덱을 제공한다.

1.2.5 SSL / TLS Support

기존 블록킹 I/O와 달리, NIO에서 SSL을 지원하는 것은 쉬운 일이 아니다. 데이터를 암호화 하거나 복호화 하기 위해 단순히 스트림을 감쌀 수는 없다. 대신 javax.net.ssl.SSLEngine을 사용해야만 한다. SSLEngine은 SSL처럼 매우 복잡한 상태 머신이다. 여러분은 암호 모음, 암호화 키 협상(혹은 재협상), 인증서 교환 및 유효성 검사와 같은 모든 가능한 상태들을 관리해야 한다. 더욱이, SSLEngine은 일반적인 기대와 달리 스레드에 안전하지도 않다.

Netty에서 **SSLHandler**는 **SSLEngine**의 모든 세부적인 사항들과 곤란한 점들을 처리한다. 여러분이 할 일은 **SslHandler**를 설정하고 **ChannelPipeline**에 끼워 넣는 것이다. 이는 또한 여러분이 **StartTLS**과 같은 고급 기능들을 매우 쉽게 구현할 수 있게 해준다.

- **SSL**

  -, SSL은 TCP/IP를 사용하는 두 개의 통신 애플리케이션 간 프라이버시와 무결성을 제공하는 프로토콜이다. 클라이언트와 서버를 오고 가는 데이터는 시메트릭 알고리즘을 사용하여 암호화 된다..

- **STARTTLS**

  -. **STARTTLS**는 SSL을 사용해서 SMTP 프로토콜 연결을 암호화 하는 것

1.2.6 HTTP

Netty는 HTTP 메시지가 하위 레벨에서 어떻게 교환되는지에 대해 Netty가 완전히 제어할 수 있게 해준다. 이는 기본적으로 HTTP 코덱과 HTTP 메시지 클래스들의 조합이므로, 강제된 스레드 모델과 같은 제약이 없다. 즉, 여러분은 자신이 원하는 방식 그대로 동작하는 자신만의 HTTP 클라이언트나 서버를 작성할 수 있다. 여러분은 스레드 모델, 커넥션 생명 주기, 덩어리 인코딩 및 HTTP 명세에서 여러분에게 허용한 수많은 것들을 완전히 제어할 수 있다.

Netty는 개인화 하는 능력이 매우 뛰어나기 때문에(high customizable nature), 여러분은 다음처럼 매우 효율적

인 HTTP 서버를 작성할 수 있다.

● 영속적인 커넥션과 서버 푸시 기술을 필요로 하는 채팅 서버 (예: Comet).
● 전체 미디어가 스트리밍 될 때까지 커넥션을 지속적으로 열어 두어야 하는 미디어 스트리밍 서버 (예: 2시간짜리 영화).
● 메모리 압박 없이 덩치 큰 파일 업로드를 허용하는 파일 서버 (예: 요청당 1GB 업로드)
● 수많은 제 3업체 웹 서비스에 비 동기로 연결하는 확장성 좋은 클라이언트.

1.2.7 Google Protocol Buffer Integration

-.정의 파일에서 고성능 고효율의 코드를 생성한다.
Encoder/Decoder 구현이 필요없으며 잘 작성된 정의파일을 통해 하위 호환성을 확보할 수 있으며 Java, C++, Phython등 대부분의 언어에서 사용이 가능하다.
Netty의 경우에는

- Encoder (ProtobufEncoder 와 LengthFiledPrepender)
- Decoder (ProtobufDecoder 와 LengthFiledBasedFrameDecoder)

로 간단히 통합이 가능하다.

- **Google Protocol Buffers**

-. 프로토콜 버퍼는 구조화된 데이터(XML보다 더 작고, 빠르고, 간단함)를 직렬화하는, Google의 언어 중립적이고 플랫폼 중립적인 확장 가능한 메커니즘이다. 데이터를 구조화하는 방식을 정의해두면 특별하게 생성된 소스 코드를 사용하여 구조화된 데이터를 여러 가지 데이터 스트림 간에 작성하고 읽을 수 있으며, Java, C++ 또는 Python과 같은 여러 가지 언어를 사용할 수 있다.

1.3 Netty 구성요소

1.3.1 Bootstrap

-. Channel을 초기화해주는 Helper Class이다. Common data structure(공통의 데이터 구조)를 사용하여 Channel과 그 자식 Channels들을 초기화해준다.

이름	설명
ClientBootstrap	새로운 Client-side Channel를 생성해주고 새로운연결을 시도하는 Helper Class이다. Channel을 설정해주는 Options을 사용할수 있다.(ChannelConfig 참조)
ServerBootstrap	새로운 Server-side Channel를 생성해주고 들어오는 연결시도를 받아주는 helper

	class이다. ServerBootstrap의 부모채널은 bind()나 bind(SocketAddress)를 통하여 bootstrap의 ChannelFactory에 의해서 자식채널을 생성한다.
	즉 부모채널은 들어오는 연결을 받아주는 Server이고, 성공적으로 bound가되어진 Clinet연결은 그 부모의 자식채널이된다.
	또한 Options을 사용하여 부모채널과 그 자식채널들의 config를 적용할 수 있다.
	// Options for a parent channel
	b.setOption("localAddress", new InetSocketAddress(8080));
	b.setOption("reuseAddress", true);
	
	// Options for its children
	b.setOption("child.tcpNoDelay", true);
	b.setOption("child.receiveBufferSize", 1048576);

ConnectionlessBootstrap	UDP/IP와 같은 Connectionless 전송을 위한 새로운 server-side을 생성해준다. Options을 사용하여 채널을 설정해줄수있다.

1.3.2 Channel

-. Read, write, connect, bind와 같은 I/O Operations들을 할수 있는 component들이나 network socket들의 복합이다.

1) 채널이 재공하는 것

-. 현재의 채널상태

-. 채널의 설정된 파라미터값

-. 채널이 지원하는 I/O Operation

-. Channel에 관련된 모든 I/O 요청이나 Event을 관장하는 ChannelPipeline

-. 모든 채널의 Operation은 비동기이다. 이것은 어떠한 I/O요청도 요청의 끝에 바로 결과값이 돌아오는 것을 보장하지 않는다는 것을 의미한다. 그 대신 요청한 I/O Operation이 succeeded, failed, canceled이 되었을 때 알림을 주는 ChannelFuture를 받을수 있다.

현재 채널의 상태확인

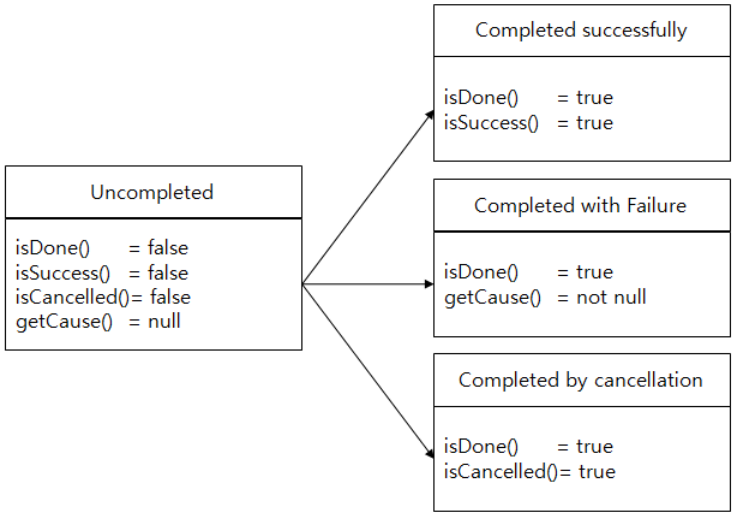
```
@Override
public void handleUpstream(
    ChannelHandlerContext ctx, ChannelEvent e) throws Exception {
        System.out.println("Channel :"+e.getChannel());
        System.out.println("isBound() :"+e.getChannel().isBound());
        System.out.println("isConnected() :"+e.getChannel().isConnected());
        System.out.println("isOpen() :"+e.getChannel().isOpen());
```

```
            System.out.println("isReadable() :"+e.getChannel().isReadable());
            System.out.println("isWritable() :"+e.getChannel().isWritable());
    }
}
```

- 출력정보

```
Channel : [id: 0x0053c015, /127.0.0.1:22466 => /127.0.0.1:10000]
isBound() : true
isConnected() : true
isOpen() : true
isReadable() : true
isWritable() : true
```

1.3.3 ChannelFuture



-. I/O Operation이 시작되었을때 새로운 ChannelFuture 객체가 시작된다. 새로운 ChannelFuture는 처음에는 complete되지 않으며 succeeded, failed, cancelled중 어느것도 아니다. 왜냐하면 I/O Operation이 아직 끝나지 않았기때문이다. 만약 I/O Operation이 succeeded, failed, cancelled 중 어느것으로 끝났다면 ChannelFuture는 Completed하고 더욱 구체적인 정보를 넘겨준다.

또한 ChannelFutureListener를 통해서 I/O Operation이 완료되었을대 알림을 받을수있다.

등록된 handler 파일안에서 설정이 가능

@Override

```
public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) {
    if (e.getMessage() instanceof GoodByeMessage) {
        ChannelFuture future = e.getChannel().close();
        future.addListener(new ChannelFutureListener() {
            public void operationComplete(ChannelFuture future) {

                // operation

            }
        });
    }
}
```

1.3.4 ChannelConfig

이름	내용
bufferFactory setBufferFactory(ChannelBufferFactory)	새로운 channelBuffer를 생성하여 사용할 때 channelbufferFactory를 설정한다.
connectTimeoutMillis setConnectTimeoutMillis(int)	Connection Timeout 설정
pipelineFactory setPipelineFactory(ChannelPipelineFactory)	새로운 child channel를 사용할 때 ChannelPipeline를 설정한다.

1.3.5 SocketChannelConfig

-. Netty에서는 네트워크 소켓의 설정을 필요에 따라서 설정해줄수있다.

이름	내용
keepAlive setKeepAlive(boolean)	KEEPALIVE 옵션을 설정해두면 TCP 프로토콜 수준에서 주기적으로 끊어진 연결을 감지하여 불필요하게 열린 소켓을 닫을 수 있게 된다.
reuseAddress setReuseAddress(boolean)	이 옵션을 사용하면 커널은 이미 특정 프로세스가 특정 소켓을 bind하고 있더라도, 프로세스가 그것을 무시하고 다시 bind할 수 있도록 해준다.
soLinger setSoLinger(int)	TCP에서 적용되는 옵션으로, 소켓 종료 시 버퍼의 처리 방법에 대해 정의하는 것이지만, 일반적으로는 close 함수의 행동에 대해 정의한다고 생각하면 된다. TCP에서 close는 예외적으로 4-way handshake를 쓰는데 만약 송수신 버퍼에 데이터가 남아있다면 남아있는 데이터를 전송하고 난 뒤 서버는 그 전송한 데이터에 대한

	ACK를 기다리게 된다. 즉 클라이언트 쪽에서 이 데이터를 다 처리하고 ack를 보내주기 전에 서버쪽에서는 해당 포트를 사용할 수가 없게되는 것이다. 이러한 문제를 해결하기 위해 SO_LINGER 옵션을 사용할수 있다.
tcpNoDelay setTcpNoDelay(boolean)	Nagle 알고리즘은 아주 간단하다. 한마디로 정의하면 "기존에 전송한 패킷이 있을 경우그 패킷에 대한 ACK 를 받아야만 다음 전송을 진행하는 알고리즘 이다. tcpNoDelay 옵션은 TRUE 로 설정되는 경우, Nagle 알고리즘을 적용하지 않겠다는 의미이고 FALSE 인 경우 Nagle 알고리즘을 적용한다는 의미가 된다.
receiveBufferSize setReceiveBufferSize(int)	커널의 수신버퍼의 사이즈를 바꾸는 옵션이다.
sendBufferSize setSendBufferSize(int)	커널의 송신버퍼의 사이즈를 바꾸는 옵션이다.
trafficClass setTrafficClass(int)	이 소켓으로부터 패킷전송을 위해 IP 헤더안에 TrafficClass나 type-of-service를 설정하는 옵션이다.

1.3.6 SimpleChannelHandler

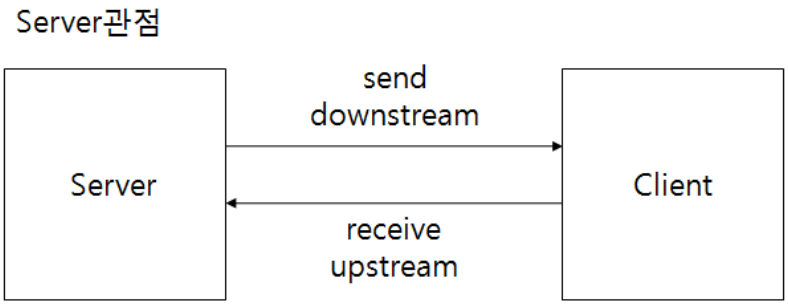
-. ChannelHandler은 각각의 event를 위해서 각각의 handler메소드를 지원한다. 만약 당신이 upstream와 downstream이벤트중 하나의 이벤트만을 intercept하고싶을때는 SimpleChannelUpstreamHandler 나 SimpleChannelDownstreamHandler를 사용한다.

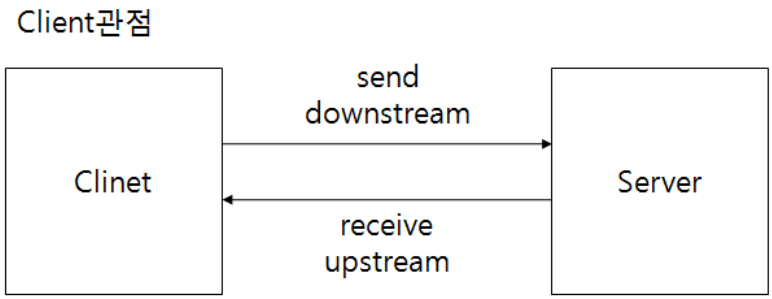
```
public class MyChannelHandler extends SimpleChannelHandler {

    @Override
    public void handleUpstream(ChannelHandlerContext ctx, ChannelEvent e) throws Exception {

        // Log all channel state changes.
        if (e instanceof ChannelStateEvent) {
            logger.info("Channel state changed: " + e);
        }

        super.handleUpstream(ctx, e);
    }

    @Override
    public void handleDownstream(ChannelHandlerContext ctx, ChannelEvent e) throws Exception {
```

```
            // Log all channel state changes.
            if (e instanceof MessageEvent) {
                logger.info("Writing:: " + e);
            }

            super.handleDownstream(ctx, e);
        }
    }
```

1.3.7 ChannelEvent

-. Channel에 관련된 I/O Event나 I/O Operation이다. ChannelEvent는 ChannelPipeline의 ChannelHandler에 의해서 조정된다. ChannelEvent는 기본적으로 upstream과 downstream으로 설명가능하다.





이것은 기본적으로 Send일경우는 downStream이 일어나며 Receive의 경우 upstream이 일어나는것으로 설명할수 있다.

Upstream Event		
Name	**Event Type and condition**	**Meaning**
messageReceived	MessageEvent	원격연결로부터 받은 메시지객체
exceptionCaught	ExceptionEvent	I/O Thread나 ChannelHandler로부터 받은 Exception

channelOpen	ChannelStateEvent (state = OPEN, value = true)	채널은 Open하지만 not bound, not connect
channelClosed	ChannelStateEvent (state = OPEN, value = false)	채널이 closed그리고 관련된 모든 리소스가 해제된상태
channelBound	ChannelStateEvent (state = BOUND, value = SocketAddress)	채널이 open그리고 local address로 bound된상태 하지만 not connect
channelUnbound	ChannelStateEvent (state = BOUND, value = null)	채널이 현재의 local address로부터 unbound된상태
channelConnected	ChannelStateEvent (state = CONNECTED, value = SocketAddress)	채널이 open, local address로 bound 그리고 connect된상태
writeComplete	WriteCompletionEven	어떤 것이 원격연결점으로 writed된상태
channelDisconnected	ChannelStateEvent (state = CONNECTED, value = null)	채널이 원격연결점으로부터 disconnected된상태
channelInterestChanged	ChannelStateEvent (state = INTEREST_OPS, no value)	채널의 interestOps가 바낀상태

Downstream Event		
Name	**Event Type and condition**	**Meaning**
write	MessageEvent	Channel로 message를 보낸다.
bind	ChannelStateEvent (state = BOUND, value = SocketAddress)	특정한 local address로 bind한다.
unbind	ChannelStateEvent (state = BOUND, value = null)	현재의 local address로부터 unbind한다.
connect	ChannelStateEvent (state = CONNECTED, value = SocketAddress)	특정한 원격의 address로 connect한다.
disconnect	ChannelStateEvent (state = CONNECTED, value = null)	현재의 원격의 address로부터 Disconnect한다.
close	ChannelStateEvent (state = OPEN, value = false)	Channel을 close한다.

- 자식 Channel를 가진 부모 Channel만이 사용할수 있는 event type

Name	**Event Type and condition**	**Meaning**
childChannelOpen	ChildChannelStateEvent (childChannel.isOpen() = true)	자식채널이 Open

childChannelClosed	ChildChannelStateEvent (childChannel.isOpen() = false)	자식채널이 Close

- 중요사항

-. **위의 표에 없는 다른 Event Type나 conditions(상태들)은 무시되거나 버려진다.**

- 이벤트 흐름

연결시도 flow

2010. 11. 18 오전 11:09:39 org.jboss.netty.example.telnet.TelnetServerHandler handleUpstream

정보: [id: 0x0053c015, /127.0.0.1:20400 => /127.0.0.1:10000] **OPEN**

2010. 11. 18 오전 11:09:39 org.jboss.netty.example.telnet.TelnetServerHandler handleUpstream

정보: [id: 0x0053c015, /127.0.0.1:20400 => /127.0.0.1:10000] **BOUND**: /127.0.0.1:10000

2010. 11. 18 오전 11:09:39 org.jboss.netty.example.telnet.TelnetServerHandler handleUpstream

정보: [id: 0x0053c015, /127.0.0.1:20400 => /127.0.0.1:10000] **CONNECTED**: /127.0.0.1:20400

연결종료 flow

2010. 11. 18 오전 11:14:37 org.jboss.netty.example.telnet.TelnetServerHandler handleUpstream

정보: [id: 0x0053c015, /127.0.0.1:20400 :> /127.0.0.1:10000] **DISCONNECTED**

2010. 11. 18 오전 11:14:37 org.jboss.netty.example.telnet.TelnetServerHandler handleUpstream

정보: [id: 0x0053c015, /127.0.0.1:20400 :> /127.0.0.1:10000] **UNBOUND**

2010. 11. 18 오전 11:14:37 org.jboss.netty.example.telnet.TelnetServerHandler handleUpstream

정보: [id: 0x0053c015, /127.0.0.1:20400 :> /127.0.0.1:10000] **CLOSED**

WriteComplete

2010. 11. 18 오전 11:09:39 org.jboss.netty.example.telnet.TelnetServerHandler handleUpstream

정보: [id: 0x0053c015, /127.0.0.1:20400 => /127.0.0.1:10000] WRITTEN_AMOUNT: 70

2010. 11. 18 오전 11:09:55 org.jboss.netty.example.telnet.TelnetServerHandler handleUpstream

정보: [id: 0x0053c015, /127.0.0.1:20400 => /127.0.0.1:10000] WRITTEN_AMOUNT: 19

2010. 11. 18 오전 11:10:06 org.jboss.netty.example.telnet.TelnetServerHandler handleUpstream

정보: [id: 0x0053c015, /127.0.0.1:20400 => /127.0.0.1:10000] WRITTEN_AMOUNT: 20

WRITTEN_AMOUNT : 전송한 byte수

//연결시 메세지를 보내준다. write 2번을 사용했지만. WriteComplete이벤트의 발생수는 1번

```
@Override
public void channelConnected(ChannelHandlerContext ctx, ChannelStateEvent e) throws Exception {
// Send greeting for a new connection.
e.getChannel().write("Welcome to " + InetAddress.getLocalHost().getHostName() + "!₩r₩n");
e.getChannel().write("It is " + new Date() + " now.₩r₩n");
```

```
}
2010. 11. 18 오전 11:09:39 org.jboss.netty.example.telnet.TelnetServerHandler handleUpstream
정보: [id: 0x0053c015, /127.0.0.1:20400 => /127.0.0.1:10000] WRITTEN_AMOUNT: 70
```

WriteComplete이벤트는 1번만 발생했다.

1.3.8 Buffer

-. 1.2.1 Rich Buffer Data Structure 에서도 언급했듯이. Netty는 NIO ByteBuffer 대신 자신만의 버퍼 API를 사용하여 바이트를 표현한다. 이러한 접근 방법은 ByteBuffer를 사용하는 것보다 훨씬 더 큰 장점을 가지고 있다. Netty의 새로운 버퍼 타입인 ChannelBuffer는 신장성, 투명한 Zero-Copy, Automatic Capacity Extension를 가지고있다.

1.3.9 ChannelBuffer

- 생성

-. org.jboss.netty.buffer.ChannelBuffers는 arrays, byte buffer, String으로 존재하는 copy와 wrapping이 가능한 새로운 공간이다. 이 ChannelBuffers의 생성자를 통해서 생성이 가능한 것이 ChannelBuffer이다. 아래의 방식으로 생성될수있다.

```
import static org.jboss.netty.buffer.ChannelBuffers.*;
//a new fixed-capacity heap buffer.
ChannelBuffer heapBuffer     = buffer(128);
//a new fixed-capacity direct buffer.
ChannelBuffer directBuffer   = directBuffer(256);
// a new dynamic-capacity heap buffer, whose capacity increases automatically as needed by a write operation.
ChannelBuffer dynamicBuffer = dynamicBuffer(512);
ChannelBuffer wrappedBuffer = wrappedBuffer(new byte[128], new byte[256]);
ChannelBuffer copiedBuffer = copiedBuffer(ByteBuffer.allocate(128));
```

-. HeapBuffer : JVM의 자바객체를 저장하는 공간인 Heap를 사용하는 버퍼

-. DirectBuffer : 일반메모리에 접근하여 저장하는 버퍼

- 기능

ChannelBuffer는 **순차적 읽기와 쓰기**를 지원하기위해서 두개의 포인터 값을 제공하고있습니다.

이는 프로그램에서 ChannelBuffer 객체를 출력해보면 알수 있다.

```
//동적버퍼 생성
```

ChannelBuffer header = ChannelBuffers.dynamicBuffer(12);

System.err.println(header);

//header ChannelBuffer 객체 출력

DynamicChannelBuffer(ridx=0, widx=0, cap=12)

이채널 버퍼의 종류인 DynamicChannelBuffer과 ridx(read Index), widx(write Index), cap(capacity:용량)

현재 상태를 알수 있다.

방금 막 생성한 상태로 ridx와 widx둘다 0인 상태이며, cap는 현재 시작시 선언해준 12를 나타내고 있는 것을 알수 있다.

header.**writeBytes**("Yuu_HeaVen ".getBytes());

ChannelBuffer 에 writeBytes를 입력 후

DynamicChannelBuffer(ridx=0, widx=12, cap=12)

Widx (12) 가 증가한 것을 확인할 수 있다.

또한

header.**writeByte**('2'); 를 추가했을 시에는

DynamicChannelBuffer(ridx=0, widx=13, cap=24)

으로 widx가 1, capacity가 12 증가한것을 확인 할 수 있다.

이부분에서 capacity가 동적으로 증가하는 것을 확인할 수 있다.

header.**readByte**(); 로 1자리를 읽었을경우

DynamicChannelBuffer(ridx=1, widx=13, cap=24)

Ridx가 1자리 증가한 것을 확인할 수 있다.

이렇게 Read와 Wirte의 인덱스를 각각 지원함으로써 flip()메소드를 사용할 필요가 없다.

또한 모든 버퍼에 2개의 **Marker**를 지원합니다. 하나는 readIndex를 위한것이며 다른하나는 writeIndex를 위한 것입니다. 이는 index를 reset을 했을시 reset지점을 미리 정해주기위해서 사용합니다.

markReaderIndex

이 마크는 이 버퍼안의 현재의 readIndex를 마크합니다. 이것은 **resetReaderIndex**()를 부르는것으로 인하여 mark한 지점으로 갈수 있습니다.

markWriterIndex

이 마크는 이 버퍼안의 현재의 writeIndex를 마크합니다. 이것은 **resetWriterIndex**()를 부르는것으로 인하여 mark한 지점으로 갈수 있습니다.

예제

```
ChannelBuffer header = ChannelBuffers.dynamicBuffer(12);

// ChannelBuffer에서 사용할 바이트배열
byte[] start = new byte[3];
byte[] message = new byte[12];
//readIndex를 Mark 한다.
header.markReaderIndex();
//Yuu read
header.writeBytes("Yuu".getBytes());
//WriteIndex을 Mark한다.
header.markWriterIndex();
System.err.println("111"+header);
header.readBytes(start);
System.err.println(new String(start));
```

- 출력값

```
111DynamicChannelBuffer(ridx=0, widx=3, cap=12)
Yuu
```

Yuu가 ChannelBuffer에 Write되었고 widx는 3으로 증가되었다.

현재 **markWriterIndex는 3이다.**

```
header.writeBytes("_HeaVen12".getBytes());
System.err.println("222"+header);
```

- 출력값

```
222DynamicChannelBuffer(ridx=3, widx=12, cap=12)
```

위에서 Yuu를 출력하기 위해서 ridx가 3으로 증가하였다. 또한 _HeaVen12 을 Write하여 widx가 12로 증가한 것을 볼수 있다.

```
header.resetReaderIndex();
System.err.println("333"+header);
```

- 출력값

```
333DynamicChannelBuffer(ridx=0, widx=12, cap=12)
```

header.resetReaderIndex();을 사용하여 ridx가 0으로 된 것을 확인 할 수 있다.

---

header.readBytes(message);

System.err.println(new String(message));

System.err.println("444"+header);

- 출력값

Yuu_HeaVen12

444DynamicChannelBuffer(ridx=12, widx=12, cap=12)

Ridx가 0인 상태에서 message byte배열을 이용하여 총 12의 자리를 읽어 출력하였다.

그 결과 현재 입력한 전체의 메시지가 출력되었으며

현재의 idx상황은 12/12/12가 되었다.

---

header.resetReaderIndex();

System.err.println("555"+header);

- 출력값

555DynamicChannelBuffer(ridx=0, widx=12, cap=12)

resetReadIndex()로 인하여 ridx가 0으로 되었다.

---

header.resetWriterIndex();

System.err.println("666"+header);

- 출력값

666DynamicChannelBuffer(ridx=12, widx=3, cap=12)

resetWriterIndex으로 인하여 현재의 widx의 값은 3으로 변했다.

중요 참고

현재 위의 resetWriterIndex를 사용에 대해서 알아보자.

현재 총데이타는 [Yuu_HeaVen12] 총 12자리가 들어가있다. 만약 resetWriterIndex, resetReaderIndex 를 사용해서 각각 DynamicChannelBuffer(ridx=0, widx=3, cap=12) 으로 인덱스의 위치가 변화되었다고 생각해보자.

만약 여기서 총 12자리의 byte배열로 데이터를 읽어온다면 인덱스 오류 에러가 발생한다.

이 이유는 현재의 widx가 3이기 때문이다. ChannelBuffer은 이 버퍼안에 최대 몇자리까지의 Data가 들어있는것과는 관계 없이 widx의 자리수까지의 data만을 읽어올수 있기 때문이다.

header.writeBytes("_Hell1234".getBytes());

System.err.println(header);

header.resetReaderIndex();

header.readBytes(message2);

System.err.println(new String(message2));

- 출력값

DynamicChannelBuffer(ridx=12, widx=12, cap=12)

Yuu_Hell1234

Widx가 3의 위치에서 총 9자리의 [_Hell1234]를 넣어주면 위와같이 기존의 값을 덮어쓰기가 일어나게된다.

- 또한 각각의 discardReadBytes() , clear(), byte array로의 전환, skipBytes()와 같은 메소드를 지원합니다.

1) discardReadBytes ()

현재까지 읽은 bytes를 버릴수 있다.

2) clear()

이 메소드를 ridx,widx를 0으로 초기화 시켜준다. 하지만 이것은 버퍼의 콘덴츠마저 삭제해주는 buffer.clear()과는 다르게 인덱스만을 초기화 시켜주는것이다. **본래의 버퍼 내용이 삭제가 되는 것이 아니니 주의하자.**

3) Array(), hasArray()

만약 channelBuffer이 byte array로 돌려질수있다면, 당신은 array()메소드를 통하여 직접 접근할수있다.또한 array()를 사용할수 있는 확인할수 있는 hasArray()메소드 또한 지원한다.

4) skipBytes()

이 메소드를 read Index를 필요한만큼 skip()하는 역할을 한다.

- 또한 버퍼의 전체 복사, 혹은 부분 복사를 지원한다.

1) Duplicate()

Mark와 index를 그대로 유지한 상태의 똑같은 채널버퍼로 복사한다. 이것은 buf.slice(0, buf.capacity())동일하다.

2) slice()

이 버퍼의 readable bytes를 채널버퍼로 리턴해주는 메소드이다. 이것은 buf.slice(buf.readerIndex(), buf.readableBytes())와 동일하다.

3) slice(int index, int length)

이 버퍼의 부분조각을 리턴한다.

1.3.10 WrappedChannelBuffer

-. WrappedChannelBuffer를 사용하여 자신만의 버퍼타입을 만들 수 있다.

Write

String line = in.readLine();

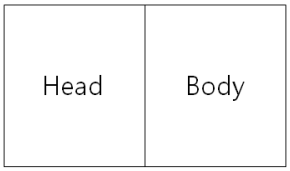
//동적버퍼 생성

```
ChannelBuffer header = ChannelBuffers.dynamicBuffer(12);
ChannelBuffer body = ChannelBuffers.dynamicBuffer(512);

//head, body 내용
header.writeBytes("Yuu_HeaVen   ".getBytes());
body.writeBytes(line.getBytes());

//wrappedBuffer 버퍼 생성
ChannelBuffer buffer = ChannelBuffers.wrappedBuffer(header, body);

//전송
lastWriteFuture = channel.write(buffer);
```

READ

```
//Receive Message를 위한 ChannelBuffer를 생성
ChannelBuffer request = (ChannelBuffer) e.getMessage();

//받은 메세지의 길이 측정
int totalLength = request.readableBytes();
//Head 길이는 고정
int headLength =  12;
//Body를 위한 message byte[] 선언
byte[] message = new byte[totalLength - headLength];

//Head 길이 만큼 readIndex skip
request.skipBytes(12);
//Body길이만큼 Message Read
request.readBytes(message);
//Body 출력
System.out.println(new String(message));
```

위의 예제소스는 WrappedChannelBuffer를 사용법에 대한 간략한 소개를 보여준다.
위의 WrappedChannelBuffer는 기본적으로 아래와 같은 구조를 가지고있다.



Send에서 Head의 길이는 12이고 body의 길이는 512의 동적버퍼를 각각 선언하여 메시지를 담고 WrappedChannelBuffer를 생성하여 전송하여 주었고, 이것을 다시 ChannelBuffer로 받아 Head의 길이만큼은 생략한채 Body의 내용을 출력해주는 예제이다.

.

1.3.11 CODEC

-. Netty는 네트워크 통신에서 필요한 CODEC을 지원해주며 이는 개발자가 decoder와 encoder을 구현해야할 필요를 줄여준다.

구분	내용
codec.base64	기본적으로 64진수(2^6,6비트)로 모든 정보를 표현하고자 하는데 있다. MIME에 주로 사용하고 Web인증중 하나인 기본인증에도 사용된다. 2진데이터를 ASCII형태의 텍스트로 표현가능하다.
codec.compression	Zlib나 gzip과 같은 압축포멧의 압축과 압축을 푸는 디코더와 인코더를 제공한다.
codec.embedder	encoder 나 decoder (codec)를 감싸는 헬퍼로 유닛테스트나 높은 레벨의 코덱이 실제 하는 것 없이 사용할수 있다.
codec.http	HTTP를 위한 인코더, 디코더 그리고 그들과 관련있는 메시지 타입들
codec.http.websocket	순수 웹 환경에서 실시간 양방향 통신을 위한 스펙이 바로 '**웹 소켓(Web Socket)**' Http.websocket를 위한 인코더 디코더 그리고 그들과 관련있는 메시지타입들
codec.oneone	다른 object나 그반대로 한 object를 변형시키는 인코더와 디코더의 수행을 돕는 매우 간단한 추상클래스
codec.protobuf	ChannelBuffer이나 그반대로 Google Protocol Buffer로 변형시켜주는 인코더와 디코더
codec.replay	FrameDecoder의 전문적인 변화는 blocking I/O문법에서 non-blocking I/O디코더의 실행을 가능케해준다.
codec.rtsp	HTTP codec를 기반으로한 RTSP표현식 **Real Time Streaming Protocol** RTSP는 리얼 플레이어로 유명한 리얼 네트워크사와 넷스케이프, BMm 콜롬비아 대학이 공동으로 개발 참여하여 표준화한 것, 클라이언트 서버 기반으로 동작하며, 실시간으로 스트리밍을 제공하는 프로토콜입니다 주요사용처는 인터넷을 기반으로 해서 멀티미디어 정보를 전달하는데 많이 사용됩니다.
codec.serialization	Bytebyffer나 그반대로 직렬화가능한 object를 변형하는 것을 실행하는 인코더, 디

	코더, 그리고 그들의 호환성있는 스트림 Serialization : Object를 byte배열의 스트림을 네트워크 저장매체인 파일이나 데이터베이스로 저장하기위해서 쓰인다.
codec.string	channelbuffer이나 그반대로 String를 변형시키는 인코더 디코더
codec.frame	확장할수 있는 decoder와 패킷단편화을 나누고 TCP/IP와 같은 스트림기반의 전송에서 재조립 문제를 발견하는 것을 공통적으로 시행한다.

2 예제

2.1 기본예제

-. Netty의 예제를 작성하며 기본사용법을 숙지한다.

- **Bind**

ServerBootstrap을 이용하여 서버쪽의 채널을 생성하고 들어오는 연결을
받아들이는 ServerSocketChannel를 생성한다.

```
ServerBootstrap bootstrap = new ServerBootstrap(
                    new NioServerSocketChannelFactory(
                                Executors.newCachedThreadPool(),
                                Executors.newCachedThreadPool()));
bootstrap.bind(new InetSocketAddress(10000));
```

- **Connect**

ClientBootstrap를 이용하여 서버에 접속한다.

```
ClientBootstrap bootstrap = new ClientBootstrap(
            new NioClientSocketChannelFactory(
                        Executors.newCachedThreadPool(),
                        Executors.newCachedThreadPool()));
ChannelFuture future = bootstrap.connect(new InetSocketAddress(host, port));
```

- **Write**

현재의 채널의 write를 통해서 자바객체를 원격연결점으로 보낼수있다.
ChannelFuture는 Channel동작의 현재의 상태값을 알수있다.

```
ChannelFuture WriteFuture = null;
```

WriteFuture = channel.**write**(line);

- **Receive**

SimpleChannelHandler나 SimpleChannelUpstreamHandler를 통해서 UpstreamEvent의 messageReceived롤 통해서 메세지를 받을 수 있다.

Handler.java의 messageReceived

```
//메세지를 받았을시
@Override
public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) {

        String request = (String) e.getMessage();
        System.out.println("Receive Message : " +request);
}
```

- 연결종료

Client의 연결을 종료하가위해서는 아래의 단계가 필요하다.

```
// 메세지가 flushed되기를 대기한다.
ChannelFuture.awaitUninterruptibly();

// Netty는 모든 I/O가 비동기이므로 종료시 operation를 반드시 닫아야한다.
channel.close().awaitUninterruptibly();

//ChannelFactory의 자원해제
bootstrap.releaseExternalResources();
```

2.2 Telnet

-. Telnet 서버를 통하여 Netty의 전체개념을 숙지한다.

```
public class TelnetServer {
    public static void main(String[] args) throws Exception {
        // Configure the server.
```

```
        ServerBootstrap bootstrap = new ServerBootstrap(
                        new NioServerSocketChannelFactory(
                        Executors.newCachedThreadPool(),
                        Executors.newCachedThreadPool()));

        TelnetServerHandler handler = new TelnetServerHandler();
        bootstrap.setPipelineFactory(new TelnetPipelineFactory(handler));

        // Bind and start to accept incoming connections.
        bootstrap.bind(new InetSocketAddress(8080));
    }
}
```

Bootstrap	채널을 설정하는 도움클래스로 이 클래스는 채널을 설정하고 그들의 자식 채널들이 공통의 데이타구조를 사용하도록 제공한다.
ServerBootstrap	새로운 서버쪽의 Channel를 생성하고 들어오는 연결을 받아들인다.
NioServerSocketChannelFactory	서버쪽의 NIO기반의 ServerSocketChannel를 생성하다. non-blocking I/O 모드를 활용하여 많은수의 동시다발적인 연결을 효율적으로 받는 NIO를 도입한다

```
// Channel당 Handler를 생성해주기 위해서 추가하는 Factory

public class TelnetPipelineFactory implements
        ChannelPipelineFactory {

    private final ChannelHandler handler;
    public TelnetPipelineFactory(ChannelHandler handler) {
        this.handler = handler;
    }
    public ChannelPipeline getPipeline() throws Exception {
        // Create a default pipeline implementation.
        ChannelPipeline pipeline = pipeline();

        // Add the text line codec combination first,
        pipeline.addLast("framer", new DelimiterBasedFrameDecoder(
```

```
                8192, Delimiters.lineDelimiter()));
        pipeline.addLast("decoder", new StringDecoder());
        pipeline.addLast("encoder", new StringEncoder());

        // and then business logic.
        pipeline.addLast("handler", handler);

        return pipeline;
    }
}
```

DelimiterBasedFrameDecoder	Decoder는 받은 ChannelBuffer을 NUL 이나 newline characters(₩n₩r) 과 같은 구분자로 구분한다.

```
public class TelnetServerHandler extends SimpleChannelUpstreamHandler {

    private static final Logger logger = Logger.getLogger(
            TelnetServerHandler.class.getName());

  //handleUpstream : 명시된 이벤트를 처리하다.
    @Override
    public void handleUpstream(
            ChannelHandlerContext ctx, ChannelEvent e) throws Exception {
        if (e instanceof ChannelStateEvent) {
            logger.info(e.toString());
        }
        super.handleUpstream(ctx, e);
    }

    //연결시 메세지를 보내준다.
    @Override
    public void channelConnected(
            ChannelHandlerContext ctx, ChannelStateEvent e) throws Exception {
        // Send greeting for a new connection.
        e.getChannel().write(
                "Welcome to " + InetAddress.getLocalHost().getHostName() + "!₩r₩n");
```

```
        e.getChannel().write("It is " + new Date() + " now.₩r₩n");
    }

    //메세지를 받았을시
    @Override
    public void messageReceived(
            ChannelHandlerContext ctx, MessageEvent e) {

        // Cast to a String first.
        // We know it is a String because we put some codec in TelnetPipelineFactory.
        String request = (String) e.getMessage();

        // Generate and write a response.
        String response;
        boolean close = false;
        if (request.length() == 0) {
            response = "Please type something.₩r₩n";
        } else if (request.toLowerCase().equals("bye")) {
            response = "Have a good day!₩r₩n";
            close = true;
        } else {
            response = "Did you say '" + request + "'?₩r₩n";
        }

        // We do not need to write a ChannelBuffer here.
        // We know the encoder inserted at TelnetPipelineFactory will do the conversion.
        ChannelFuture future = e.getChannel().write(response);

        // Close the connection after sending 'Have a good day!'
        // if the client has sent 'bye'.
        if (close) {
            future.addListener(ChannelFutureListener.CLOSE);
        }
    }

    //exception
```

```
    @Override
    public void exceptionCaught(
            ChannelHandlerContext ctx, ExceptionEvent e) {
        logger.log(
                Level.WARNING,
                "Unexpected exception from downstream.",
                e.getCause());
        e.getChannel().close();
    }
}
```

```
public class TelnetClient {

    public static void main(String[] args) throws Exception {
        // Parse options.
        String host = "127.0.0.1";
        int port = 8080;
        // Configure the client.
        ClientBootstrap bootstrap = new ClientBootstrap(
                new NioClientSocketChannelFactory(
                        Executors.newCachedThreadPool(),
                        Executors.newCachedThreadPool()));

        TelnetClientHandler handler = new TelnetClientHandler();
        bootstrap.setPipelineFactory(new TelnetPipelineFactory(handler));

        // Start the connection attempt.
        ChannelFuture future = bootstrap.connect(new InetSocketAddress(host, port));

        // 커넥션 시도의 성공, 실패여부를 판단하기위하여 대기한다.
        Channel channel = future.awaitUninterruptibly().getChannel();
        if (!future.isSuccess()) {
            future.getCause().printStackTrace();
            //ChannelFactory의 자원해제
            bootstrap.releaseExternalResources();
            return;
```

```
}

// Read commands from the stdin.
ChannelFuture lastWriteFuture = null;
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
for (;;) {
    String line = in.readLine();
    if (line == null) {
        break;
    }

    // Sends the received line to the server.
    lastWriteFuture = channel.write(line + '₩n');

    // If user typed the 'bye' command, wait until the server closes
    // the connection.
    if (line.toLowerCase().equals("bye")) {
        channel.getCloseFuture().awaitUninterruptibly();
        break;
    }
}

// Wait until all messages are flushed before closing the channel.
// 메세지가 flushed되기를 대기한다.
if (lastWriteFuture != null) {
    lastWriteFuture.awaitUninterruptibly();
}

// Close the connection.  Make sure the close operation ends because
// all I/O operations are asynchronous in Netty.
// Netty는 모든 I/O가 비동기이므로 종료시 operation를 반드시 닫아야한다.
channel.close().awaitUninterruptibly();

// Shut down all thread pools to exit.
//ChannelFactory의 자원해제
bootstrap.releaseExternalResources();
```

```
    }
}
```

Bootstrap	채널을 설정하는 도움클래스로 이 클래스는 채널을 설정하고 그들의 자식 채널들이 공통의 데이타구조를 사용하도록 제공한다.
ClientBootstrap	이 클래스는 새로운 client-side Channel을 만들고 커넥션을 시도한다.
NioClientSocketChannelFactory	NIO를 기반으로한 클라이언트 SocketChannel를 생성해주며 이것은 하나의 보스 스레드와 하나혹은 다수의 워크 스레드를 가진다.
Boss thread	요청에 의하여 하나의 연결을 시도하며 연결이 성공했을시 NioClientSocketChannelFactory를 관리하는 하나의 워크스레드의 연결된 채널이된다.
Worker threads	non-blocking 모드에서 하나혹은 다수의 채널이 non-blocking read/write를 한다.
ChannelFuture	비동기 채널 I/O동작의 결과이다. Netty에서의 모든 I/O동작은 비동기이다. 이것은 어떤 I/O요청이 요청이 끝나는 즉시 완료되는것을 보장하지 않는다. ChannelFuture는 I/O동작상태의 결과 에관한 정보를 리턴해준다. ChannelFuture는 완료되엇던 완료되지 않았던 I/O operation이 시작될때 새로운 Future를 생성해주고 I/O operation이 성공, 실패, 취소가 되었을시. 각각 상황의 보다 자세한 상황정보를 만들어준다. 또한 I/O operation이 끝났을때 통지해주는 ChannelFutureListener를 등록하는것을 허락한다.
channel	read, write, connect, bind와 같은 I/O operations이 가능한 네트워크 소켓이나 컴포넌트의 집합이다. 채널은 유저에게 채널의 현상태를 제공하고, 채널의 파라미터를 설정할수 있으며 채널이 지원하는 I/O operation과 ChannelPipeline로 채널에 대한 모든 I/O 이벤트와 요청을 조정할수있다.

3 참고개념

3.1 OIO/NIO

OLD Blocking I/O	NIO New Non-blocking I/O
-.1 boss thread -.1worker thread per connection -.장점 1) 단순한다. 2) 적은수의 커넥션에서 고성능	-. 1 boss thread - 'N' worker thread for all connections -.장점 1) 많은수의 커넥션에서 고효율 고성능 2) 많은수의 커넥션에서 디버그 쉬움

3) 이해하기 쉬운 직선적 코드 진행 -.단점 1) 많은 수의 커넥션에서 자원소모심함 2) 많은 수의 커넥션에서 디버그 어려움	-.단점 1) 복잡한 스레드모델 2) 이해하기 어려운 State machine기반 3) 벤더별 NIO 구현 차이

3.2 DOS및 과부하 방어

1) Message flooding
-. (Ordered)MemoryAwareThreadPoolExecutor
-. Write buffer high low water mark
2) 접속후 요청 응답없이 자원소모의 경우
-. ReadTimeoutHandler
-. WriteTimeoutHandler
-.LdleStateHandler
3) 적극적 방어를 위해서는 비지니스 로직에 맞게 최적화
-. '의심스러운' 커넥션 제거
Flooding 기준정의,
엄격한 전문형식 State machine 상태 유효성 검사
-.긴 전문은 Chunk로 쪼개 처리

3.2.1 org.jboss.netty.handler.timeout과 Handler 예제

-. org.jboss.netty.handler.timeout는 Timer를 사용하여 Read와 Write의 Timeout과 idle connection의 알림을 위해서 추가되었다.

Handler Name	Meaning
IdleStateHandler	한 채널에서 read나 write, 혹은 둘다 특정기간동안 수행되지 않을때 IdleStateEvent를 발생시킨다. 인자값으로 ReadIdleTime, WriteIdelTime, AllIdelTim을 순서대로 입력받고 단위는 Second로 만약 0으로 설정을 했을경우 해당 Idle동작은 Disable된다.
ReadTimeoutHandler	특정하게 정해진 기간내에 Data를 읽을수 없을때 ReadTimeoutException을 발생시킨다.
WriteTimeoutHandler	특정하게 정해진 기간내에 Data를 읽을수 없을때 WriteTimeoutException을 발생시킨다.

IdleState와 TimeExceprion	
Name	**Meaning**
IdleState	채널의 Idle 상태를 표시해주는 Enum이다. ALL_IDLE, READER_IDLE, WRITER_IDLE 3개자의 상태값이 존재한다.

TimeouteException	특정하게 정해진기간내에 읽기나 쓰기 각각 발생하지 않았을 경우 발생한다.
ReadTimeouteException	특정하게 정해진기간내에 Data를 read할수 없을경우 ReadTimeoutHandler에 의해서 발생된다.
WriteTimeouteException	특정하게 정해진기간내에 Data를 write할수 없을경우 WriteTimeoutHandler에 의해서 발생된다

- 예제

-. Telnet 예제를 적용해서 테스트해본다.

TelnetPipelineFactory.java

```
private final ChannelHandler handler;
private final Timer timer;

public ChannelPipeline getPipeline() throws Exception {
.....
ChannelPipeline pipeline = pipeline();
pipeline.addLast("decoder", new StringDecoder());
pipeline.addLast("encoder", new StringEncoder());
pipeline.addLast("handler", handler);
pipeline.addLast("WriteTimeOuthandler", new JungTimeoutHandler(timer, 5, 5, 5));
```

JungTimeoutHandler.java

```
import org.jboss.netty.handler.timeout.IdleState;
import org.jboss.netty.handler.timeout.IdleStateHandler;

public class JungTimeoutHandler extends IdleStateHandler{
        public JungTimeoutHandler(Timer timer, int readerIdleTimeSeconds,
                        int writerIdleTimeSeconds, int allIdleTimeSeconds) {
                super(timer, readerIdleTimeSeconds, writerIdleTimeSeconds, allIdleTimeSeconds);
        }

        protected void channelIdle(ChannelHandlerContext ctx, IdleState state, long lastActivityTimeMillis) {

                        System.out.println("IdleState..... : "+state);
```

```
        }
]
```

테스트결과

IdleState..... : WRITER_IDLE

IdleState..... : READER_IDLE

IdleState..... : ALL_IDLE

특정 상태를 지정할 경우 아래와 같이 하거나. IdleHandler의 시간설정시 0으로 설정하면 Disable된다.

```
protected void channelIdle(ChannelHandlerContext ctx, IdleState state, long lastActivityTimeMillis) {
        if(state.equals(IdleState.WRITER_IDLE)){
                System.out.println("IdleState..... : "+state);
        }
}
```

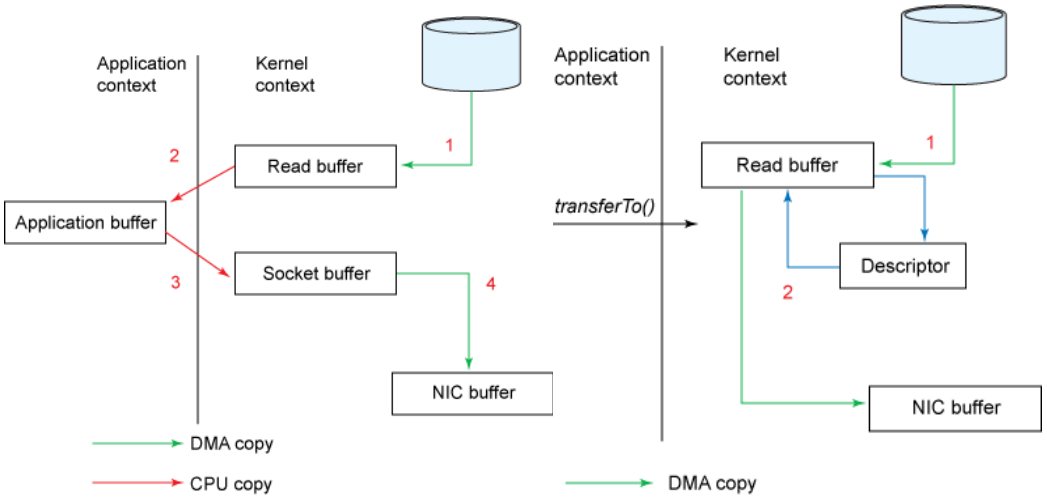
3.3 filp() 메소드

flip() 메소드를 실행하면 버퍼의 처음부터 데이터를 읽어올 수 있게 되며, 또한 채널을 통해서 버퍼에 있는 데이터를 모두 출력할 수 있게 된다.

- Netty ChannelBuffer

-. Netty에서 제공하는 ChannelBuffer는 Sequential Access Indexing으로 순차적인 읽기,쓰기,폐기가 가능합니다.위 기능을 이용하여 버퍼의 내용을 가져올수있어 flip()메소드를 호출할 필요가 없습니다.

3.4 ZeroCopy



-. 기존(그림1)

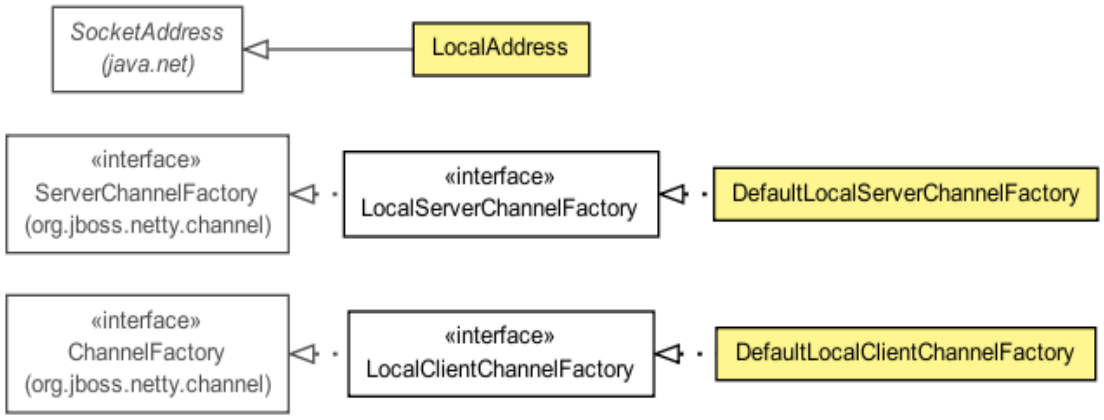
-. Zero Copy(그림2)

기존 방식을 다시 살펴 보면 두 번째와 세 번째 복사는 실제 필요하지 않음을 알 수 있을 것이다.
애플리케이션은 그저 데이터를 캐시했다가 소켓 버퍼로 보낼 뿐 아무런 일도 하지 않고
그림 2와 같이 데이터를 읽기 버퍼에서 소켓 버퍼로 직접 전송할 수 있다
즉 zero copy 기법은 데이터를 소켓을 이용하여 전송하려 할 때 중간 버퍼 간의 불필요한 데이터 복사를 피하게 해 주고, 사용자와 커널 사이의 context switching을 효과적으로 줄여줄 수 있다.
java.nio.channels.FileChannel 클래스의 transferTo() 메서드를 통해 무복사 기법을 지원한다

-. DMA(direct memory access)
CPU의 입출력 명령 하나에 의하여 CPU를 거치지 않고 일련의 정보(block)를 곧바로 입출력 장치에서 메모리로 또는 메모리로부터 입출력장치에 전달하는 기술.
대량의 데이터가 전송될 경우 CPU가 직접 제어한다면 CPU에 상당히 많은 부하가 걸리게 된다. 그러므로 CPU를 거치지 않고 직접 데이터를 전송할 수 있게 함으로써 CPU의 부하를 줄여서 전반적인 시스템 성능을 향상시키는 기술이 DMA이다.
DMA는 데이터를 불럭단위로 한번에 전송하게 되며 CPU의 개입이 없기 때문에 더욱 빠른 속도로 전송이 가능함으로 시스템의 전반적인 속도도 증가하게 하고, 하드디스크의 경우 DMA를 활성화 할 경우하드디스크 속도가 향상(약20-30%정도) 된다.

3.5 org.jboss.netty.channel.local

: 한 가상머신안에서 두파트의 가상통신을 지원한다.



이름	내용
LocalChannel	로컬 전송을 위한 Channel
LocalClientChannelFactory	클라이언트쪽의 로컬채널을 생성해주는 ChannelFactory
LocalServerChannel	로컬전송을 위한 ServerChannel
LocalServerChannelFactory	LocalServerChannel을 생성해주는 ServerChannelFactory
DefaultLocalClientChannelFactory	기본 LocalClientChannelFactory를 이행한다.

DefaultLocalClientServerFactory	기본 LocalserverChannelFactory를 이행한다.
LocalAddress	로컬전송의 종점

3.6 Executors

-. JDK 1.5 이상의 버전에서 지원한다.

Executors - 스레드풀의 내부의 스레드를 사용해 작업을 실행하는 방법을 사용하면, 작업별로 매번 스레드를 생성해 처리하는 방법보다는 굉장히 많은 장점이 있다. 매번 스레드를 생성하는 대신 이전에 사용했던 스레드를 재사용하기 때문에 스레드를 계속해서 생성할 필요가 없고, 따라서 여러개의 요청을 처리하는데 필요한 시스템 자원이 줄어드는 효과가 있다. 더군다나 클라이언트가 요청을 보냈을때 해당 요청을 처리할 스레드가 이미 만들어진 상태로 대기 하고 있기 때문에 작업을 실행하는 데 딜레이가 발생하지 않아 전체적인 반응 속도도 향상된다.

newFixedThreadPool : 처리할 작업이 등록되면 그에 따라 실제 작업할 스레드를 하나씩 생성한다. 생성 할 수 있는 스레드의 최대 개수는 제한되어 있으며 제한된 개수까지 스레드를 생성하고 나면 더 이상 생성하지 않고 스레드 수를 유지한다.
(만약 스레드가 작업하는 도중에 예상치 못한 예외가 발생해서 스레드가 종료되거나 하면 하나씩 더 생성하기도 한다.)

newCachedThreadPool : 캐시 스레드 풀은 현재 풀에 갖고 있는 스레드의 수가 처리 할 작업의 수보다 많아서 쉬는 스레드가 많이 발생할 때 쉬는 스레드를 종료시켜 훨씬 유연하게 대응할 수 있으며, 처리할 작업의 수가 많아지면 필요할 만큼 스레드를 새로 생성한다. 반면 스레드의 수에는 제한을 두지 않는다.

newSingleThreadExecutor : 단일 스레드로 동작하는 Executor로서 작업을 처리하는 스레드가 단 하나뿐이다. 만약 작업 중에 Exception이 발생해 비정상적으로 종료되면서 새로운 스레드를 하나 생성해 나머지 작업을 실행한다. 등록된 작업은 설정된 큐에서 지정하는 순서에 따라 반듯이 순차적으로 처리된다.

newScheduledThreadPool : 일정 시간 이후에 실행되거나 주기적으로 작업을 실행할 수 있으며, 스레드의 수가 고정되어 있는 형태의 Executor.Timer 클래스의 기능과 유사하다

3.7 SO_LINGER

TCP 는 여러개의 State 를 가질수 있습니다. 예를들어 CLOSED(연결이 닫혔을때) LISTEN(연결을 기다리고 있을때), ESTABLISHED(연결이 되었을때) 등등의 상태를 가질수 있습니다. 이중에서 TIME_WAIT 라는 상태가 있는데, 이해하기 가장 어려운 부분인것 같습니다. TIME_WAIT 상태는 다음과 같은 경우에 발생합니다. 우선 Client 와 Server 가 TCP 로 연결이 되어 있다고 가정합시다. 이때 클라이언트가 연결을 끊으려면

close 함수를 호출합니다. 이 함수를 호출하면 서버에 FIN segment 를 보내게 됩니다. 그러면 서버는 이 메시지를 받고 클라이언트가 접속을 끊으려고 하는 것을 알게됩니다. 따라서 서버가 CLOSE_WAIT 상태가 되면서 클라이언트에게 ack segment 를 보냅니다. 즉 "네가 접속을 끊는 다는 신호를 받았다" 이런 의미입니다. 이 메시지를 받으면 클라이언트는 FIN_WAIT_2 상태가 됩니다. 이 상태에서 서버는 자신의 socket 을 close 하고 다시 클라이언트에게 FIN segment 를 보냅니다. 즉 자신도 연결을 닫았다는 신호를 클라이언트에게 보내는 것입니다. 이 메시지를 받으면 클라이언트는 ack segment 를 보내면서 TIME_WAIT 상태가 됩니다. 즉 서로간의 확인하에 완전히 연결이 끊기게 됩니다. 근데 이 상태에서 곧바로 CLOSED 상태가 되는 것이 아니라 2 MSL(maximum segment lifetime - 1 분~4 분) 동안 TIME_WAIT 상태를 유지합니다. 그 이유는 TCP 는 신뢰성을 보장해 주는 프로토클 입니다. 따라서 연결시에도 신뢰성을 보장하기 위하여 Three-way handshake 라는 기법을 사용하여 연결을 하고 종료시에도 위와같이 복잡한 과정을 거쳐 서로가 close 된것을 확인하게 되는 것입니다. TIME_WAIT 상태도 이와같은 신뢰성 보장을 위한 한가지 방법이라고 보시면 됩니다.

SO_LINGER 옵션은 위 TIME_WAIT 상태를 제거하기위한 옵션으로 소켓의 빠른종료를 위해서 사용합니다 위 옵션을 사용할 경우 TIME_WAIT 상태가 없기 때문에 packet 전송이 안될 가능성이 발생합니다

3.8 OSGi

-. OSGi(Open Service Gateway initiative) Alliance는 1999년에 썬 마이크로시스템즈, IBM, 에릭손 등이 구성한 개방형 표준 단체이다. (OSGi Alliance는 처음에 Connected Alliance라고 불렸음) 그 뒤 여러 해 동안 OSGi Alliance는 원격 관리 될 수 있는 자바 기반의 서비스 플랫폼을 제정해왔다. 이 표준 사양의 핵심은 응용 프로그램의 생명주기(Life cycle) 모델과 서비스 레지스트리(Service Registry)를 정의하는 프레임워크(Framework)이다. OSGi 표준 사양에는 이 프레임워크에 기반하여 매우 다양한 OSGi 서비스가 정의되어 있다.

OSGi 프레임워크는 독립적인 자바/가상 머신 환경에서 제공하고 있지 못한 세련되고, 완전하며 동적인 컴포넌트 모델을 구현한다. 응용 프로그램 또는 구성 요소(번들, Bundle)는 다시 시동 과정 없이 원격지를 통해 설치(installed), 시작(started), 정지(stopped), 업데이트(updated) 그리고 제거(uninstalled)할 수 있다.

OSGi는 Embeddable(응용 프로그램 내부로 포함될 수 있는) SOA를 구현하고 있다. 이를 통해 응용 프로그램 개발에서 가장 복잡하고 관리하기가 어려운, 모듈간의 동적(Dynamic) 관계와 의존을 매우 효과적으로 관리할 수 있게 한다. (Web service based SOA가 네트워크를 중심으로 하는 SOA라면 OSGi는 Java Object based SOA 이다.)

3.9 ByteBuffer보다 ChannelBuffer의 성능이 좋은 이유

-. NIO의 ByteBuffer은 여러가지 바운더리 체크가 항상일어나지만 Netty의 ChannelBuffer은 바운더리 체크 제약이 보다 느슨하여 이로인한 복잡한 경비체크비용이 줄었으며, ByteBuffer과는 다른 인덱스의 장점과 JVM이 Buffer에 Access하기 쉬운 장점이 있다.

3.10 CODEC 예제

3.10.1 Codec.http

httpServer.java

```
public class httpServer {
    public static void main(String[] args) {
        // Configure the server.
        ServerBootstrap bootstrap = new ServerBootstrap(
                new NioServerSocketChannelFactory(
                        Executors.newCachedThreadPool(),
                        Executors.newCachedThreadPool()));

        // Set up the event pipeline factory.
        bootstrap.setPipelineFactory(new HttpServerPipelineFactory());

        // Bind and start to accept incoming connections.
        bootstrap.bind(new InetSocketAddress(8080));
    }
}
```

HttpServerPipelineFactory.java

```
public class HttpServerPipelineFactory implements ChannelPipelineFactory {
    public ChannelPipeline getPipeline() throws Exception {
        // Create a default pipeline implementation.
        ChannelPipeline pipeline = pipeline();
        pipeline.addLast("decoder", new HttpRequestDecoder());
        pipeline.addLast("aggregator", new HttpChunkAggregator(65536));
        pipeline.addLast("encoder", new HttpResponseEncoder());
        pipeline.addLast("handler", new httpServerHandler());
        return pipeline;
    }
}
```

httpServerIndexPage.java

```
public class httpServerIndexPage {
```

```
    private static final String NEWLINE = "₩r₩n";

    public static ChannelBuffer getContent(String webSocketLocation) {
        return ChannelBuffers.copiedBuffer(
                "<html><head><meta  http-equiv=₩"Content-Type₩"  content=₩"text/html;  charset=utf-
8₩">" +
                "<title>Web Socket Test</title></head>" + NEWLINE +
        "<body>" + NEWLINE +
        "Yuu_HeaVen"+
        "</body>" + NEWLINE +
        "</html>" + NEWLINE,
        CharsetUtil.UTF_8);
    }
    public static ChannelBuffer getContent2(String webSocketLocation) {
        return ChannelBuffers.copiedBuffer(
                "<html><head><meta  http-equiv=₩"Content-Type₩"  content=₩"text/html;  charset=utf-
8₩">" +
                "<title>Web Socket Test</title></head>" + NEWLINE +
                "<body>" + NEWLINE +
                "test.html <br> 테스트화면"+
                "</body>" + NEWLINE +
                "</html>" + NEWLINE,
                CharsetUtil.UTF_8);
            }
}
```

httpServerHandler.java

```
public class httpServerHandler extends SimpleChannelUpstreamHandler {

    private static final String WEBSOCKET_PATH = "/http";

    @Override
    public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) throws Exception {
        Object msg = e.getMessage();
        System.out.println(">>>>>MSG");
        System.out.println(msg);
```

```
        if (msg instanceof HttpRequest) {
            handleHttpRequest(ctx, (HttpRequest) msg);
        }
    }

    private void handleHttpRequest(ChannelHandlerContext ctx, HttpRequest req) throws Exception {
        // Allow only GET methods.
        if (req.getMethod() != GET) {
            sendHttpResponse(
                    ctx, req, new DefaultHttpResponse(HTTP_1_1, FORBIDDEN));
            return;
        }

        // Send the demo page.
        if (req.getUri().equals("/")) {
            HttpResponse res = new DefaultHttpResponse(HTTP_1_1, OK);

            ChannelBuffer content =
                httpServerIndexPage.getContent(getWebSocketLocation(req));

            res.setHeader(CONTENT_TYPE, "text/html; charset=utf-8");
            setContentLength(res, content.readableBytes());

            res.setContent(content);
            sendHttpResponse(ctx, req, res);
            return;
        }
        // Send the demo page.
        if (req.getUri().equals("/test.html")) {
            HttpResponse res = new DefaultHttpResponse(HTTP_1_1, OK);

            ChannelBuffer content =
                httpServerIndexPage.getContent2(getWebSocketLocation(req));

            res.setHeader(CONTENT_TYPE, "text/html; charset=utf-8");
            setContentLength(res, content.readableBytes());
```

```
            res.setContent(content);
            sendHttpResponse(ctx, req, res);
            return;
        }


        // Send an error page otherwise.
        sendHttpResponse(
                ctx, req, new DefaultHttpResponse(HTTP_1_1, FORBIDDEN));
    }

    private void sendHttpResponse(ChannelHandlerContext ctx, HttpRequest req, HttpResponse res) {
        // Generate an error page if response status code is not OK (200).
        if (res.getStatus().getCode() != 200) {
            res.setContent(
                    ChannelBuffers.copiedBuffer(
                            res.getStatus().toString(), CharsetUtil.UTF_8));
            setContentLength(res, res.getContent().readableBytes());
        }

        // Send the response and close the connection if necessary.
        ChannelFuture f = ctx.getChannel().write(res);
        if (!isKeepAlive(req) || res.getStatus().getCode() != 200) {
            f.addListener(ChannelFutureListener.CLOSE);
        }
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, ExceptionEvent e)
            throws Exception {
        e.getCause().printStackTrace();
        e.getChannel().close();
    }

    private String getWebSocketLocation(HttpRequest req) {
```

```
            return "ws://" + req.getHeader(HttpHeaders.Names.HOST) + WEBSOCKET_PATH;
        }
}
```

테스트 결과

Codec.http를 이용하여 작성한 테스트

http://127.0.0.1:8080/ 요청시

Yuu_HeaVen

http://127.0.0.1:8080/test.html 요청시

test.html
테스트화면

3.10.2 Codec.http.websocket

WebSocketServer.java

```
public class WebSocketServer {
    public static void main(String[] args) {
        // Configure the server.
        ServerBootstrap bootstrap = new ServerBootstrap(
                new NioServerSocketChannelFactory(
                        Executors.newCachedThreadPool(),
                        Executors.newCachedThreadPool()));

        // Set up the event pipeline factory.
        bootstrap.setPipelineFactory(new WebSocketServerPipelineFactory());
```

```
        // Bind and start to accept incoming connections.
        bootstrap.bind(new InetSocketAddress(8080));
    }
}
```

WebSocketServerPipelineFactory.java

```
public class WebSocketServerPipelineFactory implements ChannelPipelineFactory {
    public ChannelPipeline getPipeline() throws Exception {
        // Create a default pipeline implementation.
        ChannelPipeline pipeline = pipeline();
        pipeline.addLast("decoder", new HttpRequestDecoder());
        pipeline.addLast("aggregator", new HttpChunkAggregator(65536));
        pipeline.addLast("encoder", new HttpResponseEncoder());
        pipeline.addLast("handler", new WebSocketServerHandler());
        return pipeline;
    }
}
```

WebSocketServerHandler.java

```
public class WebSocketServerHandler extends SimpleChannelUpstreamHandler {

    private static final String WEBSOCKET_PATH = "/websocket";

    @Override
    public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) throws Exception {
        Object msg = e.getMessage();
        if (msg instanceof HttpRequest) {
            handleHttpRequest(ctx, (HttpRequest) msg);
        } else if (msg instanceof WebSocketFrame) {
            handleWebSocketFrame(ctx, (WebSocketFrame) msg);
        }
    }
    private void handleHttpRequest(ChannelHandlerContext ctx, HttpRequest req) throws Exception {
        // GET인지 확인
        if (req.getMethod() != GET) {
            sendHttpResponse(
                    ctx, req, new DefaultHttpResponse(HTTP_1_1, FORBIDDEN));
            return;
```

```
}

// HTTP 화면 전송
if (req.getUri().equals("/")) {
    HttpResponse res = new DefaultHttpResponse(HTTP_1_1, OK);

    ChannelBuffer content =
        WebSocketServerIndexPage.getContent(getWebSocketLocation(req));

    res.setHeader(CONTENT_TYPE, "text/html; charset=UTF-8");
    setContentLength(res, content.readableBytes());

    res.setContent(content);
    sendHttpResponse(ctx, req, res);
    return;
}

// Serve the WebSocket handshake request.
if (req.getUri().equals(WEBSOCKET_PATH) &&
    Values.UPGRADE.equalsIgnoreCase(req.getHeader(CONNECTION)) &&
    WEBSOCKET.equalsIgnoreCase(req.getHeader(Names.UPGRADE))) {

    // Create the WebSocket handshake response.
    HttpResponse res = new DefaultHttpResponse(
            HTTP_1_1,
            new HttpResponseStatus(101, "Web Socket Protocol Handshake"));
    res.addHeader(Names.UPGRADE, WEBSOCKET);
    res.addHeader(CONNECTION, Values.UPGRADE);

    // Fill in the headers and contents depending on handshake method.
    if (req.containsHeader(SEC_WEBSOCKET_KEY1) &&
        req.containsHeader(SEC_WEBSOCKET_KEY2)) {
        // New handshake method with a challenge:
        res.addHeader(SEC_WEBSOCKET_ORIGIN, req.getHeader(ORIGIN));
        res.addHeader(SEC_WEBSOCKET_LOCATION, getWebSocketLocation(req));
        String protocol = req.getHeader(SEC_WEBSOCKET_PROTOCOL);
```

```
        if (protocol != null) {
            res.addHeader(SEC_WEBSOCKET_PROTOCOL, protocol);
        }

        // Calculate the answer of the challenge.
        String key1 = req.getHeader(SEC_WEBSOCKET_KEY1);
        String key2 = req.getHeader(SEC_WEBSOCKET_KEY2);
        int a = (int) (Long.parseLong(key1.replaceAll("[^0-9]", "")) / key1.replaceAll("[^ ]", "").length());
        int b = (int) (Long.parseLong(key2.replaceAll("[^0-9]", "")) / key2.replaceAll("[^ ]", "").length());
        long c = req.getContent().readLong();
        ChannelBuffer input = ChannelBuffers.buffer(16);
        input.writeInt(a);
        input.writeInt(b);
        input.writeLong(c);
        ChannelBuffer output = ChannelBuffers.wrappedBuffer(
                MessageDigest.getInstance("MD5").digest(input.array()));
        res.setContent(output);
    } else {
        // Old handshake method with no challenge:
        res.addHeader(WEBSOCKET_ORIGIN, req.getHeader(ORIGIN));
        res.addHeader(WEBSOCKET_LOCATION, getWebSocketLocation(req));
        String protocol = req.getHeader(WEBSOCKET_PROTOCOL);
        if (protocol != null) {
            res.addHeader(WEBSOCKET_PROTOCOL, protocol);
        }
    }

    // Upgrade the connection and send the handshake response.
    ChannelPipeline p = ctx.getChannel().getPipeline();
    p.remove("aggregator");
    p.replace("decoder", "wsdecoder", new WebSocketFrameDecoder());

    ctx.getChannel().write(res);

    p.replace("encoder", "wsencoder", new WebSocketFrameEncoder());
    return;
```

```
        }

        // Send an error page otherwise.
        sendHttpResponse(
                ctx, req, new DefaultHttpResponse(HTTP_1_1, FORBIDDEN));
    }

    private void handleWebSocketFrame(ChannelHandlerContext ctx, WebSocketFrame frame) {
        //  WebSocket 받은 메세지를 echo 전송
        ctx.getChannel().write(
                new DefaultWebSocketFrame(frame.getTextData().toUpperCase()));
    }
    //HttpResponse전송
    private void sendHttpResponse(ChannelHandlerContext ctx, HttpRequest req, HttpResponse res) {
        // Generate an error page if response status code is not OK (200).
        if (res.getStatus().getCode() != 200) {
            res.setContent(
                    ChannelBuffers.copiedBuffer(
                            res.getStatus().toString(), CharsetUtil.UTF_8));
            setContentLength(res, res.getContent().readableBytes());
        }

        // Send the response and close the connection if necessary.
        ChannelFuture f = ctx.getChannel().write(res);
        if (!isKeepAlive(req) || res.getStatus().getCode() != 200) {
            f.addListener(ChannelFutureListener.CLOSE);
        }
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, ExceptionEvent e)
            throws Exception {
        e.getCause().printStackTrace();
        e.getChannel().close();
    }
```

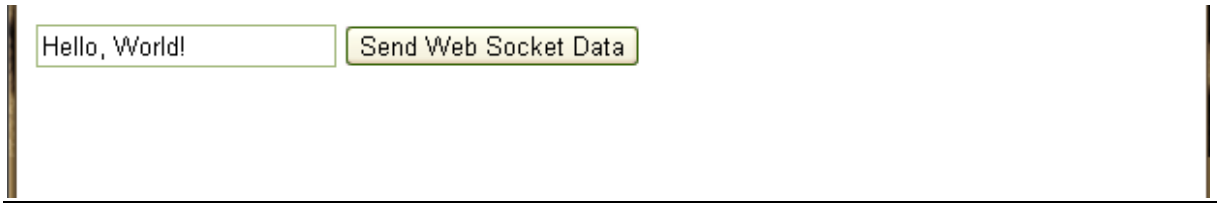
```
    private String getWebSocketLocation(HttpRequest req) {
        return "ws://" + req.getHeader(HttpHeaders.Names.HOST) + WEBSOCKET_PATH;
    }
}
```

```
// WebSocketServerIndexPage.java
public class WebSocketServerIndexPage {

    private static final String NEWLINE = "₩r₩n";

    public static ChannelBuffer getContent(String webSocketLocation) {
        return ChannelBuffers.copiedBuffer(
        "<html><head><title>Web Socket Test</title></head>" + NEWLINE +
        "<body>" + NEWLINE +
        "<script type=₩"text/javascript₩">" + NEWLINE +
        "var socket;" + NEWLINE +
        "if (window.WebSocket) {" + NEWLINE +
        "    socket = new WebSocket(₩"" + webSocketLocation + "₩");" + NEWLINE +
        "    socket.onmessage = function(event) { alert(event.data); };" + NEWLINE +
        "    socket.onopen = function(event) { alert(₩"Web Socket opened!₩"); };" + NEWLINE +
        "    socket.onclose = function(event) { alert(₩"Web Socket closed.₩"); };" + NEWLINE +
        "} else {" + NEWLINE +
        "    alert(₩"Your browser does not support Web Socket.₩");" + NEWLINE +
        "}" + NEWLINE +
        "" + NEWLINE +
        "function send(message) {" + NEWLINE +
        "    if (!window.WebSocket) { return; }" + NEWLINE +
        "    if (socket.readyState == WebSocket.OPEN) {" + NEWLINE +
        "        socket.send(message);" + NEWLINE +
        "    } else {" + NEWLINE +
        "        alert(₩"The socket is not open.₩");" + NEWLINE +
        "    }" + NEWLINE +
        "}" + NEWLINE +
        "</script>" + NEWLINE +
        "<form onsubmit=₩"return false;₩">" + NEWLINE +
        "<input type=₩"text₩" name=₩"message₩" value=₩"Hello, World!₩"/>" +
        "<input type=₩"button₩" value=₩"Send Web Socket Data₩" onclick=₩"send(this.form.message.value)₩" />" + NEWLINE
```

```
            "</form>" + NEWLINE +
            "</body>" + NEWLINE +
            "</html>" + NEWLINE,
            CharsetUtil.US_ASCII);
    }
}
```

테스트 결과

1) 서버에 연결시(http://localhost:8080) WebSocketServerIndexPage를 보내준다.



webSocketServer log

```
>>>>>MSG
DefaultHttpRequest(chunked: false)
GET / HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Cache-Control: max-age=0
Accept:
application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/534.7 (KHT
Gecko) Chrome/7.0.517.41 Safari/534.7
Accept-Encoding: gzip,deflate,sdch
Accept-Language: ko-KR,ko;q=0.8,en-US;q=0.6,en;q=0.4
Accept-Charset: windows-949,utf-8;q=0.7,*;q=0.3
>>>>>MSG
DefaultHttpRequest(chunked: false)
GET /favicon.ico HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Accept: */*
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US) AppleWebKit/534.7 (KHT
Gecko) Chrome/7.0.517.41 Safari/534.7
Accept-Encoding: gzip,deflate,sdch
Accept-Language: ko-KR,ko;q=0.8,en-US;q=0.6,en;q=0.4
```

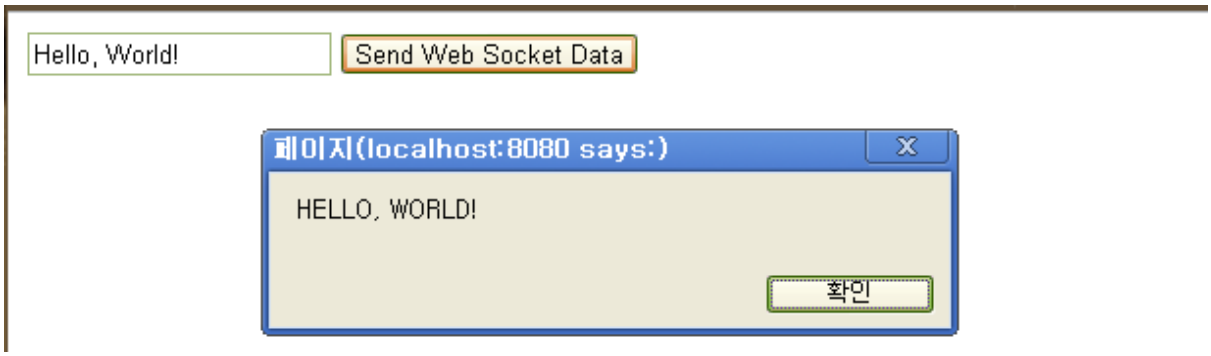
```
Accept-Charset: windows-949,utf-8;q=0.7,*;q=0.3
>>>>>MSG
DefaultHttpRequest(chunked: false)
GET /websocket HTTP/1.1
Upgrade: WebSocket
Connection: Upgrade
Host: localhost:8080
Origin: http://localhost:8080
Sec-WebSocket-Key1: 38c5   !  432  :  0   272
Sec-WebSocket-Key2: 9@R5 86218Uh2( 6WWl/n
----------------------------------------
```

2) 연결이 완료될시

```
p.remove("aggregator");
p.replace("decoder", "wsdecoder", new WebSocketFrameDecoder());
p.replace("encoder", "wsencoder", new WebSocketFrameEncoder());
```

Pipeline의 decoder, encoder 재정의

3) Send Web Socket Data를 클릭시 websocket로 메시지 전송



서버에서 받은 메시지는

ctx.getChannel().write(

new DefaultWebSocketFrame(frame.getTextData().toUpperCase()));

의 실행된 화면 toUpperCase를 통해서 전부 대문자로 변경되어 echo전송된다.

3.10.3 codec.oneone

TelnetServerHandler.java

```
public class TelnetServerHandler extends SimpleChannelUpstreamHandler {

    private static final Logger logger = Logger.getLogger(
            TelnetServerHandler.class.getName());

  //handleUpstream : 명시된 이벤트를 처리하다.
    @Override
    public void handleUpstream(
            ChannelHandlerContext ctx, ChannelEvent e) throws Exception {
        if (e instanceof ChannelStateEvent) {
            logger.info(e.toString());
        }
        super.handleUpstream(ctx, e);
    }

    //연결시 메세지를 보내준다.
    @Override
    public void channelConnected(
            ChannelHandlerContext ctx, ChannelStateEvent e) throws Exception {
        // Send greeting for a new connection.

        ChannelBuffer send1 = ChannelBuffers.dynamicBuffer(50);
        send1.writeBytes(("Welcome to " + InetAddress.getLocalHost().getHostName() + "!₩r₩n").getBytes());
        send1.writeBytes(("It is " + new Date() + " now.₩r₩n").getBytes());

        e.getChannel().write(send1);
    }

    //메세지를 받았을시
    @Override
    public void messageReceived(
            ChannelHandlerContext ctx, MessageEvent e) {

        // Cast to a String first.
        // We know it is a String because we put some codec in TelnetPipelineFactory.
```

```
        ChannelBuffer request = (ChannelBuffer) e.getMessage();

        if (request != null){
                System.err.println("request.readableBytes(): : ["+request.readableBytes()+"]");
        }else{
                System.err.println("request.readableBytes(): : [null]");
        }

    int totalLength = request.readableBytes();
    int headLength =  12;
    System.out.println("length :"+totalLength);
    System.out.println("headlength :"+headLength);
    byte[] message = new byte[totalLength - headLength];
    //System.arraycopy(request.array(), headLength, message, 0, length-headLength);

    request.skipBytes(12);
    request.readBytes(message);



    ChannelBuffer response = ChannelBuffers.dynamicBuffer(50);
    response.writeBytes(message);
    System.err.println(new String(message));

    ChannelFuture future = e.getChannel().write(response);
}

    //종료
    @Override
    public void exceptionCaught(
            ChannelHandlerContext ctx, ExceptionEvent e) {
        logger.log(
                Level.WARNING,
                "Unexpected exception from downstream.",
                e.getCause());
```

```
            e.getChannel().close();
        }
    }
```

TelnetServer.java

```
public class TelnetServer {

    public static void main(String[] args) throws Exception {
        // Configure the server.
        ServerBootstrap bootstrap = new ServerBootstrap(
                new NioServerSocketChannelFactory(
                        Executors.newCachedThreadPool(),
                        Executors.newCachedThreadPool()));

        TelnetServerHandler handler = new TelnetServerHandler();
        bootstrap.setPipelineFactory(new TelnetPipelineFactory(handler));

        // Bind and start to accept incoming connections.
        bootstrap.bind(new InetSocketAddress(10000));
    }
}
```

TelnetClient.java

```
public class TelnetClient {

    public static void main(String[] args) throws Exception {

        // Parse options.
        String host = "127.0.0.1";
        int port = 10000;

        // Configure the client.
        ClientBootstrap bootstrap = new ClientBootstrap(
                new NioClientSocketChannelFactory(
                        Executors.newCachedThreadPool(),
                        Executors.newCachedThreadPool()));

        TelnetClientHandler handler = new TelnetClientHandler();
```

```
bootstrap.setPipelineFactory(new TelnetPipelineFactory(handler));

// Start the connection attempt.
ChannelFuture future = bootstrap.connect(new InetSocketAddress(host, port));

// 커넥션 시도의 성공, 실패여부를 판단하기위하여 대기한다.
Channel channel = future.awaitUninterruptibly().getChannel();
if (!future.isSuccess()) {
    future.getCause().printStackTrace();
    //ChannelFactory의 자원해제
    bootstrap.releaseExternalResources();
    return;
}

// Read commands from the stdin.
ChannelFuture lastWriteFuture = null;
BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
for (;;) {
    String line = in.readLine();
    if (line == null) {
        break;
    }

    // Sends the received line to the server.
    ChannelBuffer header = ChannelBuffers.dynamicBuffer(12);
    ChannelBuffer payload = ChannelBuffers.dynamicBuffer(512);

    System.out.println(header);
    header.writeBytes("Yuu_HeaVen12".getBytes());

    payload.writeBytes(line.getBytes());

    ChannelBuffer buffer = ChannelBuffers.wrappedBuffer(header, payload);
    System.out.println("" + buffer.readableBytes());
    lastWriteFuture = channel.write(buffer);
}
```

```
        // Wait until all messages are flushed before closing the channel.
        // 메세지가 flushed되기를 대기한다.
        if (lastWriteFuture != null) {
            lastWriteFuture.awaitUninterruptibly();
        }

        // Close the connection.  Make sure the close operation ends because
        // all I/O operations are asynchronous in Netty.
        // Netty는 모든 I/O가 비동기이므로 종료시 operation를 반드시 닫아야한다.
        channel.close().awaitUninterruptibly();

        // Shut down all thread pools to exit.
        //ChannelFactory의 자원해제
        bootstrap.releaseExternalResources();
    }
}
```

TelnetClientHandler.java

```
public class TelnetClientHandler extends SimpleChannelUpstreamHandler {

    private static final Logger logger = Logger.getLogger(
            TelnetClientHandler.class.getName());

    //handleUpstream : 명시된 이벤트를 처리하다.
    @Override
    public void handleUpstream(
            ChannelHandlerContext ctx, ChannelEvent e) throws Exception {
        if (e instanceof ChannelStateEvent) {
            logger.info(e.toString());
        }
        super.handleUpstream(ctx, e);
    }

    @Override
    public void messageReceived(ChannelHandlerContext ctx, MessageEvent e) {
        // Print out the line received from the server.
```

```
            ChannelBuffer sendMessage = (ChannelBuffer)e.getMessage();
            byte[] message = sendMessage.array();

            System.err.println("To server   ---------- ");
            System.err.println(new String(message));
            System.err.println("--------------------------");
        }

        @Override
        public void exceptionCaught(
                ChannelHandlerContext ctx, ExceptionEvent e) {
            logger.log(
                    Level.WARNING,
                    "Unexpected exception from downstream.",
                    e.getCause());
            e.getChannel().close();
        }
}
```

oneToone codec의 사용법

```
public class TelnetPipelineFactory implements
        ChannelPipelineFactory {

    private final ChannelHandler handler;

    public TelnetPipelineFactory(ChannelHandler handler) {
        this.handler = handler;
    }

    public ChannelPipeline getPipeline() throws Exception {
        // Create a default pipeline implementation.
        ChannelPipeline pipeline = pipeline();

        // Add the text line codec combination first,
        pipeline.addLast("decoder", new StringDecoder());
        pipeline.addLast("encoder", new StringEncoder());
```

```
pipeline.addLast("customDecoder", new OneToOneDecoder(){
                @Override
                protected Object decode(ChannelHandlerContext arg0, Channel arg1,
                                Object arg2) throws Exception {
                        System.out.println("decode ChannelHandlerContext" + arg0);
                        System.out.println("decode Channel" + arg1);
                        System.out.println("decode Object" + arg2);

                        String message = ((String)arg2);
                        ChannelBuffer content = ChannelBuffers.dynamicBuffer(50);
                        content.writeBytes(message.getBytes());

                        return (Object)content;
                        }
});
pipeline.addLast("customEncoder", new OneToOneEncoder(){

                @Override
                protected Object encode(ChannelHandlerContext arg0, Channel arg1,
                                Object arg2) throws Exception {
                        // TODO Auto-generated method stub
                        System.out.println("encode ChannelHandlerContext" + arg0);
                        System.out.println("encode Channel" + arg1);
                        System.out.println("encode Object" + arg2);

                        ChannelBuffer message = ChannelBuffers.dynamicBuffer(50);
                        message = ((ChannelBuffer) arg2).copy(0, ((ChannelBuffer)
arg2).readableBytes());

                    byte[] array = new byte[message.readableBytes()];
                    array = message.array();
                        String content = new String(array);

                        return (Object)content;
                }
```

```
        });

        // and then business logic.
        pipeline.addLast("handler", handler);
        return pipeline;
    }
}
```

3.10.4 codec.serialization

흔히 사용하지 않는 double[] 같은 자바객체를 전송하는대 유용하며 굳이 double[]같은것이 아니더래도 자바객체라면 어떤것이든 관계없다. (**단, Serializable 인터페이스를 구현한 객체만 가능합니다**.)
위는 텔넷예제에서 decoder와 encoder을 수정하여서 전송한것이다.

```
public class TelnetPipelineFactory implements
        ChannelPipelineFactory {

    private final ChannelHandler handler;

    public TelnetPipelineFactory(ChannelHandler handler) {
        this.handler = handler;
    }

    public ChannelPipeline getPipeline() throws Exception {
        // Create a default pipeline implementation.
        ChannelPipeline pipeline = pipeline();

        // Add the text line codec combination first,
        pipeline.addLast("decoder", new ObjectDecoder());
        pipeline.addLast("encoder", new ObjectEncoder());

        // and then business logic.
        pipeline.addLast("handler", handler);

        return pipeline;
    }
}
```

Serializable 는 구현할 필요없이 선언만하면된다. 아래는 테스트용으로 만든 직렬화객체

sendObject.java

```
import java.io.Serializable;

public class sendObject implements Serializable {

	String name;
	int age;
	public void setName(String value){
		name = value;
	}
	public String getName(){
		return name;
	}
	public void setAge(int value){
		age = value;
	}
	public int getAge(){
		return age;
	}
}
```

TelnetClient.java 중 아래부분을 수정하였다.

```
sendObject a = new sendObject();
a.setName("Yuu");
a.setAge(29);
lastWriteFuture = channel.write(a);
```

TelnetServerHandler에서 messageReceived를 수정한부분을 추가하였다.

```
    //메세지를 받았을시
    @Override
    public void messageReceived(
            ChannelHandlerContext ctx, MessageEvent e) {

        sendObject a = (sendObject)e.getMessage();
        String response;
```

```
        response = "Did your Name '" + a.getName() + "'?₩r₩n";
        response += "Did your Age '" + a.getAge() + "'?₩r₩n";
        ChannelFuture future = e.getChannel().write(response);
    }
```

테스트 결과

```
2010. 12. 13 오전 9:49:43
org.jboss.netty.example.serialization.TelnetClientHandler handleUpstream
정보: [id: 0x00f81843] OPEN
2010. 12. 13 오전 9:49:43
org.jboss.netty.example.serialization.TelnetClientHandler handleUpstream
정보: [id: 0x00f81843, /127.0.0.1:46066 => /127.0.0.1:10000] BOUND:
/127.0.0.1:46066
2010. 12. 13 오전 9:49:43
org.jboss.netty.example.serialization.TelnetClientHandler handleUpstream
정보: [id: 0x00f81843, /127.0.0.1:46066 => /127.0.0.1:10000] CONNECTED:
/127.0.0.1:10000
Welcome to your-9488f82c8b!

It is Mon Dec 13 09:49:43 KST 2010 now.

Did your Name 'Yuu'?
Did your Age '29'?
```