Netty

Netty 中重要的几个点如下：

- java 原生NIO
- 拆包与粘包
- Netty 内存池
- Netty 时间轮
- Netty EventLoop 模型
- Netty Handlelr
- Netty writeAndFlush 流程
- Netty 内存泄漏排查

Nio

- 零拷贝
- MappedByteBuffer
- 底层API
- Epoll poll select （Selector）
- PageBuffer
-

```
1、ChannelPool、ChannelPoolMap（Channel pool）
2、ObjectCleaner（添加删除）
3、PendingWrite（不重要）
4、DefaultPriorityQueue、RecyclableArrayList、ThreadLocalRandom
5、Recycler（对象池）
6、ResourceLeakDetector、ResourceLeakDetectorFactory（用于检测内存泄漏）
7、HashWheelTimer
8、ScheduledFutureTask
9、Promise、FastThreadLocal
10、EmbeddedChannel
```

关注点

客户端：

- SocketChannel 什么时候 注册到Selector的
- SocketChannel 什么时候注册OP_READ事件的

服务端

- ServerSocketChannel 什么时候 注册到Selector的
- ServerSocketChannel 是什么时候注册Accept 事件，并且接收SocketChannel的
- NioEventLoop 是什么工作的
- Pipeline的TailContext 与 HeadContext
- writeAndFlush的工作机制

协议

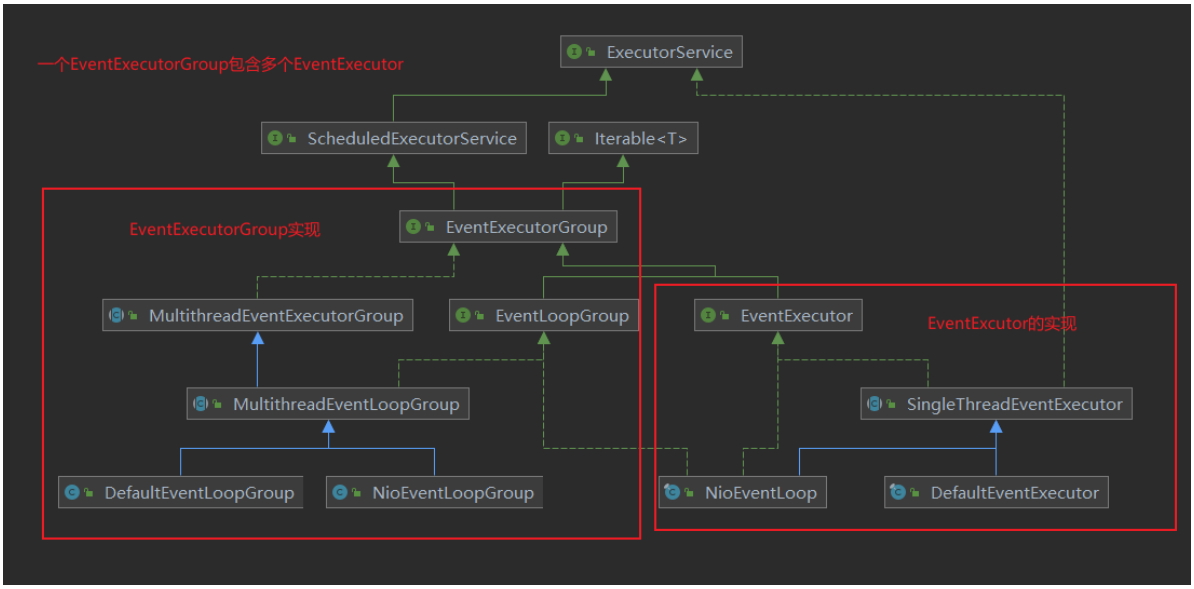
- Encoder 与 Decoder
- 协议序列化
- 拆包与粘包
- Netty 的事件机制
- Websocket 协议

Netty 核心

EventExecutor

- MultiThreadEventExecutor
- MultiThreadEventLoop
-
- `DefaultEventLoopGroup#newChild`
- NioEventLoopGroup # newChild

所有的EventLoop 都继承了SingleEvenetExecutor



NioEventLoop 的run方法

NioEventLoop 的run方法主要包括三个步骤，分别是select、processSelectedKeys 以及 runAllTasks 三部分

select 方法

select 方法主要是用于阻塞监听是否有对应的Select 事件

```
private void select(boolean oldWakenUp) throws IOException {
    Selector selector = this.selector;
    try {
        int selectCnt = 0;
        long currentTimeNanos = System.nanoTime();
        //计算select需要阻塞的时间，这里delayNanos方法是通过定时任务队列来进行计算的
        // 1、判断定时任务队列中是否有任务，如果有任务则返回第一个任务的delayTime
        // 2、如果定时任务队列没有定时任务，则返回默认delayTime 为 60s
        long selectDeadLineNanos = currentTimeNanos +
delayNanos(currentTimeNanos);

        for (;;) {
            //计算需要阻塞的延时时间
            long timeoutMillis = (selectDeadLineNanos - currentTimeNanos +
500000L) / 1000000L;

            //如果延时时间 <= 0 表示当前的定时任务已经可以执行了，直接通过
selector.selectNow查询底层事件，并且返回
            if (timeoutMillis <= 0) {
                if (selectCnt == 0) {
                    selector.selectNow();
                    selectCnt = 1;
                }
                break;
            }
```

```
            // If a task was submitted when wakenUp value was true, the task
didn't get a chance to call
            // Selector#wakeup. So we need to check task queue again before
executing select operation.
            // If we don't, the task might be pended until select operation was
timed out.
            // It might be pended until idle timeout if IdleStateHandler existed
in pipeline.
            // 判断任务队列中是否有相应的任务，如果存在相应的任务，则直接返回，这里会把wakeUp
的状态改为true
            if (hasTasks() && wakenUp.compareAndSet(false, true)) {
                selector.selectNow();
                selectCnt = 1;
                break;
            }

            //调用selector.select 进行阻塞，监听io事件
            int selectedKeys = selector.select(timeoutMillis);
            selectCnt ++;

            // selectedKeys !=0 表示有相应的io事件
            // oldWakenUp：表示
            // wakenUp.get()：由外部线程通过selector.wakeUp 进行唤醒
            // hasTasks：任务队列有任务
            // hasScheduledTasks：定时任务队列有定时任务
            if (selectedKeys != 0 || oldWakenUp || wakenUp.get() || hasTasks() ||
hasScheduledTasks()) {
                // - Selected something,
                // - waken up by user, or
                // - the task queue has a pending task.
                // - a scheduled task is ready for processing
                break;
            }

            //判断是否是线程中断引起的，如果是则直接返回
            if (Thread.interrupted()) {
                // Thread was interrupted so reset selected keys and break so we
not run into a busy loop.
                // As this is most likely a bug in the handler of the user or
it's client library we will
                // also log it.
                //
                // See https://github.com/netty/netty/issues/2426
                if (logger.isDebugEnabled()) {
                    logger.debug("Selector.select() returned prematurely because
" +
                            "Thread.currentThread().interrupt() was called. Use "
+
                            "NioEventLoop.shutdownGracefully() to shutdown the
NioEventLoop.");
                }
                selectCnt = 1;
                break;
            }
```

```
            // 获取当前的时间
             long time = System.nanoTime();

             //通过当前时间 - 阻塞时间 判断是否 >= 开始的时间，如果不成立表示selector是由于
nio底层空轮训bug引起的
             if (time - TimeUnit.MILLISECONDS.toNanos(timeoutMillis) >=
currentTimeNanos) {
                 // timeoutMillis elapsed without anything selected.
                 // 如果 判断成立则重置selectCnt
                 selectCnt = 1;
             }
             // 如果 selectCnt > 512 则重新创建selector，因为在nio底层的空轮训bug时，
selector会直接返回，
             // 那么每次执行selectCnt就是自增，直到selectCnt = 512 时，就会重置selector
             else if (SELECTOR_AUTO_REBUILD_THRESHOLD > 0 &&
                     selectCnt >= SELECTOR_AUTO_REBUILD_THRESHOLD) {
                 // The selector returned prematurely many times in a row.
                 // Rebuild the selector to work around the problem.
                 logger.warn(
                         "Selector.select() returned prematurely {} times in a
row; rebuilding Selector {}.",
                         selectCnt, selector);

                //重新创建一个新的selector
                 rebuildSelector();
                 selector = this.selector;

                 // Select again to populate selectedKeys.
                 selector.selectNow();
                 selectCnt = 1;
                 break;
             }

             currentTimeNanos = time;
         }

         if (selectCnt > MIN_PREMATURE_SELECTOR_RETURNS) {
             if (logger.isDebugEnabled()) {
                 logger.debug("Selector.select() returned prematurely {} times in
a row for Selector {}.",
                         selectCnt - 1, selector);
             }
         }
     } catch (CancelledKeyException e) {
         if (logger.isDebugEnabled()) {
             logger.debug(CancelledKeyException.class.getSimpleName() + " raised
by a Selector {} - JDK bug?",
                     selector, e);
         }
         // Harmless exception - log anyway
     }
}
```

processSelectedKeys

Channel

ServerSocketChannel

初始化

1、doBind

2、initAndRegister

3、

SocketChannel

初始化

1、processSelectKeys（）

2、unsafe.read()： 这里通过accept来获取连接

3、ServerBootstrapAcceptor：将获取的channel 注册到相应的selector上并且设置pipeline 以及相关属性

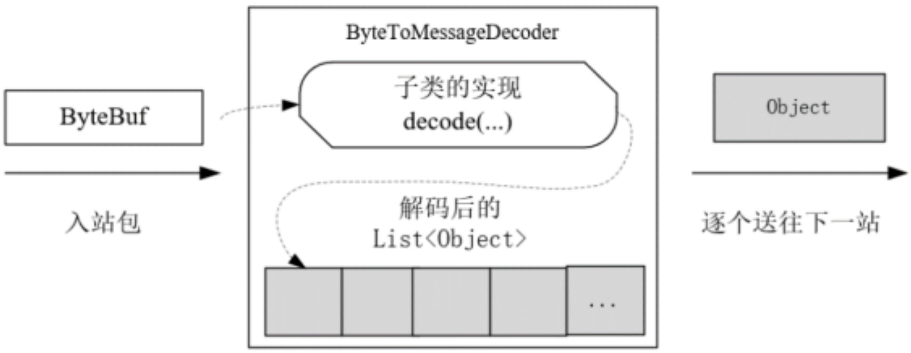
Handler

Decoder

ByteToMessageDecoder

ByteToMessageDecoder 提供了解码处理的基础逻辑和流程，ByteToMessageDecoder继承自ChannelInboundHandlerAdapter适配 器，是一个入站处理器，用于完成从ByteBuf到Java POJO对象的解码功能

ByteToMessageDecoder解码的流程，具体可以描述为：首先，它将上一站传过来的输 入到Bytebuf中的数据进行解码，解码出一个List对象列表；然后，迭代 List列表，逐个将Java POJO对象传入下一站Inbound入站处理器。大致如图7-1所 示。



ByteToMessageDecoder的解码方法名为decode，这是一个抽象方法，也就是说，decode方法中的具体解码过程，ByteToMessageDecoder没有具体的实现。那么，如何将Bytebuf中 的字节数据变成什么样的Object实例（包含多少个Object实例），需要子类去完成。所以 说，作为解码器的父类，ByteToMessageDecoder仅仅提供了一个整体框架：它会调用子类 的decode方法，完成具体的二进制字节解码，然后会获取子类解码之后的Object结果，放入 自己内部的结果列表List中，最终，父类会负责将List中的元素，一个一 个地传递给下一个站。从这个角度来说，ByteToMessageDecoder在设计上使用了模板模式 （Template Pattern）。

MessageToMessageDecoder

将一种POJO对象解码成另外一种POJO对象呢？答案是：存在的。只 不过与前面不同的是，在这种应用场景下的Decoder解码器，需要继承一个新的Netty解码 器基类：MessageToMessageDecoder

FixedLengthFrameDecoder

适用场景：每个接收到的数据包的长度，都是固定的，例如 100个字节。在这种场景 下，只需要把这个解码器加到流水线中，它会把入站ByteBuf数据包拆分成一个个长度为

100的数据包，然后发往下一个channelHandler入站处理器

LineBasedFrameDecoder

适用场景：每个ByteBuf数据包，使用换行符（或者回车换行符）作为数据包的边界分 割符。在这种场景下，只需要把这个LineBasedFrameDecoder解码器加到流水线中，Netty会 使用换行分隔符，把ByteBuf数据包分割成一个一个完整的应用层ByteBuf数据包，再发送 到下一站

DelimiterBasedFrameDecoder

DelimiterBasedFrameDecoder是LineBasedFrameDecoder按照行分割的通用版本。不同之 处在于，这个解码器更加灵活，可以自定义分隔符，而不是局限于换行符。如果使用这个 解码器，那么所接收到的数据包，末尾必须带上对应的分隔符

LengthFieldBasedFrameDecoder

这是一种基于灵活长度的解码器。在ByteBuf数据包中，加了一个长度域字段，保存了 原始数据包的长度。解码的时候，会按照这个长度进行原始数据包的提取。此解码器在所 有开箱即用解码器中是最为复杂的一种

LengthFieldPrepender

Relay

Encoder

MessageToByteEncoder

MessageToMessageEncoder

ChannelDuplexDecoder

ByteToMessageCodeC

IdleStateHandler

作用

- 心跳机制

心跳是在TCP长连接中，客户端和服务端定时向对方发送数据包通知对方自己还在线，保证连接的有效性的一种机制
在服务器和客户端之间一定时间内没有数据交互时, 即处于 idle 状态时, 客户端或服务器会发送一个特殊的数据包给对方, 当接收方收到这个数据报文后, 也立即发送一个特殊的数据报文, 回应发送方, 此即一个 PING-PONG 交互. 自然地, 当某一端收到心跳消息后, 就知道了对方仍然在线, 这就确保 TCP 连接的有效性.

- 心跳实现

使用TCP协议层的Keeplive机制，但是该机制默认的心跳时间是2小时，依赖操作系统实现不够灵活；应用层实现自定义心跳机制，比如Netty实现心跳机制；

参数

- readerIdleTime 读空闲超时时间设定，如果channelRead()方法超过readerIdleTime时间未被调用则会触发超时事件调用userEventTrigger()方法；
- writerIdleTime写空闲超时时间设定，如果write()方法超过writerIdleTime时间未被调用则会触发超时事件调用userEventTrigger()方法；
- allIdleTime所有类型的空闲超时时间设定，包括读空闲和写空闲；
- unit时间单位，包括时分秒等

应用

这里以Seata为例，Seata 分为服务端和客户端两部分，内部是通过Netty实现的RPC协议，为了保证服务连接可用，分别在客户端与服务端都添加了IdleStateHandler 用于保证连接存活

在Seata中读控线默认是写空闲的三倍，因为考虑到网络波动的原因，所以他这里增加了一个延迟容忍，写空闲默认是每 5s 触发一次

Seata Client 端如下：

```
public void userEventTriggered(ChannelHandlerContext ctx, Object evt) {
    if (evt instanceof IdleStateEvent) {
        IdleStateEvent idleStateEvent = (IdleStateEvent) evt;

        //如果是读空闲，则会关闭相应的连接，这里是通过它内置的连接池进行释放
        if (idleStateEvent.state() == IdleState.READER_IDLE) {
            if (LOGGER.isInfoEnabled()) {
                LOGGER.info("channel {} read idle.", ctx.channel());
            }
            try {
                String serverAddress =
NetUtil.toStringAddress(ctx.channel().remoteAddress());
                clientChannelManager.invalidateObject(serverAddress,
ctx.channel());
            } catch (Exception exx) {
              LOGGER.error(exx.getMessage());
            } finally {
              clientChannelManager.releaseChannel(ctx.channel(),
getAddressFromContext(ctx));
            }
        }

        //如果是触发写控线，这里会发送心跳PING 给服务端，服务端收到心跳信息后会返回PONG消息，
表示一次正常的心跳上报
        if (idleStateEvent == IdleStateEvent.WRITER_IDLE_STATE_EVENT) {
            try {
                 if (LOGGER.isDebugEnabled()) {
                    LOGGER.debug("will send ping msg,channel {}",
ctx.channel());
                 }

AbstractNettyRemotingClient.this.sendAsyncRequest(ctx.channel(),
HeartbeatMessage.PING);
            } catch (Throwable throwable) {
```

```
                    LOGGER.error("send request error: {}",
throwable.getMessage(), throwable);
                }
            }
        }
    }
```

Seata Server 端如下：

```
@Override
public void userEventTriggered(ChannelHandlerContext ctx, Object evt) {
      if (evt instanceof IdleStateEvent) {
         debugLog("idle:" + evt);
         IdleStateEvent idleStateEvent = (IdleStateEvent) evt;
         //如果是读空闲，那么就直接关闭channel
         if (idleStateEvent.state() == IdleState.READER_IDLE) {
            if (LOGGER.isInfoEnabled()) {
               LOGGER.info("channel:" + ctx.channel() + " read idle.");
            }
             handleDisconnect(ctx);
             try {
                 closeChannelHandlerContext(ctx);
             } catch (Exception e) {
               LOGGER.error(e.getMessage());
             }
         }
    }
}
```

TCP 与 Netty 参数

TCP 参数

SO_RCVBUF

Socket参数，TCP数据接收缓冲区大小。该缓冲区即TCP接收滑动窗口，linux操作系统可使用命令：cat /proc/sys/net/ipv4/tcp_rmem查询其大小。一般情况下，该值可由用户在任意时刻设置，但当设置值超过64KB时，需要在连接到远端之前设置。

SO_SNDBUF

Socket参数，TCP数据发送缓冲区大小。该缓冲区即TCP发送滑动窗口，linux操作系统可使用命令：cat /proc/sys/net/ipv4/tcp_smem查询其大小

TCP_NODELAY

TCP参数，立即发送数据，默认值为Ture（Netty默认为True而操作系统默认为False）。该值设置Nagle算法的启用，改算法将小的碎片数据连接成更大的报文来最小化所发送的报文的数量，如果需要发送一些较小的报文，则需要禁用该算法。Netty默认禁用该算法，从而最小化报文传输延时。

SO_KEEPALIVE

Socket参数，连接保活，默认值为False。启用该功能时，TCP会主动探测空闲连接的有效性。可以将此功能视为TCP的心跳机制，需要注意的是：默认的心跳间隔是7200s即2小时。Netty默认关闭该功能

SO_REUSEADDR

Socket参数，地址复用，默认值False。有四种情况可以使用：(1).当有一个有相同本地地址和端口的socket1处于TIME_WAIT状态时，而你希望启动的程序的socket2要占用该地址和端口，比如重启服务且保持先前端口。(2).有多块网卡或用IP Alias技术的机器在同一端口启动多个进程，但每个进程绑定的本地IP地址不能相同。(3).单个进程绑定相同的端口到多个socket上，但每个socket绑定的ip地址不同。(4).完全相同的地址和端口的重复绑定。但这只用于UDP的多播，不用于TCP

SO_LINGER

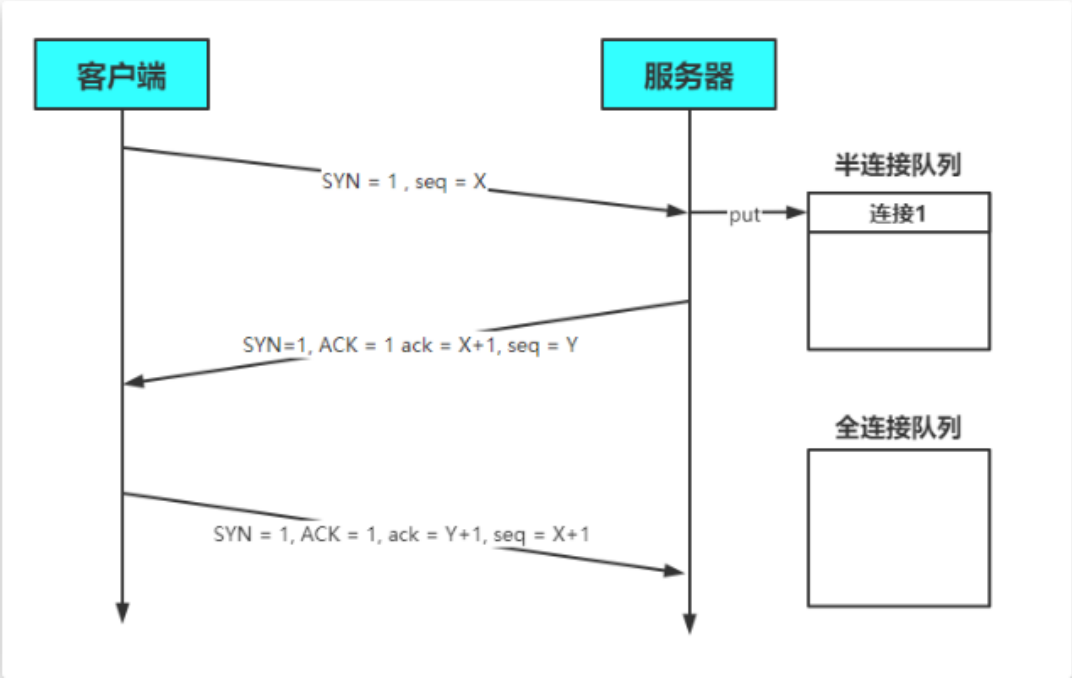
Socket参数，关闭Socket的延迟时间，默认值为-1，表示禁用该功能。-1表示socket.close()方法立即返回，但OS底层会将发送缓冲区全部发送到对端。0表示socket.close()方法立即返回，OS放弃发送缓冲区的数据直接向对端发送RST包，对端收到复位错误。非0整数值表示调用socket.close()方法的线程被阻塞直到延迟时间到或发送缓冲区中的数据发送完毕，若超时，则对端会收到复位错误

SO_BACKLOG

Socket参数，服务端接受连接的队列长度，如果队列已满，客户端连接将被拒绝。默认值，Windows为200，其他为128。

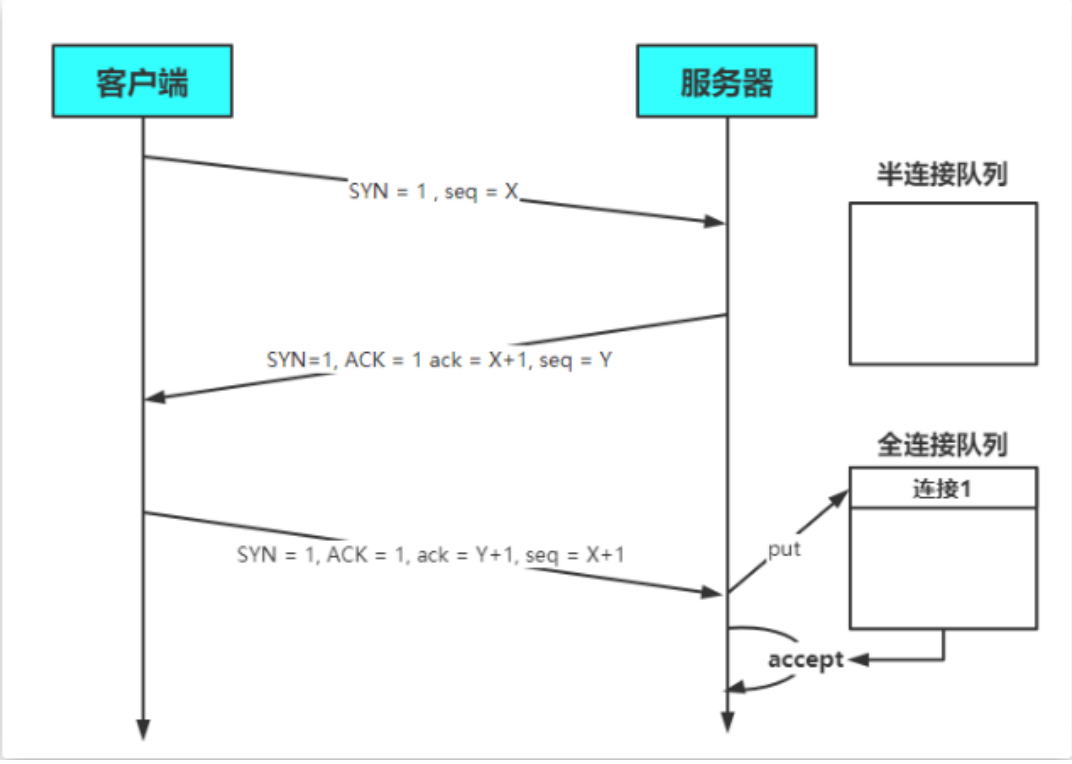
三次握手与连接队列

第一次握手时，因为客户端与服务器之间的连接还未完全建立，连接会被放入**半连接队列**中



当完成三次握手以后，连接会被放入**全连接队列中**

服务器处理Accept事件是在TCP三次握手，也就是建立连接之后。服务器会从全连接队列中获取连接并进行处理



在 linux 2.2 之前，backlog 大小包括了两个队列的大小，**在 linux 2.2 之后，分别用下面两个参数来控制**

- 半连接队列 - sync queue
  - 大小通过 /proc/sys/net/ipv4/tcp_max_syn_backlog 指定，在 `syncookies` 启用的情况下，逻辑上没有最大值限制，这个设置便被忽略
- 全连接队列 - accept queue
  - 其大小通过 /proc/sys/net/core/somaxconn 指定，在使用 listen 函数时，**内核会根据传入的 backlog 参数与系统参数，取二者的较小值**
  - 如果 accpet queue 队列满了，server 将发送一个拒绝连接的错误信息到 client

作用

在Netty中，`SO_BACKLOG`主要用于设置全连接队列的大小。**当处理Accept的速率小于连接建立的速率时，全连接队列中堆积的连接数大于`SO_BACKLOG`设置的值是，便会抛出异常**

SO_TIMEOUT

主要用在阻塞IO，因为阻塞IO的情况线程会一直阻塞，可以通过设置SO_TIMEOUT 参数，让其阻塞到指定时间后返回

UDP 参数

SO_BROADCAST

Socket参数，设置广播模式。

Netty 参数

ALLOW_HALF_CLOSURE

Netty参数，一个连接的远端关闭时本地端是否关闭，默认值为False。值为False时，连接自动关闭；为True时，触发ChannelInboundHandler的userEventTriggered()方法，事件为ChannelInputShutdownEvent

CONNECT_TIMEOUT_MILLIS

Netty参数，连接超时毫秒数，默认值30000毫秒即30秒。

- 属于 **SocketChannal** 的参数
- 用在**客户端建立连接**时，如果在指定毫秒内无法连接，会抛出 timeout 异常
- **注意**：Netty 中不要用成了SO_TIMEOUT 主要用在阻塞 IO，而 Netty 是非阻塞 IO

客户端中连接服务器的线程是 NIO 线程，抛出异常的是主线程。这是如何做到超时判断以及线程通信的呢?

`AbstractNioChannel.AbstractNioUnsafe.connect`方法中

```
public final void connect(
                final SocketAddress remoteAddress, final SocketAddress
localAddress, final ChannelPromise promise) {

    ...

    // Schedule connect timeout.
    // 设置超时时间，通过option方法传入的CONNECT_TIMEOUT_MILLIS参数进行设置
    int connectTimeoutMillis = config().getConnectTimeoutMillis();
    // 如果超时时间大于0
    if (connectTimeoutMillis > 0) {
```

```
        // 创建一个定时任务，延时connectTimeoutMillis（设置的超时时间时间）后执行
        // schedule(Runnable command, long delay, TimeUnit unit)
        connectTimeoutFuture = eventLoop().schedule(new Runnable() {
            @Override
            public void run() {
                // 判断是否建立连接，Promise进行NIO线程与主线程之间的通信
                // 如果超时，则通过tryFailure方法将异常放入Promise中
                // 在主线程中抛出
                ChannelPromise connectPromise =
AbstractNioChannel.this.connectPromise;
                ConnectTimeoutException cause = new
ConnectTimeoutException("connection timed out: " + remoteAddress);
                if (connectPromise != null && connectPromise.tryFailure(cause))
{
                    close(voidPromise());
                }
            }
        }, connectTimeoutMillis, TimeUnit.MILLISECONDS);
    }

    ...

}Copy
```

超时的判断**主要是通过 Eventloop 的 schedule 方法和 Promise 共同实现的**

- schedule 设置了一个定时任务，延迟`connectTimeoutMillis`秒后执行该方法
- 如果指定时间内没有建立连接，则会执行其中的任务
    - 任务负责创建 `ConnectTimeoutException` 异常，并将异常通过 Pormise 传给主线程并抛出

MAX_MESSAGES_PER_READ

Netty参数，一次Loop读取的最大消息数，对于ServerChannel或者NioByteChannel，默认值为16，其他Channel默认值为1。默认值这样设置，是因为：ServerChannel需要接受足够多的连接，保证大吞吐量，NioByteChannel可以减少不必要的系统调用select

WRITE_SPIN_COUNT

Netty参数，一个Loop写操作执行的最大次数，默认值为16。也就是说，对于大数据量的写操作至多进行16次，如果16次仍没有全部写完数据，此时会提交一个新的写任务给EventLoop，任务将在下次调度继续执行。这样，其他的写请求才能被响应不会因为单个大数据量写请求而耽误。

ALLOCATOR

Netty参数，ByteBuf的分配器，默认值为ByteBufAllocator.DEFAULT，4.0版本为UnpooledByteBufAllocator，4.1版本为PooledByteBufAllocator。该值也可以使用系统参数io.netty.allocator.type配置，使用字符串值："unpooled"，"pooled"

RCVBUF_ALLOCATOR

Netty参数，用于Channel分配接受Buffer的分配器，默认值为AdaptiveRecvByteBufAllocator.DEFAULT，是一个自适应的接受缓冲区分配器，能根据接受到的数据自动调节大小。可选值为FixedRecvByteBufAllocator，固定大小的接受缓冲区分配器

- 属于 **SocketChannal** 参数
- **控制 Netty 接收缓冲区大小**
- 负责入站数据的分配，决定入站缓冲区的大小（并可动态调整），**统一采用 direct 直接内存**，具体池化还是非池化由 allocator 决定

AUTO_READ

Netty参数，自动读取，默认值为True。Netty只在必要的时候才设置关心相应的I/O事件。对于读操作，需要调用channel.read()设置关心的I/O事件为OP_READ，这样若有数据到达才能读取以供用户处理。该值为True时，每次读操作完毕后会自动调用channel.read()，从而有数据到达便能读取；否则，需要用户手动调用channel.read()。需要注意的是：当调用config.setAutoRead(boolean)方法时，如果状态由false变为true，将会调用channel.read()方法读取数据；由true变为false，将调用config.autoReadCleared()方法终止数据读取

WRITE_BUFFER_HIGH_WATER_MARK

Netty参数，写高水位标记，默认值64KB。如果Netty的写缓冲区中的字节超过该值，Channel的isWritable()返回False

WRITE_BUFFER_LOW_WATER_MARK

Netty参数，写低水位标记，默认值32KB。当Netty的写缓冲区中的字节超过高水位之后若下降到低水位，则Channel的isWritable()返回True。写高低水位标记使用户可以控制写入数据速度，从而实现流量控制。推荐做法是：每次调用channl.write(msg)方法首先调用channel.isWritable()判断是否可写

MESSAGE_SIZE_ESTIMATOR

Netty参数，消息大小估算器，默认为DefaultMessageSizeEstimator.DEFAULT。估算ByteBuf、ByteBufHolder和FileRegion的大小，其中ByteBuf和ByteBufHolder为实际大小，FileRegion估算值为0。该值估算的字节数在计算水位时使用，FileRegion为0可知FileRegion不影响高低水位

SINGLE_EVENTEXECUTOR_PER_GROUP

Netty参数，单线程执行ChannelPipeline中的事件，默认值为True。该值控制执行ChannelPipeline中执行ChannelHandler的线程。如果为Trye，整个pipeline由一个线程执行，这样不需要进行线程切换以及线程同步，是Netty4的推荐做法；如果为False，ChannelHandler中的处理过程会由Group中的不同线程执行