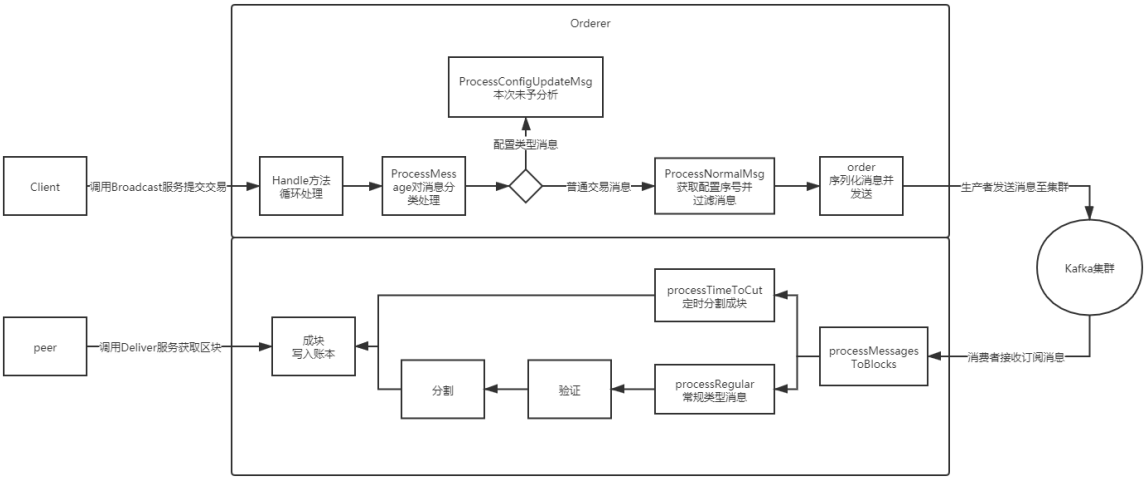
Orderer在交易处理中的功能分析

依据fabric的交易流程，本次仅分析1）客户端将背书好的交易提案发送给Orderer，Orderer接收并对消息进行预处理，主要调用的是Broadcast服务；2）Orderer使用kafka集群对普通提交消息进行接收打包分割，最后提交至账本。

总过程如下图：



本文共涉及文件如下：

```
/protos/common/common.pb.go
/protos/orderer/ab.proto
/orderer/common/broadcast/broadcast.go
/orderer/common/msgprocessor/msgprocessor.go
/orderer/common/msgprocessor/standardchannel.go
/orderer/common/mutichannel/blockwriter.go
/orderer/consensus/kafka/chain.go
```

一、 Broadcast服务

1 客户端发送消息至Orderer

客户端通过gRPC调用Orderer服务器提供的Broadcast服务，将Envelpoe格式的消息封装发送给Orderer。
其中Envelope的结构如下：

```
// /protos/common/common.pb.go
// Envelope wraps a Payload with a signature so that the message may be
authenticated
type Envelope struct {
    // 将交易、背书、读写集等封装并序列化形成的有效信息
    Payload []byte `protobuf:"bytes,1,opt,name=payload,proto3"
json:"payload,omitempty"`
    // 创建者的签名
    Signature            []byte   `protobuf:"bytes,2,opt,name=signature,proto3"
json:"signature,omitempty"`
    XXX_NoUnkeyedLiteral struct{} `json:"-"`
    XXX_unrecognized     []byte   `json:"-"`
    XXX_sizecache        int32    `json:"-"`
}
```

gRPC是由自谷歌开发的一项多语言开源的RPC技术，在fabric用于实现客户端与服务器端的远程调用。在Orderer节点启动时，Orderer会向gRPC服务器注册Broadcast服务供调用，其中Envelope格式的消息被当做参数传入Broadcast方法。
其中oederer的gRPC服务定义在/protos/orderer/ab.proto，Broadcast接收消息返回回应，Deliver接收一个包装了SeekInfo信息（即索要的block的起止范围信息）的HeaderType_CONFIG_UPDATE类型的Envelope消息，然后返回一系列block。

```
// /protos/orderer/ab.proto
service AtomicBroadcast {
    // broadcast receives a reply of Acknowledgement for each common.Envelope in
order, indicating success or type of failure
    rpc Broadcast(stream common.Envelope) returns (stream BroadcastResponse) {}

    // deliver first requires an Envelope of type DELIVER_SEEK_INFO with Payload
data as a mashaled SeekInfo message, then a stream of block replies is received.
    rpc Deliver(stream common.Envelope) returns (stream DeliverResponse) {}
}
```

在/orderer/common/server/server.go，orderer使用server实现了AtomicBroadcas接口。在orderer启动时，创建server实例并注册服务。其中两个服务Broadcast、Deliver直接对应交由两个成员broadcast.Handler（orderer/common/broadcast/broadcast.go）和deliver.Handler（orderer/common/deliver/deliver.go）的Handle处理。

```
// /orderer/common/server/server.go
type server struct {
    bh    *broadcast.Handler
    dh    *deliver.Handler
    debug *localconfig.Debug
    *multichannel.Registrar
}
```

2 Orderer对从Broadcast方法接收的消息进行预处理

orderer/server/server.go中的Broadcast方法接收到消息，交由成员bh的handle方法处理。

2.1 handle方法循环处理

方法内有一个循环用于接收、处理消息、发送回应。其中ProcessMessage()方法用于处理消息

```
// /orderer/common/broadcast/broadcast.go
```

```
// Handle reads requests from a Broadcast stream, processes them, and returns
the responses to the stream
func (bh *Handler) Handle(srv ab.AtomicBroadcast_BroadcastServer) error {
    addr := util.ExtractRemoteAddress(srv.Context())
    ......
    for {
        // 接收消息
        msg, err := srv.Recv()
        // 处理消息 具体内容在2.2节
        resp := bh.ProcessMessage(msg, addr)
        // 发送回应
        err = srv.Send(resp)
        if resp.Status != cb.Status_SUCCESS {
            return err
        }
        ......
    }

}
```

2.2 ProcessMessage()对消息分类处理

1）从消息中提取isConfig等信息，依据isConfig标志对不同交易消息分类处理，本次仅介绍普通交易消息的处理。
2）再调用processor.ProcessNormalMsg()方法获得相应通道的最新配置序号，并按照规则过滤消息。
3）最后调用processor.Order(msg, configSeq)，对消息排序出块。

```
// /orderer/common/broadcast/broadcast.go
// ProcessMessage validates and enqueues a single message
func (bh *Handler) ProcessMessage(msg *cb.Envelope, addr string) (resp
*ab.BroadcastResponse) {
    ......
    // 1)
    //channel header，从消息中提取，用于辨别和防止重放
    //isConfig，是否为配置消息，用来区分普通交易消息
    //processor，  消息的处理器，根据通道ID分配，
    //            一般情况下为StandardChannel，
    //            如果未找到则为未创建通道的情况，返回系统通道的消息处理器SystemChannel。
    chdr, isConfig, processor, err :=
bh.SupportRegistrar.BroadcastChannelSupport(msg)
    ......
    if !isConfig {  //普通交易消息
        ......
        // 2)
        //调用消息处理器获得configSeq:通道最新配置序号，并按照规则过滤消息
        //具体内容在2.3
        configSeq, err := processor.ProcessNormalMsg(msg)
        ......
        //processor.WaitReady()检查当前通道共识组件链对象是否准备好接收新消息
        if err = processor.WaitReady(); err != nil {
            ......
            return &ab.BroadcastResponse{Status: cb.Status_SERVICE_UNAVAILABLE,
Info: err.Error()}
        }
        // 3)
        //构造交易信息并排序出块，有solo、kafka、raft三种实现
        //具体内容在2.4
```

```
        err = processor.Order(msg, configSeq)
        if err != nil {
            ......
            return &ab.BroadcastResponse{Status: cb.Status_SERVICE_UNAVAILABLE,
Info: err.Error()}
        }
    } else { // isConfig 对于配置类型消息，本次不进行分析
        ......
        config, configSeq, err := processor.ProcessConfigUpdateMsg(msg)
        if err != nil {
            ......
            return &ab.BroadcastResponse{Status: ClassifyError(err), Info:
err.Error()}
        }
        ......
        if err = processor.WaitReady(); err != nil {
            ......
            return &ab.BroadcastResponse{Status: cb.Status_SERVICE_UNAVAILABLE,
Info: err.Error()}
        }
        err = processor.Configure(config, configSeq)
        if err != nil {
            ......
            return &ab.BroadcastResponse{Status: cb.Status_SERVICE_UNAVAILABLE,
Info: err.Error()}
        }
    }

    ......
    return &ab.BroadcastResponse{Status: cb.Status_SUCCESS}
}
```

2.3 ProcessNormalMsg方法

ProcessNormalMsg方法的接口定义在/orderer/common/msgprocessor/msgprocessor.go内的Processor接口，
1) 其中有ClassifyMsg方法，用于根据channelHeader对消息进行分类；
2) ProcessNormalMsg方法针对普通交易类型消息，将会根据配置验证和过滤消息，成功则返回配置序号和nil，无效信息则返回错误。ProcessConfigMsg方法针对 `ORDERER_TX` or `CONFIG` 类型消息，先解析消息，再调用ProcessConfigUpdateMsg方法；
3) ProcessConfigUpdateMsg方法会尝试将配置类型消息应用到通道配置上，成功则返回最新配置和配置序号，失败则返回错误。

```
// /orderer/common/msgprocessor/msgprocessor.go
// Processor provides the methods necessary to classify and process any message
which
// arrives through the Broadcast interface.
type Processor interface {
    // ClassifyMsg inspects the message header to determine which type of
processing is necessary
    ClassifyMsg(chdr *cb.ChannelHeader) Classification

    // ProcessNormalMsg will check the validity of a message based on the
current configuration.  It returns the current
    // configuration sequence number and nil on success, or an error if the
message is not valid
```

```
    ProcessNormalMsg(env *cb.Envelope) (configSeq uint64, err error)

    // ProcessConfigUpdateMsg will attempt to apply the config update to the
current configuration, and if successful
    // return the resulting config message and the configSeq the config was
computed from.  If the config update message
    // is invalid, an error is returned.
    ProcessConfigUpdateMsg(env *cb.Envelope) (config *cb.Envelope, configSeq
uint64, err error)

    // ProcessConfigMsg takes message of type `ORDERER_TX` or `CONFIG`, unpack
the ConfigUpdate envelope embedded
    // in it, and call `ProcessConfigUpdateMsg` to produce new Config message of
the same type as original message.
    // This method is used to re-validate and reproduce config message, if it's
deemed not to be valid anymore.
    ProcessConfigMsg(env *cb.Envelope) (*cb.Envelope, uint64, error)
}
```

对于普通交易消息调用 ProcessNormalMsg(msg)->StandardChannel.ProcessNormalMsg()方法，使用已分配的消息处理器获取最新配置序号并应用规则过滤消息。

```
// /orderer/common/msgprocessor/standardchannel.go
// ProcessNormalMsg will check the validity of a message based on the current
configuration.  It returns the current
// configuration sequence number and nil on success, or an error if the message
is not valid
func (s *StandardChannel) ProcessNormalMsg(env *cb.Envelope) (configSeq uint64,
err error) {
    // 检查Orderer配置
    ......
    // s.support.Sequence()->cs.ConfigtxValidator.Sequence()-
>ValidatorImpl.Sequence()
    // 获取通道最新配置序号configSeq，默认初始值为0，新建应用通道后该配置序号自增为1.
    // 该序号用于标志通道配置信息版本，通过比较此序号来确定是否配置发生变更，从而是否需要重新过
滤与排序
    configSeq = s.support.Sequence()
    //应用规则过滤消息，规则有：
    //EmptyRejectRule       验证不能为空
    //expiratioinRejectRule 拒绝过期签名者身份证书
    //MaxBytesRule          消息最大字节数
    //sigFilter             消息签名验证过滤器
    err = s.filters.Apply(env)
    return
}
```

2.4 Order()方法排序出块

Order方法的接口定义在/orderer/common/broadcast/broadcast.go内，
1) Order方法和Configure方法接收消息和配置序号、返回错误信息，分别对交易消息和配置消息进行处理排序成块等工作，过程中会使用到共识组件；
2) WaitReady方法用来检查共识组件是否准备好接收消息。
其中此接口的实现有solo(orderer/consensus/solo/consensus.go)、kafka(/orderer/consensus/kafka/chain.go)、etcdraft(/orderer/consensus/etcdraft/chain.go)三种。

```
// /orderer/common/broadcast/broadcast.go
```

```
// Consenter provides methods to send messages through consensus
type Consenter interface {
    // Order accepts a message or returns an error indicating the cause of
failure
    // It ultimately passes through to the consensus.Chain interface
    Order(env *cb.Envelope, configSeq uint64) error

    // Configure accepts a reconfiguration or returns an error indicating the
cause of failure
    // It ultimately passes through to the consensus.Chain interface
    Configure(config *cb.Envelope, configSeq uint64) error

    // WaitReady blocks waiting for consenter to be ready for accepting new
messages.
    // This is useful when consenter needs to temporarily block ingress messages
so
    // that in-flight messages can be consumed. It could return error if
consenter is
    // in erroneous states. If this blocking behavior is not desired, consenter
could
    // simply return nil.
    WaitReady() error
}
```

以kafka为例：
先将消息、通道最新配置序号、OriginalOffset封装为kafka常规消息格式，其中OriginalOffset传入参数值为0，标志第一次提交处理，用于与通道最后处理消息序号比较，若是重新验证重新排序的消息则此参数不为0，具体见第二部分2.4节处理kafka常规消息。
最后调用chain.enqueue方法将消息发送到kafka集群。

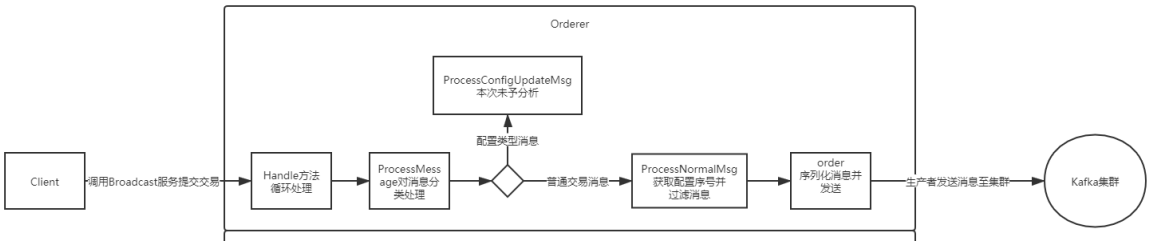
```
// /orderer/consensus/kafka/chain.go
func (chain *chainImpl) order(env *cb.Envelope, configSeq uint64, originalOffset
int64) error {
    // 将消息序列化为字节数组
    marshaledEnv, err := utils.Marshal(env)
    //newNormalMessage()方法用来将消息构造成kafka常规消息
    //Type: &ab.KafkaMessage_Regular{
    //      Regular: &ab.KafkaMessageRegular{
    //          Payload:        payload,        marshaledEnv字节数组
    //          ConfigSeq:      configSeq,
    //          Class:          ab.KafkaMessageRegular_NORMAL,
    //                          此为kafka消息类型标志，用于第二部分中对不同消息分类处理
    //          OriginalOffset: originalOffset, 为0，标志第一次提交
    //      },
    //}
    //chain.enqueue()将封装后的消息发送，见2.5 chain.enqueue()
    if !chain.enqueue(newNormalMessage(marshaledEnv, configSeq, originalOffset))
{
        return errors.Errorf("cannot enqueue")
    }
    return nil
}
```

2.5 chain.enqueue()

将消息发送到kafka集群的指定分区上请求排序，先检查当前通道的状态，如果有startChan消息，则表示此时为通道的创建或恢复过程中，因此不会发送消息。正常情况下会封装、调用SendMessage方法发送消息到kafka集群指定分区，此方法仅在全部消息发送成功或失败时返回nil，否则报错。

```
// /orderer/consensus/kafka/chain.go
// enqueue accepts a message and returns true on acceptance, or false otherwise.
func (chain *chainImpl) enqueue(kafkaMsg *ab.KafkaMessage) bool {
    ......
    select {
    // startChan在通道创建或恢复时才会关闭
    case <-chain.startChan:      // 开启的情况
        select {
        case <-chain.haltChan:   // 已停止的情况
            ......
            return false
        default:                 // 正常情况下
            payload, err := utils.Marshal(kafkaMsg)
            ......
            // 创建kafka生产者消息
            message := newProducerMessage(chain.channel, payload)
            // 发送消息到kafka集群请求排序
            // SendMessage()方法的接口定义在
            // /vendor/github.com/Shopify/sarama/sync_producer.go
            // 只有全部发送成功或失败返回nil，否则为error
            if _, _, err = chain.producer.SendMessage(message); err != nil {
                ......
                return false
            }
            ......
            return true
        }
    default:                     // 此时是通道正在创建或恢复，因此不会发送消息
        ......
        return false
    }
}
```

至此，Orderer节点对接收到的消息的预处理过程结束，下面为Orderer的共识排序服务。这一部分流程如下图：



二、共识排序服务

共识排序服务完成对消息的排序、出块，以kafka共识组件为例

1 kafka共识组件

Orderer 节点采用 Sarama 开源的 Kafka 第三方库构建 Kafka 共识组件，可以同时接受处理多个客户端发送的交易消息请求，能够有效提高 Orderer 节点处理交易消息的并发能力。同时，可利用 Kafka 集群在单一分区内按序收集相同主题消息（消息序号唯一）的功能，来保证交易消息具有确定性的顺序（以消息序号排序），从而实现对交易排序达成全局共识的目的。

Kafka 包括生产者、消费者和分区消费者。Kafka 生产者按照主题（Topic）生产消息并进行发布，Kafka 服务器集群自动对消息主题进行分类。同一个主题的消息都会被收集到一个或多个分区文件中，按照 FIFO（先进先出）的顺序追加到文件尾部，并且每个消息在分区中都会有一个 OFFSET 位置偏移量作为该消息的唯一标识 ID。目前，Hyperledger Fabric 基于 Kafka 集群为每个通道创建绑定了一个主题（即链 ID，chainID），并且只设置一个分区（分区号为 0）。Kafka 消费者管理多个分区消费者并订阅指定分区的主题消息，包括主题（即 chainID）、分区号（目前只有 1 个分区号为 0 的分区）、起始偏移量（开始订阅的消息位置 offset）等。当生产者、消费者以及分区消费者对象不再使用时，要执行 Close() 方法以释放占用的资源。

Hyperledger Fabric 采用 Kafka 集群对单个或多个 Orderer 排序节点（或称为 Orderer 服务集群）提交的交易消息进行排序。此时，Orderer 排序节点同时充当 Kafka 集群的消息生产者（分区）和消费者，发布消息与订阅消息到 Kafka 集群上的同一个主题分区，即先将 Peer 节点提交的交易消息转发给 Kafka 服务端，同时，从指定主题的 Kafka 分区上按顺序获取排序后的交易消息并自动过滤重启的交易消息。这期间可能会存在网络时延造成获取消息时间的差异。如果不考虑丢包造成消息丢失的情况，则所有 Orderer 节点获取消息的顺序与数量应该是确定的和一致的。同时，采用相同的 Kafka 共识组件链对象与出块规则等，以保证所有 Orderer 节点都可以创建与更新相同配置的通道，并切割生成相同的批量交易集合出块，再“同步”构造出相同的区块数据，从而基于 Kafka 集群达成全局共识，以保证区块数据的全局一致性。相比于 Solo 共识组件，Kafka 共识组件提供的 Orderer 服务具有良好的可用性与容错能力，一方面，Kafka 集群利用多副本机制确保发生 Kafka 服务器节点故障（CFT 错误）时的数据可用性，另一方面，Orderer 服务集群中所有 Orderer 服务节点都保存了相同的账本数据副本，如图 2-6 所示，6 个 Orderer 节点共同构成了 Orderer 服务集群。如果 Orderer 服务节点发生崩溃故障（CFT 类型错误）如 $Orderer_2$ 节点，则 Client 或 Peer 节点可以重新连接到其他可用节点（如 $Orderer_3$）继续获取服务，从而提高 Orderer 服务集群的可用性与容错能力。

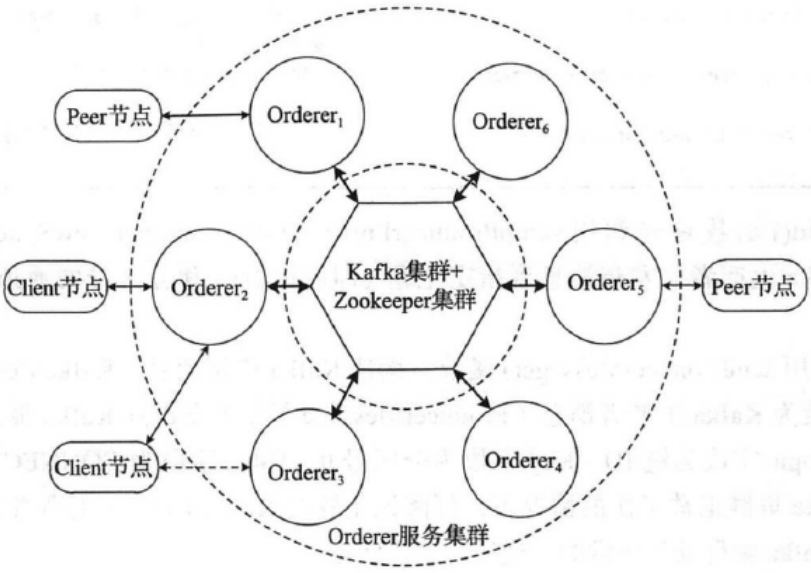


图 2-6　支持多个 Orderer 排序节点（Orderer 服务集群）的 Kafka 共识组件架构示意图

以上内容为《Hyperledger Fabric技术内幕 架构设计与实现原理》内对kafka共识组件的介绍，通过kafka集群，多个Orderer间可达到消息顺序唯一的同步效果。第一部分最后生产者调用sendMessage方法发送的消息，最终会被kafka集群的消费者在processMessagesToBlocks方法内接收。

2 消费者接收并处理消息processMessagesToBlocks()

此方法在for循环内等待消息，利用select语句阻塞等待通道上的新消息，
1）若接收到kafka连接消息，则目前不进行任何操作；
2）若接收到kafka定时分割生成区块消息，则立即对消息分割出块并提交账本；
3）若接收到kafka常规消息则进入processRegular方法对此消息进行进一步处理，其中传入的参数in.Offset为上一节介绍的唯一的序号。
4）若接收到超时定时器的消息，此消息在下文 2.4节 processRegular方法 内设置发送，则表明一个区块内的消息之间的最大时间间隔达到配置限制，将最终调用sendMessage方法向kafka集群发送定时分割生成区块消息。

```
// /orderer/consensus/kafka/chain.go
// processMessagesToBlocks drains the Kafka consumer for the given channel, and
// takes care of converting the stream of ordered messages into blocks for the
// channel's ledger.
func (chain *chainImpl) processMessagesToBlocks() ([]uint64, error) {
    ......
    for {
        select {  //没有default分支，即必须阻塞等待下面某个分支满足条件之后再继续执行
        ......
        //接收到正常的kafka分区消费者消息（大多数正常消息的情况下）
        case in, ok := <-chain.channelConsumer.Messages():
            ......
            select {
            case <-chain.errorChan: //如果该通道已经关闭则重建
                chain.errorChan = make(chan struct{}) // ...make a new one.
            default:
            }
            if err := proto.Unmarshal(in.Value, msg); err != nil {//解析消息
                ......
            }
            //根据kafka消息类型
            switch msg.Type.(type) {
            case *ab.KafkaMessage_Connect:      //kafka连接消息
                // 目前不进行任何操作
                _ = chain.processConnect(chain.ChainID())
            case *ab.KafkaMessage_TimeToCut:    //kafka定时分割生成区块消息
                //定时分割生成区块，即已经达到一个区块内消息之间的最大时间间隔，需要分割成块
                //具体内容在下文2.1
                if err := chain.processTimeToCut(msg.GetTimeToCut(), in.Offset);
err != nil {
                    ......
                }
            case *ab.KafkaMessage_Regular:      //kafka常规消息
                // processRegular()方法处理常规消息，具体内容在2.4节，
                //其中传入的in.Offset参数即是在 1.kafka共识组件 这一节内介绍的
                //kafka集群在单一分区内相同主题的消息序号唯一的序号，
                //其在整个生产消费过程中是序列化排序依次分配给每个消息的，
                //因此每个消息具有唯一的offset。
                if err := chain.processRegular(msg.GetRegular(), in.Offset); err
!= nil {
                    ......
```

```
                }
        //接收到超时定时器消息，到达配置的区块的最大等待时间，在2.4节内容中设置
        case <-chain.timer:
            //发送TimeToCut类型消息请求打包出块，
            //chain.lastCutBlockNumber+1即为下一次要处理的区块号，
            //最终调用sendMessage()方法将封装后的消息发送到kafka集群进行处理，
            //最后在本方法内的KafkaMessage_TimeToCut这一分支进行处理
            if err := sendTimeToCut(chain.producer, chain.channel,
chain.lastCutBlockNumber+1, &chain.timer); err != nil {
                ......
        }
    }
}
```

2.1 定时分割生成区块ProcessTimeToCut()

一个区块内的消息之间的最大时间间隔达到配置限制时，立即分割出块。其中消息内区块号用于确定这个超时产生的区块即为刚刚发送的超时消息指定的应处理的区块，否则为过期的定时分割生产区块消息，直接丢弃。

```
// /orderer/consensus/kafka/chain.go
func (chain *chainImpl) processTimeToCut(ttcMessage *ab.KafkaMessageTimeToCut,
receivedOffset int64) error {
    //消息内的区块号
    ttcNumber := ttcMessage.GetBlockNumber()
    //若消息内区块号是当前Orderer节点当前通道账本的下一个打包出块的区块号
    if ttcNumber == chain.lastCutBlockNumber+1 {
        //分割当前该通道上待处理缓存交易消息列表为批量交易集合batch([]*cb.Envelope)
        batch := chain.BlockCutter().Cut()
        //创建下一个区块，具体内容在2.2 创建下一个区块
        //得到的block已经拥有Header和Data
        block := chain.CreateNextBlock(batch)
        //block的最后一部分：metadata结构
        metadata := &ab.KafkaMetadata{
            //区块内保存的最后一条消息的序号
            LastOffsetPersisted:         receivedOffset,
            //最近处理过的重新验证且重新排序的消息的序号
            LastOriginalOffsetProcessed: chain.lastOriginalOffsetProcessed,
        }
        //将metadata写入block并提交账本，具体内容在2.3
        chain.WriteBlock(block, metadata)
        //最新区块号增1
        chain.lastCutBlockNumber++
        return nil
    } else if ttcNumber > chain.lastCutBlockNumber+1 {
        //若消息区块号不是下一个打包出块区块号，则丢弃
        ......
    }
    return nil
}
```

2.2 创建下一个区块

将消息封装到block.Data，并生成block.Header{Number, previousBlockHash, DataHash}

```
// /orderer/common/mutichannel/blockwriter.go
```

```
// CreateNextBlock creates a new block with the next block number, and the given
contents.
func (bw *BlockWriter) CreateNextBlock(messages []*cb.Envelope) *cb.Block {
    // 前一块区块的Hash
    previousBlockHash := bw.lastBlock.Header.Hash()
    //将message转为二维byte数组进行封装
    data := &cb.BlockData{
        Data: make([][]byte, len(messages)),
    }
    for i, msg := range messages {
        data.Data[i], err = proto.Marshal(msg)
    }
    //区块中的三个组成部分中的两部分
    //Header{Number,previousBlockHash,DataHash}
    //Data
    //在此创建并赋值，还缺少metadata部分
    block := cb.NewBlock(bw.lastBlock.Header.Number+1, previousBlockHash)
    block.Header.DataHash = data.Hash()
    block.Data = data

    return block
}
```

2.3 写入区块 writeBlock()方法

此方法用来提交区块给账本。先获得提交区块锁，然后开启一个线程异步处理提交区块，然后此方法返回，此后提交区块方法结束才释放提交区块锁。

```
// /orderer.common/multichannel/blockwriter.go
// WriteBlock should be invoked for blocks which contain normal transactions.
// It sets the target block as the pending next block,
// and returns before it is committed.
// Before returning, it acquires the committing lock, and spawns a go routine
which will
// annotate the block with metadata and signatures, and write the block to the
ledger
// then release the lock.  This allows the calling thread to
// begin assembling the next block
// before the commit phase is complete.
func (bw *BlockWriter) WriteBlock(block *cb.Block, encodedMetadataValue []byte)
{
    //加锁
    bw.committingBlock.Lock()
    bw.lastBlock = block
    //异步进行处理
    go func() {
        //提交完毕则将锁撤销
        defer bw.committingBlock.Unlock()
        bw.commitBlock(encodedMetadataValue)
    }()
}
```

2.4 处理kafka常规消息

在本方法中前半部分是提交普通交易消息方法，后半部分是对普通交易消息的检查验证部分，处理流程为先进行后半部分的处理工作，最后调用前半部分来提交普通消息方法。

其中后半部分对消息验证部分：
1）解析消息，对于普通交易类型，先判断是否为第一次提交，否则比较通道内保存的最近重新验证且重新排序的消息的序号，若小于意味着已经处理过该消息，直接丢弃。
2）检查配置序号是否过期，否则将消息重新验证重新排序。
3）调用前半部分实现的方法提交消息。

前半部分是提交普通交易类型消息方法：
1）对本条消息及缓存的消息进行分割成批，返回0//1/2批消息及是否有缓存消息标志。具体方法见2.5节。
2）设置超时定时器，以便控制一个区块内的消息之间的最大时间间隔。
3）构造提交区块。

其中区块内的offset字段为当前区块的最后一条消息的序号，因此当
1）分割后只返回一批消息且有缓存消息，则本条消息导致缓存消息形成1批消息且本条消息未进入此批消息，因此offset为上一条消息的序号；
2）分割后返回两批消息，这种情况在2.5节中介绍，此时第一批消息为本条消息之前的缓存消息，第二批消息为本条消息，因此在用第一批消息生成区块时，Offset为上一条消息的序号，用第二批消息生成区块时则再把Offset+1，即为本条消息的序号；
3）分割后返回一批且无缓存，此时为2.5节中介绍的本条消息加入缓存消息并导致缓存消息总数达到最大消息数限制，最终将包括本条消息的缓存消息作为1批消息返回，因此此条消息包括在其内，Offset为此条消息的序号。

lastOriginalOffsetProcessed字段，意为最近处理过的重新验证且重新排序的消息的序号，用于对过期的重新验证且重新排序的消息进行过滤。

```
// /orderer/consensus/kafka/chain.go
func (chain *chainImpl) processRegular(regularMessage *ab.KafkaMessageRegular,
receivedOffset int64) error {
    // 当提交一条常规消息时，我们需要利用本方法内声明的newOffset参数更新
    // 最近处理过的重新验证且重新排序的消息的序号，
    // 即block内的metadata部分的lastOriginalOffsetProcessed参数。
    // 因此newOffset需要满足以下规则：
    // 1 如果重新提交功能关闭，则始终为0，即所有消息都为第一次提交；
    // 2 如果消息第一次提交即没有重新验证重新排序，则newOffset为当前的
    //   最近处理过的重新验证且重新排序的消息的序号；
    // 3 如果消息重新验证重新排序，则需将newOffset设为其自身的消息序号，
    //   并最终更新到 最近处理过的重新验证且重新排序的消息的序号
lastOriginalOffsetProcessed。
    //
    // 提交普通交易消息方法
    commitNormalMsg := func(message *cb.Envelope, newOffset int64) {
        //区块的分割，返回0/1/2批消息和缓存标志，具体内容见2.5 Ordered方法
        batches, pending := chain.BlockCutter().Ordered(message)
        //超时定时器的使用，用于控制一个区块内消息间的时间戳的最大间隔
        switch {
        case chain.timer != nil && !pending:
            // 定时器已经启动但没有缓存消息，则定时器取消
            chain.timer = nil
        case chain.timer == nil && pending:
            // 定时器未启动且有缓存消息，则启动超时定时器
            //BatchTimeout是配置的参数
            chain.timer = time.After(chain.SharedConfig().BatchTimeout())
        default:
            // Do nothing when:
            // 1. Timer is already running and there are messages pending
            // 2. Timer is not set and there are no messages pending
```

```
        }

        if len(batches) == 0 {
            // 批大小为0的情况，即消息较少，不用提交，等待收集
            // 更新最近处理过的重新验证且重新排序的消息的序号
            chain.lastOriginalOffsetProcessed = newOffset
            return
        }

        offset := receivedOffset    //设置当前消息序号为in.Offset即消息的序号，此序号唯
一
        if pending || len(batches) == 2 {
            // If the newest envelope is not encapsulated into the first batch,
            // the `LastOffsetPersisted` should be `receivedOffset` - 1.
            //返回1批且有缓存消息或有两批消息且缓存为0，则这一条消息将不会进入这一批中
            //所以offset减1，即lastOffsetPersisted区块保存的最后一条消息的序号，
            //为本条消息的上一条的序号
            offset--
        } else {
            //返回一批消息且无缓存消息的情况，更新 最近处理过的重新验证且重新排序的消息的序号
            chain.lastOriginalOffsetProcessed = newOffset
        }

        //构造并提交第一个区块
        block := chain.CreateNextBlock(batches[0])
        metadata := &ab.KafkaMetadata{
            // 区块最后一条消息的序号
            LastOffsetPersisted:         offset,
            // 本通道最后处理的消息序号
            LastOriginalOffsetProcessed: chain.lastOriginalOffsetProcessed,
            LastResubmittedConfigOffset: chain.lastResubmittedConfigOffset,
        }
        chain.WriteBlock(block, metadata)
        chain.lastCutBlockNumber++


        // 检查构造并提交第二个区块，事实上这个区块最多包含一个交易，
        // 具体内容查看2.5 Ordered方法解释的batch=2 pending=false情况
        if len(batches) == 2 {
            chain.lastOriginalOffsetProcessed = newOffset
            //这是第二个区块且仅有1个交易，因此offset需要加1
            offset++

            block := chain.CreateNextBlock(batches[1])
            metadata := &ab.KafkaMetadata{
                LastOffsetPersisted:         offset,
                LastOriginalOffsetProcessed: newOffset,
                LastResubmittedConfigOffset: chain.lastResubmittedConfigOffset,
            }
            chain.WriteBlock(block, metadata)
            chain.lastCutBlockNumber++
        }
    }

//#################################################################################
    //对消息进行预处理部分
    seq := chain.Sequence()
    env := &cb.Envelope{}
```

```
    // 解析kafka常规消息到Envelope结构对象
    if err := proto.Unmarshal(regularMessage.Payload, env); err != nil {
        ......
    }
    ......
    switch regularMessage.Class {
    case ab.KafkaMessageRegular_UNKNOWN:
    case ab.KafkaMessageRegular_NORMAL: //普通交易类型
        // 如果第一次提交则OriginalOffset为0
        // 如果OriginalOffset不是0，则说明该消息是重新验证并重新提交排序的
        if regularMessage.OriginalOffset != 0 {
            //如果OriginalOffset消息序号不大于最近处理过的重新验证且重新排序的消息的序号，
            //则说明已经处理过该消息，此时则丢弃该消息
            if regularMessage.OriginalOffset <=
chain.lastOriginalOffsetProcessed {
                ......
                return nil
            }
        }
        // 检查通道的配置序号是否更新，否则需要重新验证重新排序
        if regularMessage.ConfigSeq < seq {
            //消息的配置序号低，说明当前通道配置升级过，而此消息是按照旧配置验证过滤的，
            //需要重新验证过滤消息
            //ProcessNormalMsg()方法在本章第一部分2.3节ProcessMornalMsg
            configSeq, err := chain.ProcessNormalMsg(env)
            //重新提交交易信息来排序，order()方法在本章第一部分
            //此时receivedOffset参数为本条消息的序号且不为0，标志此条消息不是第一次提交
            //第一次提交时的调用在第一部分中，且receivedOffset设为0
            if err := chain.order(env, configSeq, receivedOffset); err != nil {
                ......
            }
            return nil
        }

        // 第一次提交时该消息的offset为0，需更新为本通道的 最近处理过的重新验证且重新排序的消
息的序号
        // 当且仅当消息重新验证和重新排序时，才将offst赋值为此条消息本身的消息序号
        // 最终传入commitNormalMsg方法中，
        // 效果为只有 重新提交重新排序情况 才更新 最近处理过的重新验证且重新排序的消息的序号
        offset := regularMessage.OriginalOffset
        if offset == 0 {
            offset = chain.lastOriginalOffsetProcessed
        }
        //提交处理普通交易消息
        commitNormalMsg(env, offset)
    default:
        ......
    }
    return nil
}
```

2.5 Ordered方法 分割形成批消息

Ordered方法及使用到的Cut方法接口在/orderer/common/blockcutter/blockcutter.go内。

Ordered方法用来对消息排序形成成批消息，此成批消息将被用来打包成块。

Cut方法用来将排序好的消息分割成批。

```
// /orderer/common/blockcutter/blockcutter.go
// Receiver defines a sink for the ordered broadcast messages
type Receiver interface {
    // Ordered should be invoked sequentially as messages are ordered
    // Each batch in `messageBatches` will be wrapped into a block.
    // `pending` indicates if there are still messages pending in the receiver.
    Ordered(msg *cb.Envelope) (messageBatches [][]*cb.Envelope, pending bool)

    // Cut returns the current batch and starts a new one
    Cut() []*cb.Envelope
}
```

具体实现也在同一个文件内，对本条消息及缓存的消息进行分割，返回0/1/2批消息及缓存标志。

```
// /orderer/common/blockcutter/blockcutter.go
// Ordered should be invoked sequentially as messages are ordered
//本方法返回情况：
// 返回0批，缓存为0：不可能的情况，因为本方法被调用时已经传入一条消息，此消息将被放入缓存消息
// 返回0批，有缓存：  没有消息在被分割，且有缓存。
// 返回1批，缓存为0：缓存消息达到最大消息数量的情况。
// 返回1批，有缓存：  此条消息导致缓存消息达到最优最大数量。
// 返回2批，缓存为0：此条消息超过最优最大数量且之前有缓存消息，
//                  则缓存消息为1批，这条消息单独为1批，共返回2批。
// 返回2批，有缓存：  不可能的情况。
// 最多同时存在2批数据
func (r *receiver) Ordered(msg *cb.Envelope) (messageBatches [][]*cb.Envelope,
pending bool) {
    // 获取orderer的配置以调控区块的大小，相关配置项在configtx.yaml内修改
    ordererConfig, ok := r.sharedConfigFetcher.OrdererConfig()
    //batchsize包括有MaxMessageCount 指定block最多存储的消息数量
    //AbsoluteMaxBytes指定block最大字节数
    //PreferredMaxBytes指定block最优最大字节数，在分割区块过程中会尽力使每一批消息保持在这
个值上
    batchSize := ordererConfig.BatchSize()

    messageSizeBytes := messageSizeBytes(msg)
    //若此条消息字节数超过最优最大字节数
    if messageSizeBytes > batchSize.PreferredMaxBytes {
        // 如果有缓存消息则缓存消息形成一批消息
        if len(r.pendingBatch) > 0 {
            messageBatch := r.Cut()
            messageBatches = append(messageBatches, messageBatch)
        }

        // 总的效果为无论是否有缓存消息，立即形成批消息返回
        // 因此若有缓存消息则批大小为2，缓存标志未修改默认为false
        // 若无缓存消息则批大小为1，缓存标志未修改默认也为false
        messageBatches = append(messageBatches, []*cb.Envelope{msg})
        return
    }
    //若缓存消息加当前消息超出最优最大字节数，则缓存消息立即形成一批消息
    messageWillOverflowBatchSizeBytes :=
r.pendingBatchSizeBytes+messageSizeBytes > batchSize.PreferredMaxBytes
    if messageWillOverflowBatchSizeBytes {
        messageBatch := r.Cut()
        messageBatches = append(messageBatches, messageBatch)
```

```
    }
    //将此次消息加入缓存消息
    r.pendingBatch = append(r.pendingBatch, msg)
    r.pendingBatchSizeBytes += messageSizeBytes
    pending = true
    //若缓存消息达到最大消息数则立即将缓存消息形成一批消息
    if uint32(len(r.pendingBatch)) >= batchSize.MaxMessageCount {
        messageBatch := r.Cut()
        messageBatches = append(messageBatches, messageBatch)
        pending = false
    }

    return
}
```

至此第二部分Orderer的共识排序部分结束，Orderer节点生成区块并提交到账本，随后Peer节点调用Deliver服务来获取区块。总流程如下：