提纲

将GridIoManager、TcpCommunicationSpi和GridNioServer串联起来

GridIoManager在启动过程中，主要做了一下操作：

- 启动TcpCommunicationSpi；
- 为TcpCommunicationSpi设置一个CommunicationListener，在设置的CommunicationListener中定义了两个方法：onMessage和onDisconnected，分别用于处理接收到的消息和处理连接断开事件；
- 初始化MessageFormatter，如果没有定义扩展的MessageFormatter，则使用默认的MessageFormatter，在默认的MessageFormatter中会分别定义writer和reader方法;
- 初始化MessageFactory，使用可能存在的扩展的MessageFactory和Ignite为某些IgniteComponentType中定义的特殊的MessageFactory一并作为扩展的MessageFactory来初始化GridIoMessageFactory.ext，在GridIoMessageFactory中，当需要创建某个类型的消息的时候，根据消息类型，优先查找这些消息类型是否是内置的消息类型，并调用内置的方法来创建消息实例，否则去查找是否是GridIoMessageFactory.ext中定义的消息类型，并调用扩展的MessageFactory来创建消息。

GridIoManager中提供了以下几类方法：

- sendToGridTopic，sendOrderedMessage，sendToCustomTopic和sendOrderedMessageToGridTopic等，前面2个方法用于将消息发送给ignite内部的topic，而后面2个方法则用于将消息发送给用户自定义的topic。
  - 这些方法多会被其它的GridManager所调用，用于进行消息投递。
  - 这些方法最终都是调用GridIoManager.send()方法来完成消息投递。
    - 根据传递过来的Message生成GridIoMessage。
    - 如果本地节点就是消息的目标节点，则根据是否要求ordered，是否是async发送，分别调用processOrderedMessage或者processRegularMessage或者processRegularMessage0进行处理。
    - 如果本地节点不是消息的目标节点，则调用TcpCommunicationSpi.sendMessage进行发送。
- addDisconnectListener，addMessageListener，addUserMessageListener分别用于订阅某个topic的消息。
- removeDisconnectListener，removeMessageListener，removeUserMessageListener分别用于取消订阅某个topic的消息。
- checkNodeLeft，检查给定的node是否离开了集群拓扑，主要供GridCacheIoManager使用。
- formatter，messageFactory分别用于获取MessageFormatter和MessageFactory，这两者都将在GridManagerAdapter中为各SPIs设置IgniteSpiContext时使用。

前面提到，在GridIoManager启动过程中会为TcpCommunicationSpi设置一个CommunicationListener，并在该CommunicationListener中定义了两个方法：onMessage和onDisconnected。

```java
public class GridIoManager extends GridManagerAdapter<CommunicationSpi<Serializable>> {
    @Override public void start() throws IgniteCheckedException {
        ...

        getSpi().setListener(commLsnr = new CommunicationListener<Serializable>() {
            @Override public void onMessage(UUID nodeId, Serializable msg, IgniteRunnable msgC) {
                try {
                    onMessage0(nodeId, (GridIoMessage)msg, msgC);
                }
                catch (ClassCastException ignored) {
                    U.error(log, "Communication manager received message of unknown type (will
ignore): " +
                        msg.getClass().getName() + ". Most likely GridCommunicationSpi is being
used directly, " +
                        "which is illegal - make sure to send messages only via GridProjection
API.");
                }
            }

            @Override public void onDisconnected(UUID nodeId) {
                for (GridDisconnectListener lsnr : disconnectLsnrs)
                    lsnr.onNodeDisconnected(nodeId);
            }
        });

        ...
    }
}
```

从上面的代码中可以看出，系统会首先创建一个CommunicationListener，然后将这个CommunicationListener赋值给GridIoManager.commLsnr，然后会将GridIoManager.commLsnr通过TcpCommunicationSpi.setListener赋值给TcpCommunicationSpi.lsnr。那么这个CommunicationListener.onMessage方法就有2个地方可以调用了，分别是通过GridIoManager.commLsnr.onMessage和TcpCommunicationSpi.lsnr.onMessage。

查找代码发现，GridIoManager.commLsnr.onMessage只在GridIoManager.onKernalStart中会被调用，主要用于delayed messages的处理。这里暂不深究。而TcpCommunicationSpi.lsnr.onMessage在TcpCommunicationSpi中进行了封装：

```java
public class TcpCommunicationSpi extends IgniteSpiAdapter implements CommunicationSpi<Message> {
    protected void notifyListener(UUID sndId, Message msg, IgniteRunnable msgC) {
        CommunicationListener<Message> lsnr = this.lsnr;

        if (lsnr != null)
            // Notify listener of a new message.
            lsnr.onMessage(sndId, msg, msgC);
        else if (log.isDebugEnabled())
            log.debug("Received communication message without any registered listeners (will
ignore, " +
                "is node stopping?) [senderNodeId=" + sndId + ", msg=" + msg + ']');
    }
}
```

那么TcpCommunicationSpi.notifyListener又在那些地方被调用呢？主要有两个地方，一个地方是在TcpCommunicationSpi.sendMessage0(该方法被TcpCommunicationSpi.sendMessage调用)中，当消息的目标节点就是本地节点时，通过notifyListener来通知CommunicationListener接收到了消息：

```java
public class TcpCommunicationSpi extends IgniteSpiAdapter implements CommunicationSpi<Message> {
    private void sendMessage0(ClusterNode node, Message msg, IgniteInClosure<IgniteException>
ackC)
        throws IgniteSpiException {
        assert node != null;
        assert msg != null;

        if (isLocalNodeDisconnected()) {
            throw new IgniteSpiException("Failed to send a message to remote node because local
node has " +
                "been disconnected [rmtNodeId=" + node.id() + ']');
        }

        ClusterNode locNode = getLocalNode();

        if (locNode == null)
            throw new IgniteSpiException("Local node has not been started or fully initialized " +
                "[isStopping=" + getSpiContext().isStopping() + ']');

        if (node.id().equals(locNode.id()))
            # 如果是本地节点，则通知CommunicationListener.onMessage
            notifyListener(node.id(), msg, NOOP);
        else {
            GridCommunicationClient client = null;

            int connIdx = connPlc.connectionIndex();

            try {
                boolean retry;

                do {
                    # 返回一个已经存在的或者新创建一个GridCommunicationClient
                    client = reserveClient(node, connIdx);

                    UUID nodeId = null;

                    if (!client.async())
                        nodeId = node.id();

                    # 藉由该GridCommunicationClient将消息发送出去
                    retry = client.sendMessage(nodeId, msg, ackC);

                    client.release();

                    if (retry) {
                        removeNodeClient(node.id(), client);

                        ClusterNode node0 = getSpiContext().node(node.id());

                        if (node0 == null)
                            throw new IgniteCheckedException("Failed to send message to remote
node " +
                                "(node has left the grid): " + node.id());
                    }

                    client = null;
                }
                while (retry);
            }
            catch (Throwable t) {
```

```
                ...
            }
            finally {
                if (client != null && removeNodeClient(node.id(), client))
                    client.forceClose();
            }
        }
    }
}
```

TcpCommunicationSpi.notifyListener调用的另一个地方是在TcpCommunicationSpi.srvLsnr中，这是一个GridNioServerListener结构，它会被注册给GridNioServer，并在GridNioServer接收到来自于client的事件时被通知：

```java
public class TcpCommunicationSpi extends IgniteSpiAdapter implements CommunicationSpi<Message> {
    private final GridNioServerListener<Message> srvLsnr =
        new GridNioServerListenerAdapter<Message>() {
        ...

        @Override public void onMessage(final GridNioSession ses, Message msg) {
            ConnectionKey connKey = ses.meta(CONN_IDX_META);

            if (connKey == null) {
                ...
            }
            else {
                ...

                notifyListener(connKey.nodeId(), msg, c);
            }
        }

        ...
    }

    ...

    GridNioServer.Builder<Message> builder = GridNioServer.<Message>builder()
        .address(locHost)
        .port(port)
        # 设置GridNioServerListener
        .listener(srvLsnr)
        .logger(log)
        .selectorCount(selectorsCnt)
        .igniteInstanceName(igniteInstanceName)
        .serverName("tcp-comm")
        .tcpNoDelay(tcpNoDelay)
        .directBuffer(directBuf)
        .byteOrder(ByteOrder.nativeOrder())
        .socketSendBufferSize(sockSndBuf)
        .socketReceiveBufferSize(sockRcvBuf)
        .sendQueueLimit(msgQueueLimit)
        .directMode(true)
        .metricsListener(metricsLsnr)
        .writeTimeout(sockWriteTimeout)
        .selectorSpins(selectorSpins)
        .filters(filters)
        .writerFactory(writerFactory)
        .skipRecoveryPredicate(skipRecoveryPred)
        .messageQueueSizeListener(queueSizeMonitor)
        .readWriteSelectorsAssign(usePairedConnections);
}
```

前面分析了TcpCommunicationSpi.notifyListener的调用关系，总结如下：

```
第一个调用：
GridIoManager.sendToGridTopic|sendToCustomTopic等
    - GridIoManager.send
        - TcpCommunicationSpi.sendMessage => 当且仅当接收到的消息的目标节点不是本节点时
            - TcpCommunicationSpi.sendMessage0
                - TcpCommunicationSpi.notifyListener => 当且仅当接收到的消息的目标节点就是本节点时
                    - TcpCommunicationSpi.lsnr.onMessage

第二个调用：
在TcpCommunicationSpi中初始化TcpCommunicationSpi.srvLsnr，这是一个GridNioServerListener结构，其中定义了onMessage()等方法；
TcpCommunicationSpi.srvLsnr
    - new GridNioServerListenerAdapter
        - onMessage
            - TcpCommunicationSpi.notifyListener

在TcpCommunicationSpi中创建GridNioServer时会将TcpCommunicationSpi.srvLsnr赋值给GridNioServer.lsnr，然后GridNioServer.lsnr又会进一步赋值给GridNioFilterChain.lsnr：
GridNioServer.AbstractNioClientWorker.body
    - GridNioServer.AbstractNioClientWorker.bodyInternal
        - GridNioServer.AbstractNioClientWorker.processSelectedKeys|GridNioServer.AbstractNioClientWorker.processSelectedKeysOptimized
            - GridNioServer.ByteBufferNioClientWorker.processRead|GridNioServer.DirectNioClientWorker.processRead
                - GridNioFilterChain.onMessageReceived
                    - GridNioFilterChain.TailFilter.onMessageReceived
                        - GridNioFilterChain.lsnr.onMessage  => 实际上就是TcpCommunicationSpi.srvLsnr.onMessage
                            - TcpCommunicationSpi.notifyListener
                                - TcpCommunicationSpi.lsnr.onMessage
```

TcpCommunicationSpi.lsnr.onMessage会进一步调用GridIoManager.onMessage0来完成它的逻辑（见GridIoManager.start方法）：

```java
public class GridIoManager extends GridManagerAdapter<CommunicationSpi<Serializable>> {
    private void onMessage0(UUID nodeId, GridIoMessage msg, IgniteRunnable msgC) {
        assert nodeId != null;
        assert msg != null;

        Lock busyLock0 = busyLock.readLock();

        busyLock0.lock();

        try {
            if (stopping) {
                # 如果当前节点处于stopping状态，则直接返回
                return;
            }

            if (msg.topic() == null) {
                int topicOrd = msg.topicOrdinal();

                msg.topic(topicOrd >= 0 ? GridTopic.fromOrdinal(topicOrd) :
                    U.unmarshal(marsh, msg.topicBytes(), U.resolveClassLoader(ctx.config())));
            }

            if (!started) {
                # 如果GridIoManager还没有完全起来，则将消息添加到waitMap中
                lock.readLock().lock();

                try {
                    if (!started) { // Sets to true in write lock, so double checking.
                        // Received message before valid context is set to manager.
                        if (log.isDebugEnabled())
                            log.debug("Adding message to waiting list [senderId=" + nodeId +
                                ", msg=" + msg + ']');

                        Deque<DelayedMessage> list = F.<UUID, Deque<DelayedMessage>>addIfAbsent(
                            waitMap,
                            nodeId,
                            ConcurrentLinkedDeque::new
                        );

                        assert list != null;

                        list.add(new DelayedMessage(nodeId, msg, msgC));

                        return;
                    }
                }
                finally {
                    lock.readLock().unlock();
                }
            }

            # 否则，根据消息对应的policy来分别处理
            // If message is P2P, then process in P2P service.
            // This is done to avoid extra waiting and potential deadlocks
            // as thread pool may not have any available threads to give.
            byte plc = msg.policy();

            switch (plc) {
                case P2P_POOL: {
                    # 如果是p2p消息，则发送给P2P线程池来进行处理
```

```
                processP2PMessage(nodeId, msg, msgC);

                break;
            }

            case PUBLIC_POOL:
            case SYSTEM_POOL:
            case MANAGEMENT_POOL:
            case AFFINITY_POOL:
            case UTILITY_CACHE_POOL:
            case IDX_POOL:
            case IGFS_POOL:
            case DATA_STREAMER_POOL:
            case QUERY_POOL:
            case SCHEMA_POOL:
            case SERVICE_POOL:
            {
                # 根据是否设置了ordered标志分别处理，分别交给响应的线程池进行处理
                if (msg.isOrdered())
                    processOrderedMessage(nodeId, msg, plc, msgC);
                else
                    processRegularMessage(nodeId, msg, plc, msgC);

                break;
            }

            default:
                assert plc >= 0 : "Negative policy [plc=" + plc + ", msg=" + msg + ']';

                if (isReservedGridIoPolicy(plc))
                    throw new IgniteCheckedException("Failed to process message with policy of
reserved range. " +
                        "[policy=" + plc + ']');

                if (msg.isOrdered())
                    processOrderedMessage(nodeId, msg, plc, msgC);
                else
                    processRegularMessage(nodeId, msg, plc, msgC);
        }
    }
    catch (IgniteCheckedException e) {
        U.error(log, "Failed to process message (will ignore): " + msg, e);
    }
    finally {
        busyLock0.unlock();
    }
  }
}
```

前面分析了当GridIoManager接收到其他GridManager的发送消息的请求时，如果消息的目标节点就是本地节点的情况下，会最终通知给TcpCommunicationSpi.lsnr.onMessage来处理消息（此时这个待发送的消息其实就是接收到的消息）；当GridNioServer中的AbstractNioClientWorkers接收到readable事件时，会最终通知给TcpCommunicationSpi.lsnr.onMessage来处理该消息。这些都是与接收到消息相关的，那么GridIoManager中发送消息又是怎么样的呢？前面提到，GridIoManager中提供了sendToGridTopic，sendOrderedMessage，sendToCustomTopic和sendOrderedMessageToGridTopic等，前面2个方法用于将消息发送给ignite内部的topic，而后面2个方法则用于将消息发送给用户自定义的topic。这些方法最终都是调用GridIoManager.send()方法来完成消息投递，如果本地节点不是消息的目标节点，则调用TcpCommunicationSpi.sendMessage进行发送。

```
GridIoManager.sendToGridTopic|sendToCustomTopic等
    - GridIoManager.send
        - TcpCommunicationSpi.sendMessage => 当且仅当接收到的消息的目标节点不是本节点时
            - TcpCommunicationSpi.sendMessage0 => 后续过程只适用于接收到的消息的目标节点不是本节点时
                - TcpCommunicationSpi.reserveClient => 返回GridCommunicationClient
                    - TcpCommunicationSpi.createNioClient => 如果clinnt需要创建的情况下
                        - TcpCommunicationSpi.createTcpClient
                            - new GridTcpNioCommunicationClient => 创建的时候会绑定到一个GridNioSession，实际绑定的是GridSelectorNioSessionImpl类型，这个GridSelectorNioSessionImpl会关联到GridNioServer.filterChain
                - GridCommunicationClient.sendMessage
                    - GridNioSessionImpl.send|GridNioSessionImpl.sendNoFuture
                        - GridNioFilterChain.onSessionWrite => 这里GridNioFilterChain实际上就是GridNioServer.filterChain
                            - TailFilter.onSessionWrite
                                - GridNioFilterAdapter.proceedSessionWrite
                                    - nextFilter.onSessionWrite(ses, msg, fut, ackC) => 从TailFilter开始，逐一调用每一个GridNioFilter的onSessionWrite方法，直到HeadFilter为止，HeadFilter.onSessionWrite方法见后续分析
                - GridCommunicationClient.release

HeadFilter.onSessionWrite
    - GridNioServer.send
        - GridNioServer.send0(GridSelectorNioSessionImpl ses, ...)
            - GridSelectorNioSessionImpl.offerSystemFuture|GridSelectorNioSessionImpl.offerFuture => 将SessionWriteRequest添加到GridSelectorNioSessionImpl的队列中，如果是offerSystemFuture，则添加到队列头部，否则添加到队列尾部，在GridNioServer.ByteBufferNioClientWorker.processWrite或者GridNioServer.DirectNioClientWorker.processWrite中会通过GridSelectorNioSessionImpl.pollFuture来获取待处理的写请求
            - AbstractNioClientWorker.offer((SessionChangeRequest)req)
                - GridNioServer.AbstractNioClientWorker.changeReqs.offer => 这样会添加到GridNioServer.AbstractNioClientWorker的change requests队列中，该队列中的请求将在GridNioServer.AbstractNioClientWorker.body中被处理，见后续分析

AbstractNioClientWorker.body
    - AbstractNioClientWorker.bodyInternal
        - 首先处理changeReqs队列中的消息，直到所有消息处理完毕
            while ((req0 = changeReqs.poll()) != null) {
                switch (req0.operation()) {
                    case REQUIRE_WRITE: {
                        SessionWriteRequest req = (SessionWriteRequest)req0;

                        registerWrite((GridSelectorNioSessionImpl)req.session());
                            - SelectionKey key = ses.key()
                            - key.interestOps(key.interestOps() | SelectionKey.OP_WRITE) => 设置当前SelectionKey关注WRITE事件

                        break;
                    }
                }
            }
        - 尝试在当前AbstractNioClientWorker对应的selector上进行select，如果没有select到，则至多尝试selectorSpins次，如果select到了，则处理之
            for (long i = 0; i < selectorSpins && res == 0; i++) {
                # select并且立即返回
                res = selector.selectNow();

                if (res > 0) {
                    # select到了一些待处理的SelectionKey，则处理之
```

```
                    updateHeartbeat();

                    if (selectedKeys == null)
                        # 对select到的消息进行处理
                        processSelectedKeys(selector.selectedKeys());
                    else
                        processSelectedKeysOptimized(selectedKeys.flip());
                }

                if (!changeReqs.isEmpty())
                    continue mainLoop;

                ...

                if (isCancelled())
                    return;
            }
        - 如果前面的selectNow没有select到，则尝试使用带超时时间的select
            try {
                if (!changeReqs.isEmpty())
                    continue;

                updateHeartbeat();

                # select的超时时间是2s，如果select到了，则处理之
                if (selector.select(2000) > 0) {
                    // Walk through the ready keys collection and process network events.
                    if (selectedKeys == null)
                        processSelectedKeys(selector.selectedKeys());
                    else
                        processSelectedKeysOptimized(selectedKeys.flip());
                }

                // select() call above doesn't throw on interruption; checking it here to
propagate timely.
                if (!closed && !isCancelled && Thread.interrupted())
                    throw new InterruptedException();
            }
            finally {
                select = false;
            }

GridNioServer.AbstractNioClientWorker.processSelectedKeys
    - 逐一遍历select到的各SelectionKeys，对每一个SelectionKey分别检查它上面的Connectable事件、Readable
事件和Writable事件，如果有相应的时间，则分别调用processConnect，processRead和processWrite进行处理
        - processConnect|processRead|processWrite
            - 在AbstractNioClientWorker中只提供了processConnect的实现，而processRead和processWrite均
是抽象方法，由相应的子类去实现，如ByteBufferNioClientWorker.processRead和
ByteBufferNioClientWorker.processWrite

GridNioServer.AbstractNioClientWorker.processConnect(SelectionKey key)
    - SocketChannel ch = (SocketChannel)key.channel()
    - NioOperationFuture<GridNioSession> sesFut =
(NioOperationFuture<GridNioSession>)key.attachment()
    - ch.finishConnect() => 结束当前的SocketChannel
    - register(sesFut)
        - 分配readBuf和writeBuf
        - 分配一个GridSelectorNioSessionImpl，并设置它的meta信息
        - 注册当前SocketChannel到selector上，并设置返回的SlectionKey关注READ事件
        - 将当前的GridNioSession添加到ridNioServer.sessions中，同时添加到
```

```
AbstractNioClientWorker.workerSessions中
        - 通知GridNioServer.filterChain关于GridNioSession的open事件

GridNioServer.ByteBufferNioClientWorker.processRead(SelectionKey key)
    - ReadableByteChannel sockCh = (ReadableByteChannel)key.channel()
    - GridSelectorNioSessionImpl ses = (GridSelectorNioSessionImpl)key.attachment()
    - sockCh.read(readBuf) => 从SocketChannel中读取数据到readBuf中
    - filterChain.onMessageReceived(ses, readBuf) => 通知GridNioFilterChain对读取到的数据进行处理
        - GridNioFilterChain.TailFilter.onMessageReceived
            - GridNioFilterChain.lsnr.onMessage  => 实际上就是TcpCommunicationSpi.srvLsnr.onMessage
                - TcpCommunicationSpi.notifyListener
                    - TcpCommunicationSpi.lsnr.onMessage

GridNioServer.ByteBufferNioClientWorker.processWrite(SelectionKey key)
    - ReadableByteChannel sockCh = (ReadableByteChannel)key.channel()
    - GridSelectorNioSessionImpl ses = (GridSelectorNioSessionImpl)key.attachment()
    - SessionWriteRequest req = ses.pollFuture() => GridSelectorNioSessionImpl队列中存放着等待处理的
SessionWriteRequest
    - buf = (ByteBuffer)req.message()
    - sockCh.write(buf)
```