Netty编解码

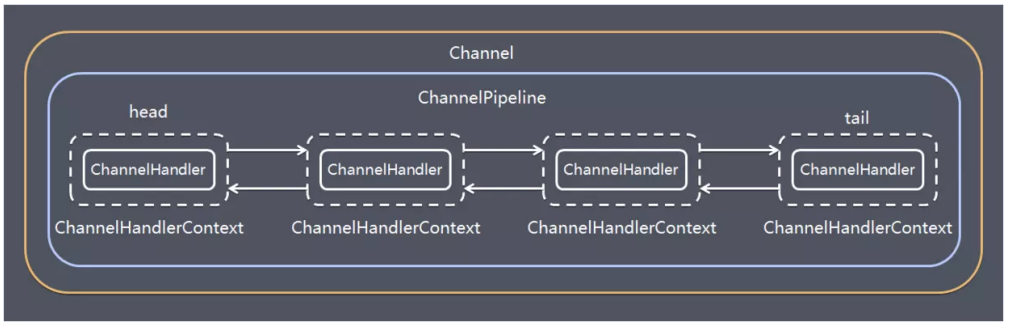
Netty涉及到编解码的组件有Channel、ChannelHandler、ChannelPipe等，先大概了解下这几个组件的作用。

ChannelHandler

ChannelHandler充当了处理入站和出站数据的应用程序逻辑容器。例如，实现ChannelInboundHandler接口（或ChannelInboundHandlerAdapter），你就可以接收入站事件和数据，这些数据随后会被你的应用程序的业务逻辑处理。当你要给连接的客户端发送响应时，也可以从ChannelInboundHandler冲刷数据。你的业务逻辑通常写在一个或者多个ChannelInboundHandler中。ChannelOutboundHandler原理一样，只不过它是用来处理出站数据的。

ChannelPipeline

ChannelPipeline提供了ChannelHandler链的容器。**以客户端应用程序为例**，如果事件的运动方向是从客户端到服务端的，那么我们称这些事件为**出站的**，即客户端发送给服务端的数据会通过pipeline中的一系列**ChannelOutboundHandler(ChannelOutboundHandler调用是从tail到head方向逐个调用每个handler的逻辑)**，并被这些Handler处理，反之则称为**入站的，**入站只调用pipeline里的**ChannelInboundHandler**逻辑**(ChannelInboundHandler调用是从head到tail方向逐个调用每个handler的逻辑)**。



编码解码器

当你通过Netty发送或者接受一个消息的时候，就将会发生一次数据转换。入站消息会被**解码**：从字节转换为另一种格式（比如java对象）；如果是出站消息，它会被**编码成字节。**

Netty提供了一系列实用的编码解码器，他们都实现了ChannelInboundHadnler或者ChannelOutboundHandler接口。在这些类中，channelRead方法已经被重写了。以入站为例，对于每个从入站Channel读取的消息，这个方法会被调用。随后，它将调用由已知解码器所提供的decode()方法进行解码，并将已经解码的字节转发给ChannelPipeline中的下一个ChannelInboundHandler。

Netty提供了很多编解码器，比如编解码字符串的StringEncoder和StringDecoder，编解码对象的ObjectEncoder和ObjectDecoder等。

如果要实现高效的编解码可以用protobuf，但是protobuf需要维护大量的proto文件比较麻烦，现在一般可以使用protostuff。

protostuff是一个基于protobuf实现的序列化方法，它较于protobuf最明显的好处是，在几乎不损耗性能的情况下做到了不用我们写.proto文件来实现序列化。使用它也非常简单，代码如下：

引入依赖：

```
<dependency>
  <groupId>com.dyuproject.protostuff</groupId>
  <artifactId>protostuff-api</artifactId>
  <version>1.0.10</version>
</dependency>
<dependency>
  <groupId>com.dyuproject.protostuff</groupId>
  <artifactId>protostuff-core</artifactId>
  <version>1.0.10</version>
 </dependency>
 <dependency>
   <groupId>com.dyuproject.protostuff</groupId>
   <artifactId>protostuff-runtime</artifactId>
   <version>1.0.10</version>
 </dependency>
```

protostuff使用示例：

```
import com.dyuproject.protostuff.LinkedBuffer;
import com.dyuproject.protostuff.ProtostuffIOUtil;
import com.dyuproject.protostuff.Schema;
import com.dyuproject.protostuff.runtime.RuntimeSchema;

```

```java
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

/**
 * protostuff 序列化工具类，基于protobuf封装
 */
public class ProtostuffUtil {

 private static Map<Class<?>, Schema<?>> cachedSchema = new ConcurrentHashMap<Class<?>, Schema<?>>();

 private static <T> Schema<T> getSchema(Class<T> clazz) {
 @SuppressWarnings("unchecked")
 Schema<T> schema = (Schema<T>) cachedSchema.get(clazz);
 if (schema == null) {
 schema = RuntimeSchema.getSchema(clazz);
 if (schema != null) {
 cachedSchema.put(clazz, schema);
 }
 }
 return schema;
 }

 /**
 * 序列化
 *
 * @param obj
 * @return
 */
 public static <T> byte[] serializer(T obj) {
 @SuppressWarnings("unchecked")
 Class<T> clazz = (Class<T>) obj.getClass();
 LinkedBuffer buffer = LinkedBuffer.allocate(LinkedBuffer.DEFAULT_BUFFER_SIZE);
 try {
 Schema<T> schema = getSchema(clazz);
 return ProtostuffIOUtil.toByteArray(obj, schema, buffer);
 } catch (Exception e) {
 throw new IllegalStateException(e.getMessage(), e);
 } finally {
 buffer.clear();
 }
 }

 /**
 * 反序列化
 *
 * @param data
 * @param clazz
 * @return
 */
 public static <T> T deserializer(byte[] data, Class<T> clazz) {
 try {
 T obj = clazz.newInstance();
 Schema<T> schema = getSchema(clazz);
 ProtostuffIOUtil.mergeFrom(data, obj, schema);
 return obj;
 } catch (Exception e) {
 throw new IllegalStateException(e.getMessage(), e);
 }
 }

 public static void main(String[] args) {
```

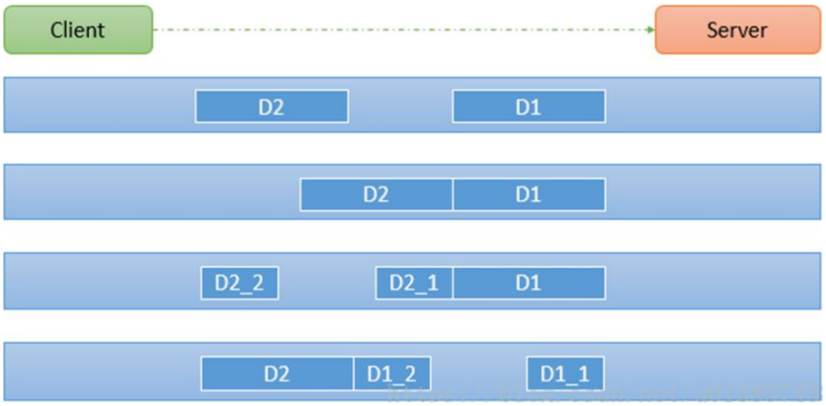
```
67  byte[] userBytes = ProtostuffUtil.serializer(new User(1, "zhuge"));
68  User user = ProtostuffUtil.deserializer(userBytes, User.class);
69  System.out.println(user);
70  }
71 }
```

参见项目示例com.tuling.netty.codec包下代码

Netty粘包拆包

TCP是一个流协议，就是没有界限的一长串二进制数据。TCP作为传输层协议并不不了解上层业务数据的具体含义，它会根据TCP缓冲区的实际情况进行数据包的划分，所以在业务上认为是一个完整的包，可能会被TCP拆分成多个包进行发送，也有可能把多个小的包封装成一个大的数据包发送，这就是所谓的TCP粘包和拆包问题。面向流的通信是无消息保护边界的。

如下图所示，client发了两个数据包D1和D2，但是server端可能会收到如下几种情况的数据。



解决方案

1）消息定长度，传输的数据大小固定长度，例如每段的长度固定为100字节，如果不够空位补空格

2）在数据包尾部添加特殊分隔符，比如下划线，中划线等，这种方法简单易行，但选择分隔符的时候一定要注意每条数据的内部一定不能出现分隔符。

3）发送长度：发送每条数据的时候，将数据的长度一并发送，比如可以选择每条数据的前4位是数据的长度，应用层处理时可以根据长度来判断每条数据的开始和结束。

Netty提供了多个解码器，可以进行分包的操作，如下：

- LineBasedFrameDecoder （回车换行分包）
- DelimiterBasedFrameDecoder（特殊分隔符分包）
- FixedLengthFrameDecoder（固定长度报文来分包）

自定义长度分包解码器，参见项目示例com.tuling.netty.split包下代码

Netty心跳检测机制

所谓心跳, 即在 TCP 长连接中, 客户端和服务器之间定期发送的一种特殊的数据包, 通知对方自己还在线, 以确保 TCP 连接的有效性.

在 Netty 中, 实现心跳机制的关键是 IdleStateHandler, 看下它的构造器：

```
1 public IdleStateHandler(int readerIdleTimeSeconds, int writerIdleTimeSeconds, int allIdleTimeSeconds) {
2  this((long)readerIdleTimeSeconds, (long)writerIdleTimeSeconds, (long)allIdleTimeSeconds, TimeUnit.SECONDS);
3 }
```

这里解释下三个参数的含义：

- readerIdleTimeSeconds: 读超时. 即当在指定的时间间隔内没有从 Channel 读取到数据时, 会触发一个 READER_IDLE 的 IdleStateEvent 事件.
- writerIdleTimeSeconds: 写超时. 即当在指定的时间间隔内没有数据写入到 Channel 时, 会触发一个 WRITER_IDLE 的 IdleStateEvent 事件.
- allIdleTimeSeconds: 读/写超时. 即当在指定的时间间隔内没有读或写操作时, 会触发一个 ALL_IDLE 的 IdleStateEvent 事件.

注：这三个参数默认的时间单位是秒。若需要指定其他时间单位，可以使用另一个构造方法：

```
1 IdleStateHandler(boolean observeOutput, long readerIdleTime, long writerIdleTime, long allIdleTime, TimeUnit unit)
```

要实现Netty服务端心跳检测机制需要在服务器端的ChannelInitializer中加入如下的代码：

```
1  pipeline.addLast(new IdleStateHandler(3, 0, 0, TimeUnit.SECONDS));
```

初步地看下IdleStateHandler源码，先看下IdleStateHandler中的channelRead方法：

```
@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
    if (readerIdleTimeNanos > 0 || allIdleTimeNanos > 0) {
        reading = true;
        firstReaderIdleEvent = firstAllIdleEvent = true;
    }
    ctx.fireChannelRead(msg);
}
```

红框代码其实表示该方法只是进行了透传，不做任何业务逻辑处理，让channelPipe中的下一个handler处理channelRead方法

我们再看看channelActive方法：

```
@Override
public void channelActive(ChannelHandlerContext ctx) throws Exception {
    // This method will be invoked only if this handler was added
    // before channelActive() event is fired.  If a user adds this handler
    // after the channelActive() event, initialize() will be called by beforeAdd().
    initialize(ctx);
    super.channelActive(ctx);
}
```

这里有个initialize的方法，这是IdleStateHandler的精髓，接着探究：

```
private void initialize(ChannelHandlerContext ctx) {
    // Avoid the case where destroy() is called before scheduling timeouts.
    // See: https://github.com/netty/netty/issues/143
    switch (state) {
    case 1:
    case 2:
        return;
    }

    state = 1;
    initOutputChanged(ctx);

    lastReadTime = lastWriteTime = ticksInNanos();
    if (readerIdleTimeNanos > 0) {
        readerIdleTimeout = schedule(ctx, new ReaderIdleTimeoutTask(ctx),
                readerIdleTimeNanos, TimeUnit.NANOSECONDS);
    }
    if (writerIdleTimeNanos > 0) {
        writerIdleTimeout = schedule(ctx, new WriterIdleTimeoutTask(ctx),
                writerIdleTimeNanos, TimeUnit.NANOSECONDS);
    }
    if (allIdleTimeNanos > 0) {
        allIdleTimeout = schedule(ctx, new AllIdleTimeoutTask(ctx),
                allIdleTimeNanos, TimeUnit.NANOSECONDS);
    }
}
```

这边会触发一个Task，ReaderIdleTimeoutTask，这个task里的run方法源码是这样的：

```
@Override
protected void run(ChannelHandlerContext ctx) {
    long nextDelay = readerIdleTimeNanos;
    if (!reading) {
        nextDelay -= ticksInNanos() - lastReadTime;
    }

    if (nextDelay <= 0) {
        // Reader is idle - set a new timeout and notify the callback.
        readerIdleTimeout = schedule(ctx, task: this, readerIdleTimeNanos, TimeUnit.NANOSECONDS);

        boolean first = firstReaderIdleEvent;
        firstReaderIdleEvent = false;

        try {
            IdleStateEvent event = newIdleStateEvent(IdleState.READER_IDLE, first);
            channelIdle(ctx, event);
        } catch (Throwable t) {
            ctx.fireExceptionCaught(t);
        }
    } else {
        // Read occurred before the timeout - set a new timeout with shorter delay.
        readerIdleTimeout = schedule(ctx, task: this, nextDelay, TimeUnit.NANOSECONDS);
    }
}
```

第一个红框代码是用当前时间减去最后一次channelRead方法调用的时间，假如这个结果是6s，说明最后一次调用channelRead已经是6s之前的事情了，你设置的是5s，那么nextDelay则为-1，说明超时了，那么第二个红框代码则会触发下一个handler的userEventTriggered方法：

```
protected void channelIdle(ChannelHandlerContext ctx, IdleStateEvent evt) throws Exception {
    ctx.fireUserEventTriggered(evt);
}
```

如果没有超时则不触发userEventTriggered方法。

Netty心跳检测代码示例：

```
1  //服务端代码
2  public class HeartBeatServer {
3
4   public static void main(String[] args) throws Exception {
5   EventLoopGroup boss = new NioEventLoopGroup();
6   EventLoopGroup worker = new NioEventLoopGroup();
7   try {
8   ServerBootstrap bootstrap = new ServerBootstrap();
9   bootstrap.group(boss, worker)
10   .channel(NioServerSocketChannel.class)
11   .childHandler(new ChannelInitializer<SocketChannel>() {
12   @Override
13   protected void initChannel(SocketChannel ch) throws Exception {
14   ChannelPipeline pipeline = ch.pipeline();
15   pipeline.addLast("decoder", new StringDecoder());
16   pipeline.addLast("encoder", new StringEncoder());
17   //IdleStateHandler的readerIdleTime参数指定超过3秒还没收到客户端的连接，
18   //会触发IdleStateEvent事件并且交给下一个handler处理，下一个handler必须
19   //实现userEventTriggered方法处理对应事件
20   pipeline.addLast(new IdleStateHandler(3, 0, 0, TimeUnit.SECONDS));
21   pipeline.addLast(new HeartBeatHandler());
22   }
23   });
24   System.out.println("netty server start。。");
25   ChannelFuture future = bootstrap.bind(9000).sync();
26   future.channel().closeFuture().sync();
27   } catch (Exception e) {
28   e.printStackTrace();
29   } finally {
30   worker.shutdownGracefully();
```

```
 boss.shutdownGracefully();
 }
 }
}

```

```
//服务端处理handler
public class HeartBeatServerHandler extends SimpleChannelInboundHandler<String> {

 int readIdleTimes = 0;

 @Override
 protected void channelRead0(ChannelHandlerContext ctx, String s) throws Exception {
 System.out.println(" ====== > [server] message received : " + s);
 if ("Heartbeat Packet".equals(s)) {
 ctx.channel().writeAndFlush("ok");
 } else {
 System.out.println(" 其他信息处理 ... ");
 }
 }

 @Override
 public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception {
 IdleStateEvent event = (IdleStateEvent) evt;

 String eventType = null;
 switch (event.state()) {
 case READER_IDLE:
 eventType = "读空闲";
 readIdleTimes++; // 读空闲的计数加1
 break;
 case WRITER_IDLE:
 eventType = "写空闲";
 // 不处理
 break;
 case ALL_IDLE:
 eventType = "读写空闲";
 // 不处理
 break;
 }
 System.out.println(ctx.channel().remoteAddress() + "超时事件: " + eventType);
 if (readIdleTimes > 3) {
 System.out.println(" [server]读空闲超过3次，关闭连接，释放更多资源");
 ctx.channel().writeAndFlush("idle close");
 ctx.channel().close();
 }
 }

 @Override
 public void channelActive(ChannelHandlerContext ctx) throws Exception {
 System.err.println("=== " + ctx.channel().remoteAddress() + " is active ===");
 }
}
```

```
//客户端代码
public class HeartBeatClient {
public static void main(String[] args) throws Exception {
EventLoopGroup eventLoopGroup = new NioEventLoopGroup();
try {
Bootstrap bootstrap = new Bootstrap();
```

```
bootstrap.group(eventLoopGroup).channel(NioSocketChannel.class)
.handler(new ChannelInitializer<SocketChannel>() {
@Override
protected void initChannel(SocketChannel ch) throws Exception {
ChannelPipeline pipeline = ch.pipeline();
pipeline.addLast("decoder", new StringDecoder());
pipeline.addLast("encoder", new StringEncoder());
pipeline.addLast(new HeartBeatClientHandler());
}
});

System.out.println("netty client start。。");
Channel channel = bootstrap.connect("127.0.0.1", 9000).sync().channel();
String text = "Heartbeat Packet";
Random random = new Random();
while (channel.isActive()) {
int num = random.nextInt(10);
Thread.sleep(num * 1000);
channel.writeAndFlush(text);
}
} catch (Exception e) {
e.printStackTrace();
} finally {
eventLoopGroup.shutdownGracefully();
}
}

static class HeartBeatClientHandler extends SimpleChannelInboundHandler<String> {

@Override
protected void channelRead0(ChannelHandlerContext ctx, String msg) throws Exception {
System.out.println(" client received :" + msg);
if (msg != null && msg.equals("idle close")) {
System.out.println(" 服务端关闭连接，客户端也关闭");
ctx.channel().closeFuture();
}
}
}
}
```

Netty断线自动重连实现

1、客户端启动连接服务端时，如果网络或服务端有问题，客户端连接失败，可以重连，重连的逻辑加在客户端。
参见代码com.tuling.netty.reconnect.NettyClient
2、系统运行过程中网络故障或服务端故障，导致客户端与服务端断开连接了也需要重连，可以在客户端处理数据的Handler的channelInactive方法中进行重连。
参见代码com.tuling.netty.reconnect.NettyClientHandler

```
文档：03-VIP-Netty编解码，粘包拆包及零拷贝详解
链接：http://note.youdao.com/noteshare?id=b8970e44473486a48178193d68929008&sub=2FBCDAF79D794F8DBBD027A5F2C29249
```