16.4.2 AbstractUnsafe 源码分析

1. register 方法

register 方法主要用于将当前 Unsafe 对应的 Channel 注册到 EventLoop 的多路复用器上，然后调用 DefaultChannelPipeline 的 fireChannelRegistered 方法。如果 Channel 被激活，则调用 DefaultChannelPipeline 的 fireChannelActive 方法。源码如图 16-40 所示。

首先判断当前所在的线程是否是 Channel 对应的 NioEventLoop 线程，如果是同一个线程，则不存在多线程并发操作问题，直接调用 register0 进行注册；如果是由用户线程或者其他线程发起的注册操作，则将注册操作封装成 Runnable，放到 NioEventLoop 任务队列中执行。注意：如果直接执行 register0 方法，会存在多线程并发操作 Channel 的问题。

```
@Override
    public final void register(final ChannelPromise promise) {
        if (eventLoop.inEventLoop()) {
            register0(promise);
        } else {
            try {
                eventLoop.execute(new Runnable() {
                    @Override
                    public void run() {
                        register0(promise);
                    }
                });
            } catch (Throwable t) {
                logger.warn(
                        "Force-closing a channel whose registration task was not
accepted by an event loop: {}",
                        AbstractChannel.this, t);
                closeForcibly();
                closeFuture.setClosed();
                promise.setFailure(t);
            }
        }
    }
```

图 16-40　AbstractUnsafe 的 register 方法

下面继续看 register0 方法的实现，代码如图 16-41 所示。

首先调用 ensureOpen 方法判断当前 Channel 是否打开，如果没有打开则无法注册，直接返回。校验通过后调用 doRegister 方法，它由 AbstractNioUnsafe 对应的 AbstractNioChannel 实现，代码如图 16-42 所示。

```
private void register0(ChannelPromise promise) {
        try {
            // check if the channel is still open as it could be closed in the mean
time when the register
            // call was outside of the eventLoop
            if (!ensureOpen(promise)) {
                return;
            }
            doRegister();
            registered = true;
            promise.setSuccess();
            pipeline.fireChannelRegistered();
            if (isActive()) {
                pipeline.fireChannelActive();
            }
        } catch (Throwable t) {
            // Close the channel directly to avoid FD leak.
            closeForcibly();
            closeFuture.setClosed();
            if (!promise.tryFailure(t)) {
                logger.warn(
                        "Tried to fail the registration promise, but it is complete
already. " +
                                "Swallowing the cause of the registration failure:", t);
            }

        }

    }
```

图 16-41　AbstractUnsafe 的 register0 方法

```
@Override
protected void doRegister() throws Exception {
    boolean selected = false;
    for (;;) {
        try {
            selectionKey = javaChannel().register(eventLoop().selector, 0, this);
            return;
        } catch (CancelledKeyException e) {
            if (!selected) {
                // Force the Selector to select now as the "canceled" SelectionKey
may still be
                // cached and not removed because no Select.select(..) operation
was called yet.
                eventLoop().selectNow();
                selected = true;
            } else {
                // We forced a select operation on the selector before but the
SelectionKey is still cached
                // for whatever reason. JDK bug ?
                throw e;
            }
        }
    }
}
```

图 16-42　AbstractNioChannel 的 doRegister 方法

该方法在前面的 AbstractNioChannel 源码分析中已经介绍过，此处不再赘述。如果 doRegister 方法没有抛出异常，则说明 Channel 注册成功。将 ChannelPromise 的结果设置为成功，调用 ChannelPipeline 的 fireChannelRegistered 方法，判断当前的 Channel 是否已

经被激活，如果已经被激活，则调用 ChannelPipeline 的 fireChannelActive 方法。

如果注册过程中发生了异常，则强制关闭连接，将异常堆栈信息设置到 ChannelPromise 中。

2. bind 方法

bind 方法主要用于绑定指定的端口，对于服务端，用于绑定监听端口，可以设置 backlog 参数；对于客户端，主要用于指定客户端 Channel 的本地绑定 Socket 地址。代码实现如图 16-43 所示。

```
boolean wasActive = isActive();
        try {
            doBind(localAddress);
        } catch (Throwable t) {
            promise.setFailure(t);
            closeIfClosed();
            return;
        }
        if (!wasActive && isActive()) {
            invokeLater(new Runnable() {
                @Override
                public void run() {
                    pipeline.fireChannelActive();
                }
            });
        }
        promise.setSuccess();
```

图 16-43　AbstractUnsafe 的 bind 方法实现

调用 doBind 方法，对于 NioSocketChannel 和 NioServerSocketChannel 有不同的实现，客户端的实现代码如图 16-44 所示。

```
@Override
protected void doBind(SocketAddress localAddress) throws Exception {
    javaChannel().socket().bind(localAddress);
}
```

图 16-44　NioSocketChannel 的 doBind 方法实现

服务端的 doBind 方法实现如图 16-45 所示。

```
@Override
protected void doBind(SocketAddress localAddress) throws Exception {
    javaChannel().socket().bind(localAddress, config.getBacklog());
}
```

图 16-45　NioServerSocketChannel 的 doBind 方法实现

如果绑定本地端口发生异常，则将异常设置到 ChannelPromise 中用于通知 ChannelFuture，随后调用 closeIfClosed 方法来关闭 Channel。

3. disconnect 方法

disconnect 用于客户端或者服务端主动关闭连接，它的代码如图 16-46 所示。

```
@Override
public final void disconnect(final ChannelPromise promise) {
    boolean wasActive = isActive();
    try {
        doDisconnect();
    } catch (Throwable t) {
        promise.setFailure(t);
        closeIfClosed();
        return;
    }
    if (wasActive && !isActive()) {
        invokeLater(new Runnable() {
            @Override
            public void run() {
                pipeline.fireChannelInactive();
            }
        });
    }
    promise.setSuccess();
    closeIfClosed(); // doDisconnect() might have closed the channel
}
```

图 16-46　AbstractUnsafe 的 disconnect 方法实现

4. close 方法

在链路关闭之前需要首先判断是否处于刷新状态，如果处于刷新状态说明还有消息尚未发送出去，需要等到所有消息发送完成再关闭链路，因此，将关闭操作封装成 Runnable 稍后再执行。如图 16-47 所示。

```
@Override
public final void close(final ChannelPromise promise) {
    if (inFlush0) {
        invokeLater(new Runnable() {
            @Override
            public void run() {
                close(promise);
            }
        });
        return;
    }
    if (closeFuture.isDone()) {
        // Closed already.
        promise.setSuccess();
        return;
    }
```

图 16-47　AbstractUnsafe 的 close 方法片段 1

如果链路没有处于刷新状态，需要从 closeFuture 中判断关闭操作是否完成，如果已经完成，不需要重复关闭链路，设置 ChannelPromise 的操作结果为成功并返回。

执行关闭操作，将消息发送缓冲数组设置为空，通知 JVM 进行内存回收。调用抽象方法 doClose 关闭链路。源码如图 16-48 所示。

```
    boolean wasActive = isActive();
        ChannelOutboundBuffer outboundBuffer = this.outboundBuffer;
        this.outboundBuffer = null; // Disallow adding any messages and
flushes to outboundBuffer.
        try {
           doClose();
           closeFuture.setClosed();
           promise.setSuccess();
        } catch (Throwable t) {
           closeFuture.setClosed();
           promise.setFailure(t);
        }
```

图 16-48　AbstractUnsafe 的 close 方法片段 2

如果关闭操作成功，设置 ChannelPromise 结果为成功。如果操作失败，则设置异常对象到 ChannelPromise 中。

调用 ChannelOutboundBuffer 的 close 方法释放缓冲区的消息，随后构造链路关闭通知 Runnable 放到 NioEventLoop 中执行。源码如图 16-49 所示。

```
// Fail all the queued messages
    try {
       outboundBuffer.failFlushed(CLOSED_CHANNEL_EXCEPTION);
       outboundBuffer.close(CLOSED_CHANNEL_EXCEPTION);
    } finally {
       if (wasActive && !isActive()) {
          invokeLater(new Runnable() {
             @Override
             public void run() {
                pipeline.fireChannelInactive();
             }
          });
       }
       deregister();
    }
```

图 16-49　AbstractUnsafe 的 close 方法片段 3

最后，调用 deregister 方法，将 Channel 从多路复用器上取消注册，代码实现如图 16-50 所示。

```
    @Override
    protected void doDeregister() throws Exception {
       eventLoop().cancel(selectionKey());
    }
```

图 16-50　AbstractUnsafe 的 close 方法片段 4

NioEventLoop 的 cancel 方法实际将 selectionKey 对应的 Channel 从多路复用器上去注册，NioEventLoop 的相关代码如图 16-51 所示。

```
void cancel(SelectionKey key) {
    key.cancel();
    cancelledKeys ++;
    if (cancelledKeys >= CLEANUP_INTERVAL) {
        cancelledKeys = 0;
        needsToSelectAgain = true;
    }
}
```

图 16-51　取消 Channel 的注册

5. write 方法

write 方法实际上将消息添加到环形发送数组中，并不是真正的写 Channel，它的代码如图 16-52 所示。

```
@Override
public void write(Object msg, ChannelPromise promise) {
    if (!isActive()) {
        // Mark the write request as failure if the channel is inactive.
        if (isOpen()) {
            promise.tryFailure(NOT_YET_CONNECTED_EXCEPTION);
        } else {
            promise.tryFailure(CLOSED_CHANNEL_EXCEPTION);
        }
        // release message now to prevent resource-leak
        ReferenceCountUtil.release(msg);
    } else {
        outboundBuffer.addMessage(msg, promise);
    }
}
```

图 16-52　写操作

如果 Channel 没有处于激活状态，说明 TCP 链路还没有真正建立成功，当前 Channel 存在以下两种状态。

（1）Channel 打开，但是 TCP 链路尚未建立成功：NOT_YET_CONNECTED_EXCEPTION；

（2）Channel 已经关闭：CLOSED_CHANNEL_EXCEPTION。

对链路状态进行判断，给 ChannelPromise 设置对应的异常，然后调用 ReferenceCountUtil 的 release 方法释放发送的 msg 对象。

如果链路状态正常，则将需要发送的 msg 和 promise 放入发送缓冲区中（环形数组）。

6. flush 方法

flush 方法负责将发送缓冲区中待发送的消息全部写入到 Channel 中，并发送给通信对方。它的代码如图 16-53 所示。

```
@Override
public void flush() {
    ChannelOutboundBuffer outboundBuffer = this.outboundBuffer;
    if (outboundBuffer == null) {
        return;
    }
    outboundBuffer.addFlush();
    flush0();
}
```

图 16-53　刷新操作

首先将发送环形数组的 unflushed 指针修改为 tail，标识本次要发送消息的缓冲区范围。然后调用 flush0 进行发送，由于 flush0 代码非常简单，我们重点分析 doWrite 方法，代码如图 16-54 所示。

```
@Override
protected void doWrite(ChannelOutboundBuffer in) throws Exception {
    for (;;) {
        // Do non-gathering write for a single buffer case.
        final int msgCount = in.size();
        if (msgCount <= 1) {
            super.doWrite(in);
            return;
        }
```

图 16-54　doWrite 方法代码片段 1

首先计算需要发送的消息个数（unflushed - flush），如果只有 1 个消息需要发送，则调用父类的写操作，我们分析 AbstractNioByteChannel 的 doWrite()方法，代码如图 16-55 所示。

```
@Override
protected void doWrite(ChannelOutboundBuffer in) throws Exception {
    int writeSpinCount = -1;
    for (;;) {
        Object msg = in.current(true);
        if (msg == null) {
            // Wrote all messages.
            clearOpWrite();
            break;
        }
```

图 16-55　doWrite 方法代码片段 2

因为只有一条消息需要发送，所以直接从 ChannelOutboundBuffer 中获取当前需要发送的消息，代码如图 16-56 所示。

```
public Object current(boolean preferDirect) {
    if (isEmpty()) {
        return null;
    } else {
        Object msg = buffer[flushed].msg;
        if (threadLocalDirectBufferSize <= 0 || !preferDirect) {
            return msg;
        }
        if (msg instanceof ByteBuf) {
            ByteBuf buf = (ByteBuf) msg;
            if (buf.isDirect()) {
                return buf;
            } else {
                int readableBytes = buf.readableBytes();
                if (readableBytes == 0) {
                    return buf;
                }
                // Non-direct buffers are copied into JDK's own internal direct
buffer on every I/O.
                // We can do a better job by using our pooled allocator. If the
current allocator does not
                // pool a direct buffer, we use a ThreadLocal based pool.
                ByteBufAllocator alloc = channel.alloc();
                ByteBuf directBuf;
                if (alloc.isDirectBufferPooled()) {
                    directBuf = alloc.directBuffer(readableBytes);
                } else {
                    directBuf = ThreadLocalPooledByteBuf.newInstance();
                }
```

图 16-56　doWrite 方法代码片段 3

首先，获取需要发送的消息，如果消息为 ByteBuf 且它分配的是 JDK 的非堆内存，则直接返回。对返回的消息进行判断，如果为空，说明该消息已经发送完成并被回收，然后执行清空 OP_WRITE 操作位的 clearOpWrite 方法，代码如图 16-57 所示。

```
protected final void clearOpWrite() {
    final SelectionKey key = selectionKey();
    final int interestOps = key.interestOps();
    if ((interestOps & SelectionKey.OP_WRITE) != 0) {
        key.interestOps(interestOps & ~SelectionKey.OP_WRITE);
    }
}
```

图 16-57　doWrite 方法代码片段 4

继续向下分析，如果需要发送的 ByteBuf 已经没有可写的字节了，则说明已经发送完成，将该消息从环形队列中删除，然后继续循环，代码如图 16-58 所示。

```
if (msg instanceof ByteBuf) {
            ByteBuf buf = (ByteBuf) msg;
            int readableBytes = buf.readableBytes();
            if (readableBytes == 0) {
                in.remove();
                continue;
            }
```

图 16-58　doWrite 方法代码片段 5

下面我们分析下 ChannelOutboundBuffer 的 remove 方法，如图 16-59 所示。

```
public boolean remove() {
    if (isEmpty()) {
        return false;
    }
    Entry e = buffer[flushed];
    Object msg = e.msg;
    if (msg == null) {
        return false;
    }
    ChannelPromise promise = e.promise;
    int size = e.pendingSize;
    e.clear();
    flushed = flushed + 1 & buffer.length - 1;
    safeRelease(msg);
    promise.trySuccess();
    decrementPendingOutboundBytes(size);
    return true;
}
```

图 16-59　doWrite 方法代码片段 6

首先判断环形队列中是否还有需要发送的消息，如果没有，则直接返回。如果非空，则首先获取 Entry，然后对其进行资源释放，同时对需要发送的索引 flushed 进行更新。所有操作执行完之后，调用 decrementPendingOutboundBytes 减去已经发送的字节数，该方法跟 incrementPendingOutboundBytes 类似，会进行发送低水位的判断和事件通知，此处不再赘述。

我们接着继续对消息的发送进行分析，代码如图 16-60 所示。

```
boolean setOpWrite = false;
        boolean done = false;
        long flushedAmount = 0;
        if (writeSpinCount == -1) {
            writeSpinCount = config().getWriteSpinCount();
        }
        for (int i = writeSpinCount - 1; i >= 0; i --) {
            int localFlushedAmount = doWriteBytes(buf);
            if (localFlushedAmount == 0) {
                setOpWrite = true;
                break;
            }
            flushedAmount += localFlushedAmount;
            if (!buf.isReadable()) {
                done = true;
                break;
            }
        }
```

第一步：对写入的字节个数进行判读，如果为 0 说明 TCP 的发送缓冲已满，需要退出并监听写操作

图 16-60　doWrite 方法代码片段 7

首先将半包标识设置为 false，从 DefaultSocketChannelConfig 中获取循环发送的次数，进行循环发送，对发送方法 doWriteBytes 展开分析，如图 16-61 所示。

```
@Override
protected int doWriteBytes(ByteBuf buf) throws Exception {
    final int expectedWrittenBytes = buf.readableBytes();
    final int writtenBytes = buf.readBytes(javaChannel(),
expectedWrittenBytes);
    return writtenBytes;
}
```

图 16-61　doWrite 方法代码片段 8

ByteBuf 的 readBytes()方法的功能是将当前 ByteBuf 中的可写字节数组写入到指定的 Channel 中。方法的第一个参数是 Channel，此处就是 SocketChannel，第二个参数是写入的字节数组长度，它等于 ByteBuf 的可读字节数，返回值是写入的字节个数。由于我们将 SocketChannel 设置为异步非阻塞模式，所以写操作不会阻塞。

从写操作中返回，需要对写入的字节数进行判断，如果为 0，说明 TCP 发送缓冲区已满，不能继续再向里面写入消息，因此，将写半包标识设置为 true，然后退出循环，执行后续排队的其他任务或者读操作，等待下一次 selector 的轮询继续触发写操作。

对写入的字节数进行累加，判断当前的 ByteBuf 中是否还有没有发送的字节，如果没有可发送的字节，则将 done 设置为 true，退出循环。

从循环发送状态退出后，首先根据实际发送的字节数更新发送进度，实际就是发送的字节数和需要发送的字节数的一个比值。执行完进度更新后，判断本轮循环是否将需要发送的消息全部发送完成，如果发送完成则将该消息从循环队列中删除；否则，设置多路复用器的 OP_WRITE 操作位，用于通知 Reactor 线程还有半包消息需要继续发送。

16.4.3　AbstractNioUnsafe 源码分析

AbstractNioUnsafe 是 AbstractUnsafe 类的 NIO 实现，它主要实现了 connect、finishConnect 等方法，下面我们对重点 API 实现进行源码分析。

1. connect 方法

首先获取当前的连接状态进行缓存，然后发起连接操作，代码如图 16-62 所示。

需要指出的是，SocketChannel 执行 connect()操作有三种可能的结果。

（1）连接成功，返回 true；

（2）暂时没有连接上，服务端没有返回 ACK 应答，连接结果不确定，返回 false；

```
@Override
protected boolean doConnect(SocketAddress remoteAddress,
SocketAddress localAddress) throws Exception {
    if (localAddress != null) {
        javaChannel().socket().bind(localAddress);
    }
    boolean success = false;
    try {
        boolean connected = javaChannel().connect(remoteAddress);
        if (!connected) {
            selectionKey().interestOps(SelectionKey.OP_CONNECT);
        }
        success = true;
        return connected;
    } finally {
        if (!success) {
            doClose();
        }
    }
}
```

如果指定了本地绑定端口，执行绑定操作发起异步 TCP 连接，可能连接成功，也可能暂时没有连接成功。如果没有立即连接成功，则监听连接操作

图 16-62　AbstractNioUnsafe 的 connect 方法代码片段 1

（3）连接失败，直接抛出 I/O 异常。

如果是第（2）种结果，需要将 NioSocketChannel 中的 selectionKey 设置为 OP_ CONNECT，监听连接应答消息。

异步连接返回之后，需要判断连接结果，如果连接成功，则触发 ChannelActive 事件，代码如图 16-63 所示。

```
private void fulfillConnectPromise(ChannelPromise promise, boolean
wasActive) {
        // trySuccess() will return false if a user cancelled the connection attempt.
        boolean promiseSet = promise.trySuccess();
        // Regardless if the connection attempt was cancelled, channelActive()
event should be triggered,
        // because what happened is what happened.
        if (!wasActive && isActive()) {
            pipeline().fireChannelActive();
        }
        // If a user cancelled the connection attempt, close the channel, which is
followed by channelInactive().
        if (!promiseSet) {
            close(voidPromise());
        }
    }
```

图 16-63　AbstractNioUnsafe 的 connect 方法代码片段 2

这里对 ChannelActive 事件处理不再进行详细说明，它最终会将 NioSocketChannel 中的 selectionKey 设置为 SelectionKey.OP_READ，用于监听网络读操作位。

如果没有立即连接上服务端，则执行如图 16-64 所示分支。

```
// Schedule connect timeout.
            int connectTimeoutMillis = config().getConnectTimeoutMillis();
            if (connectTimeoutMillis > 0) {
                connectTimeoutFuture = eventLoop().schedule(new Runnable() {
                    @Override
                    public void run() {
                        ChannelPromise connectPromise =
AbstractNioChannel.this.connectPromise;
                        ConnectTimeoutException cause =
                                new ConnectTimeoutException("connection timed out:
" + remoteAddress);
                        if (connectPromise != null &&
connectPromise.tryFailure(cause)) {
                            close(voidPromise());
                        }
                    }
                }, connectTimeoutMillis, TimeUnit.MILLISECONDS);
            }
            promise.addListener(new ChannelFutureListener() {
                @Override
                public void operationComplete(ChannelFuture future) throws
Exception {
                    if (future.isCancelled()) {
                        if (connectTimeoutFuture != null) {
                            connectTimeoutFuture.cancel(false);
                        }
                        connectPromise = null;
                        close(voidPromise());
                    }
                }
            });
```

图 16-64　AbstractNioUnsafe 的 connect 方法代码片段 3

上面的操作有两个目的。

（1）根据连接超时时间设置定时任务，超时时间到之后触发校验，如果发现连接并没有完成，则关闭连接句柄，释放资源，设置异常堆栈并发起去注册。

（2）设置连接结果监听器，如果接收到连接完成通知则判断连接是否被取消，如果被取消则关闭连接句柄，释放资源，发起取消注册操作。

2. finishConnect 方法

客户端接收到服务端的 TCP 握手应答消息，通过 SocketChannel 的 finishConnect 方法对连接结果进行判断，代码如图 16-65 所示。

```
@Override
    public void finishConnect() {
        // Note this method is invoked by the event loop only if the connection
attempt was
        // neither cancelled nor timed out.
        assert eventLoop().inEventLoop();
        assert connectPromise != null;
        try {
            boolean wasActive = isActive();
            doFinishConnect();
```

图 16-65　AbstractNioUnsafe 的 finishConnect 方法代码片段 1

首先缓存连接状态，当前返回 false，然后执行 doFinishConnect 方法判断连接结果，代码如图 16-66 所示。

```
@Override
  protected void doFinishConnect() throws Exception {
    if (!javaChannel().finishConnect()) {
      throw new Error();
    }
  }
```

图 16-66　AbstractNioUnsafe 的 finishConnect 方法代码片段 2

通过 SocketChannel 的 finishConnect 方法判断连接结果，执行该方法返回三种可能结果。

◎ 连接成功返回 true；

◎ 连接失败返回 false；

◎ 发生链路被关闭、链路中断等异常，连接失败。

只要连接失败，就抛出 Error()，由调用方执行句柄关闭等资源释放操作，如果返回成功，则执行 fulfillConnectPromise 方法，它负责将 SocketChannel 修改为监听读操作位，用来监听网络的读事件，代码如图 16-67 所示。

```
  private void fulfillConnectPromise(ChannelPromise promise, boolean
wasActive) {
        // trySuccess() will return false if a user cancelled the connection attempt.
        boolean promiseSet = promise.trySuccess();
        // Regardless if the connection attempt was cancelled, channelActive()
event should be triggered,
        // because what happened is what happened.
        if (!wasActive && isActive()) {
            pipeline().fireChannelActive();
        }
```

图 16-67　AbstractNioUnsafe 的 fulfillConnectPromise 方法

最后对连接超时进行判断：如果连接超时时仍然没有接收到服务端的 ACK 应答消息，则由定时任务关闭客户端连接，将 SocketChannel 从 Reactor 线程的多路复用器上摘除，释放资源，代码如图 16-68 所示。

```
} finally {
            // Check for null as the connectTimeoutFuture is only created if a
connectTimeoutMillis > 0 is used
            // See https://github.com/netty/netty/issues/1770
            if (connectTimeoutFuture != null) {
                connectTimeoutFuture.cancel(false);
            }
            connectPromise = null;
        }
```

图 16-68　AbstractNioUnsafe 的 finishConnect 方法代码片段 3

16.4.4　NioByteUnsafe 源码分析

我们重点分析它的 read 方法，源码如图 16-69 所示。

```
@Override
        public void read() {
            final ChannelConfig config = config();
            final ChannelPipeline pipeline = pipeline();
            final ByteBufAllocator allocator = config.getAllocator();
            final int maxMessagesPerRead = config.getMaxMessagesPerRead();
            RecvByteBufAllocator.Handle allocHandle = this.allocHandle;
            if (allocHandle == null) {
                this.allocHandle = allocHandle =
config.getRecvByteBufAllocator().newHandle();
            }
```

图 16-69　NioByteUnsafe 的 read 方法代码片段 1

首先，获取 NioSocketChannel 的 SocketChannelConfig，它主要用于设置客户端连接的 TCP 参数，接口如图 16-70 所示。

继续看 allocHandle 的初始化。如果是首次调用，从 SocketChannelConfig 的 RecvByteBufAllocator 中创建 Handle。下面我们对 RecvByteBufAllocator 进行简单地代码分析：RecvByteBufAllocator 默认有两种实现，分别是 AdaptiveRecvByteBufAllocator 和 FixedRecvByteBufAllocator。由于 FixedRecvByteBufAllocator 的实现比较简单，我们重点分析 AdaptiveRecvByteBufAllocator 的实现。如图 16-71 所示。

顾名思义，AdaptiveRecvByteBufAllocator 指的是缓冲区大小可以动态调整的 ByteBuf 分配器。它的成员变量定义如图 16-72 所示。

图 16-70　SocketChannelConfig 的 API 列表

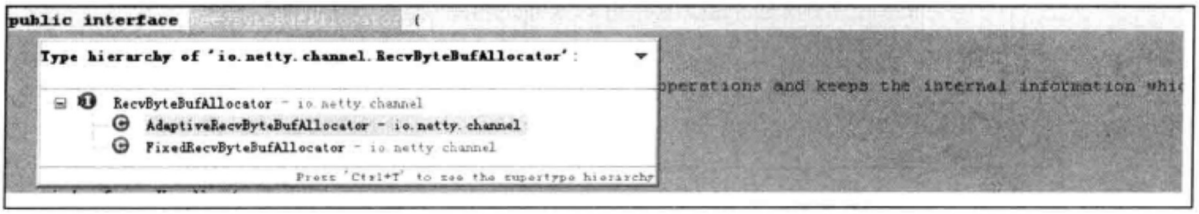

图 16-71　RecvByteBufAllocator 接口的继承关系

```
public class AdaptiveRecvByteBufAllocator implements RecvByteBufAllocator {
        static final int DEFAULT_MINIMUM = 64;
        static final int DEFAULT_INITIAL = 1024;

        static final int DEFAULT_MAXIMUM = 65536;
        private static final int INDEX_INCREMENT = 4;
        private static final int INDEX_DECREMENT = 1;
```

图 16-72　RecvByteBufAllocator 的成员变量定义

它分别定义了三个系统默认值：最小缓冲区长度 64 字节、初始容量 1024 字节、最大容量 65536 字节。还定义了两个动态调整容量时的步进参数：扩张的步进索引为 4、收缩的步进索引为 1。