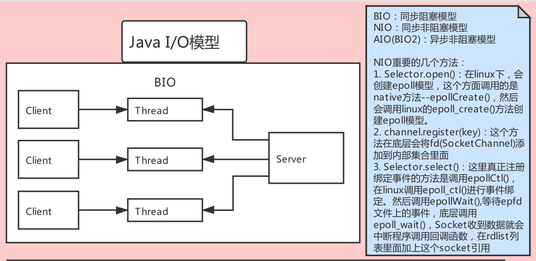
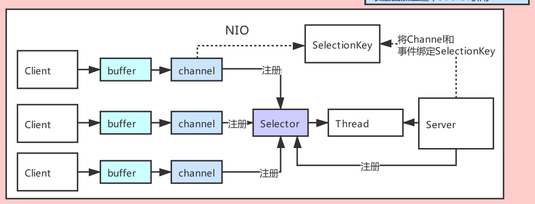
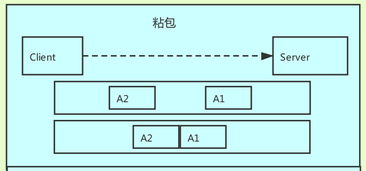


BIO：同步阻塞模型
NIO：同步非阻塞模型
AIO(BIO2)：异步非阻塞模型

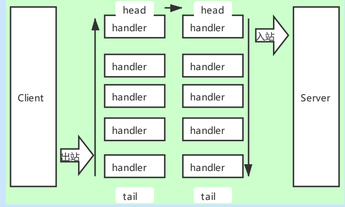
NIO重要的几个方法：
1. Selector.open()：在linux下，会创建epoll模型，这个方面调用的是native方法--epollCreate()，然后会调用linux的epoll_create()方法创建epoll模型。
2. channel.register(key)：这个方法在底层会将fd(SocketChannel)添加到内部集合里面
3. Selector.select()：这里真正注册绑定事件的方法是调用epollCtl()，在linux调用epoll_ctl()进行事件绑定。然后调用epollWait()等待epfd文件上的事件，底层调用epoll_wait()，Socket收到数据就会中断程序调用回调函数，在rdlist列表里面加上这个socket引用





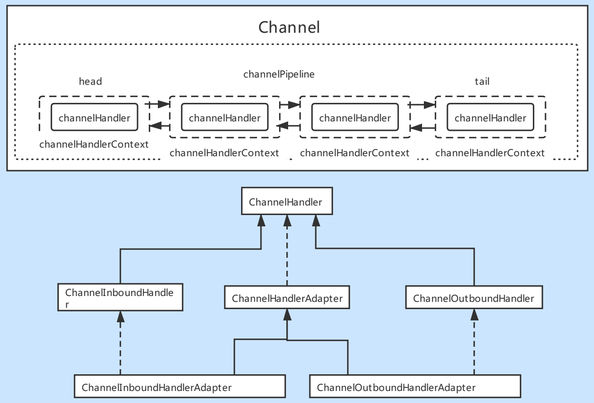
粘包拆包解决方法：
(1)消息定长度：规定每天数据的长度，不足空位补齐。
(2)在数据包尾部添加特殊分隔符
(3)发送数据长度：发送数据的时候带上数据的长度

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)。



handler的生命周期回调接口调用顺序:
handlerAdded -> channelRegistered -> channelActive -> channelRead -> channelReadComplete-> channelInactive -> channelUnRegistered -> handlerRemoved

andlerAdded: 新建立的连接会按照初始化策略，把handler添加到该channel的pipeline里面，也就是channel.pipeline.addLast(new LifeCycleInBoundHandler)执行完成后的回调；
hannelRegistered: 当连接分配到具体的worker线程后，该回调会被调用。
hannelActive：channe的准备工作已经完成，所有的pipeline添加完成，并分配到具体的线上上，说明该channel准备就绪，可以使用了。
hannelRead：客户端向服务端发来数据，每次都会回调此方法，表示有数据可读；
hannelReadComplete：服务端每次读完一次完整的数据之后，回调该方法，表示数据读取完毕；
hannelInactive：当连接断开时，该回调会被调用，说明这时候底层的TCP连接已经被断开了。
hannelUnRegistered: 对应channelRegistered，当连接关闭后，释放绑定的workder线程；
andlerRemoved： 对应handlerAdded，将handler从该channel的pipeline移除后的回调方法。



心跳机制

心跳机制的实现，需要通过实现IdleHandler：
```
public IdleStateHandler(int readerIdleTimeSeconds, int writerIdleTimeSeconds, int allIdleTimeSeconds) {
        this((long)readerIdleTimeSeconds, (long)writerIdleTimeSeconds, (long)allIdleTimeSeconds, TimeUnit.SECONDS);
    }
```
重要的三个参数：
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。

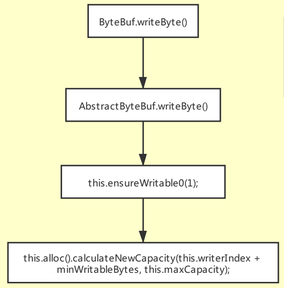
实现Netty服务端心跳检测机制需要在服务器端的ChannelInitializer加上：
```
pipeline.addLast(new IdleStateHandler(3, 0, 0, TimeUnit.SECONDS));
```
IdleStateHandler继承了ChannelInboundHandlerAdapter，所以是入站处理器

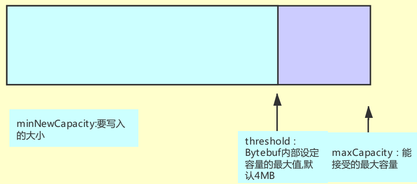
在IdleStateHandler的channelActive方法中有initialize(核心)方法：
```
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        this.initialize(ctx);
        super.channelActive(ctx);
    }

    private void initialize(ChannelHandlerContext ctx) {
        switch(this.state) {
        case 1:
        case 2:
            return;
        default:
            this.state = 1;
            this.initOutputChanged(ctx);
            this.lastReadTime = this.lastWriteTime = this.ticksInNanos();
            if (this.readerIdleTimeNanos > 0L) {
                this.readerIdleTimeout = this.schedule(ctx, new IdleStateHandler.ReaderIdleTimeoutTask(ctx), this.readerIdleTimeNanos,
TimeUnit.NANOSECONDS);
            }

            if (this.writerIdleTimeNanos > 0L) {
                this.writerIdleTimeout = this.schedule(ctx, new IdleStateHandler.WriterIdleTimeoutTask(ctx), this.writerIdleTimeNanos,
TimeUnit.NANOSECONDS);
            }

            if (this.allIdleTimeNanos > 0L) {
                this.allIdleTimeout = this.schedule(ctx, new IdleStateHandler.AllIdleTimeoutTask(ctx), this.allIdleTimeNanos,
TimeUnit.NANOSECONDS);
            }

        }
    }
```
在这里面有传进去一个ReaderIdleTimeoutTask，它实现了Runable，查看它的run()方法：
```
        protected void run(ChannelHandlerContext ctx) {
            long nextDelay = IdleStateHandler.this.readerIdleTimeNanos;
            if (!IdleStateHandler.this.reading) {
                nextDelay -= IdleStateHandler.this.ticksInNanos() - IdleStateHandler.this.lastReadTime;
            }

            if (nextDelay <= 0L) {
                IdleStateHandler.this.readerIdleTimeout = IdleStateHandler.this.schedule(ctx, this,
IdleStateHandler.this.readerIdleTimeNanos, TimeUnit.NANOSECONDS);
                boolean first = IdleStateHandler.this.firstReaderIdleEvent;
                IdleStateHandler.this.firstReaderIdleEvent = false;

                try {
                    IdleStateEvent event = IdleStateHandler.this.newIdleStateEvent(IdleState.READER_IDLE, first);
                    IdleStateHandler.this.channelIdle(ctx, event);
                } catch (Throwable var6) {
                    ctx.fireExceptionCaught(var6);
                }
            } else {
                IdleStateHandler.this.readerIdleTimeout = IdleStateHandler.this.schedule(ctx, this, nextDelay, TimeUnit.NANOSECONDS);
            }

        }
    }
```
它会判断现在的时间-上次收到数据的时间是否<=0,如果是的话，就会执行channelIdle(ctx, event)，触发userEventTriggered，所以下一个handler必须实现userEventTriggered()方法，从而实现超时处理：
```
pipeline.addLast(new IdleStateHandler(3, 0, 0, TimeUnit.SECONDS));
pipeline.addLast(new HeartBeatServerHandler());
```

断开重连

在客户端的handler中，重写channelInactive()，做重连操作
```
    // channel 处于不活动状态时调用
    @Override
    public void channelInactive(ChannelHandlerContext ctx) throws Exception {
        System.err.println("运行中断开重连。。。");
        nettyClient.connect();
    }
```

ByteBuf扩容机制






```
    public int calculateNewCapacity(int minNewCapacity,
int maxCapacity) {
        ObjectUtil.checkPositiveOrZero(minNewCapacity,
"minNewCapacity");
        if (minNewCapacity > maxCapacity) {
            throw new
IllegalArgumentException(String.format("minNewCapacit
y: %d (expected: not greater than maxCapacity(%d)",
minNewCapacity, maxCapacity));
        } else {
            int threshold = 4194304;
            if (minNewCapacity == 4194304) {
                return 4194304;
            } else {
                int newCapacity;
                if (minNewCapacity > 4194304) {
                    newCapacity = minNewCapacity / 4194304 *
4194304;
                    if (newCapacity > maxCapacity - 4194304) {
                        newCapacity = maxCapacity;
                    } else {
                        newCapacity += 4194304;
                    }

                    return newCapacity;
                } else {
                    for(newCapacity = 64; newCapacity <
minNewCapacity; newCapacity <<= 1) {
                    }

                    return Math.min(newCapacity, maxCapacity);
                }
            }
        }
    }
```
扩容过程： 默认门限阈值为4MB(这个阈值是一个经验值，不同场景，可能取值不同)，当需要的容量等于门限阈值，使用阈值作为新的缓存区容量目标容量，如果大于阈值，采用每次步进4MB的方式进行内存扩张（需要扩容值/4MB)*4MB），扩张后需要和最大内存（maxCapacity）进行比较，大于maxCapacity的话就用maxCapacity,否则使用扩容值作为目标容量，如果小于阈值，采用倍增的方式，以64（字节）作为基本数值，每次翻倍增长64 -->128 --> 256，直到倍增后的结果大于或等于需要的容量值。