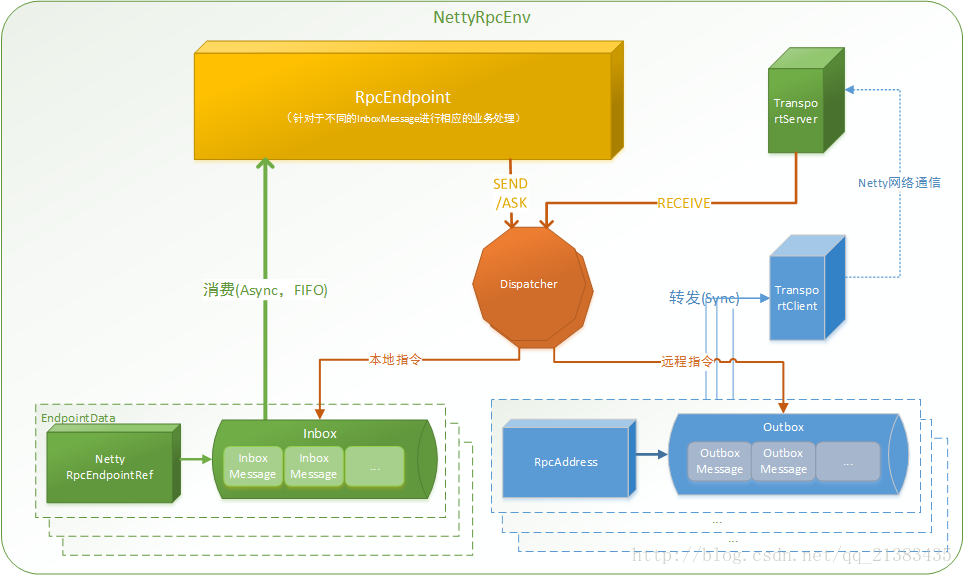
Spark Network模块分析

Spark Network-Common模块底层使用Netty作为通信框架，实现Rpc消息、数据块和数据流的传输，下图是Spark RPC流程图：



* RpcEndpoint，RPC端点，每个节点都称之为Rpc端点，且都实现RpcEndpoint接口，根据不同端点的需求，设计不同的消息和业务处理。
* RpcEnv，Rpc上下文环境，每个Rpc端点依赖的上下文
* Dispatcher，消息分发器，针对于RPC端点需要发送消息或者从远程RPC接收到消息，分发至对应的指令Inbox和Oubbox。
* Inbox，指令消息收件箱，本地端点对应Inbox，Dispatcher每次向Inbox存入消息时，都将对应的EndpointData加入到对应的Receiver Queue中，另外Dispacher创建时会启动一个单独线程轮询Receiver Queue，进行收件箱消息的消费
* Outbox，指令消息发件箱，一个远程端点对应一个发件箱，当消息放入Outbox后，紧接着将消息通过TransportClient发送过去，消息放入发件箱和发送过程在同一个线程中进行。主要做的原因是远程消息分为RpcOutboxMessage和OneWayOutboxMessage两种消息
* TransportClient，Netty通信客户端，根据Outbox消息的receive消息，请求对应远程的TransportServer

Spark RPC的核心模块是Dispatcher，读取每个Endpoint的Inbox中的Message进行处理，包括RpcMessage（远程调用）和OneWayMessage(单向消息，远程调用)等，然后根据客户端指定的Endpoint进行发送。Dispatcher的消息分发的调用是通过NettyRpcHandler来实现，这就是Reactor中的线程的工作，RpcHandler是org.apache.spark.spark-network-common提供的handler，当远程数据包解析成功后，会调用handler进行处理。Spark-Network-common是封装Netty来实现Spark RPC的主要模块，其中包括Byte流的处理、协议、编解码、Netty通信Handler及客户、Server端的封装等，下面进行逐一介绍。

# 1.RpcEnv NettyServer

Spark 1.6版本后RPC底层的实现使用netty，其中核心类是RpcEnv，用于各个组件之间的通信，每个节点之间（Driver或者Worker）组件的Endpoint和对应EndpointRef之间的信息通信和方法调用都是通过RpcEnv作协调，通信底层是通过Netty NIO框架实现（2.0之后统一由Netty替换成Akka，实现了通信传输统一化）。SparkEnv中的RpcEnv的使用如下：

*val systemName = if (isDriver) driverSystemName else executorSystemName*

*val rpcEnv = RpcEnv.create(systemName, bindAddress, advertiseAddress, port.getOrElse(-1), conf,*

*securityManager, numUsableCores, !isDriver)*

RpcEnv分为Driver及Executor System，目前支持的NettyRpcEnv是对network-common模块中的组件进行封装来提供Rpc服务，启动过程如下：



在spark-network-common模块中：

1. TransportContext维护了Transport的上下文环境，用于创建TransportServer和TransportClientFactory
2. TransportServer通过构造函数启动netty，提供底层通信服务
3. TransportClientFactory，用于创建TransportClient，与TransportServer通信

还有MessageEncoder,MessageDecoder,RpcHandler等与Netty的业务逻辑相关。

# 2.Buffer

对于Network通信，不管传输的是序列化后的对象还是文件，在网络上表现的都是字节流：在传统IO中，字节流表示为Stream；在NIO中，字节流表示为ByteBuffer；在Netty中字节流表示为ByteBuf或FileRegion;在Spark中，针对Byte也做了一层包装，支持对Byte和文件流进行处理，即ManagedBuffer。相关类图如下：

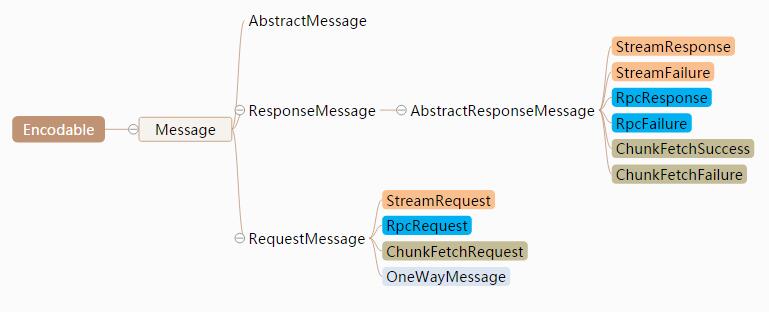


ManagedBuff包含三个函数：createInputStream,nioByteBuffer,convertToNetty来对Buffer进行封装，分别获取stream,ByteBuffer,ByteBuf或FileRegion。相对应的三种实现类：

* NioManagedBuff，对Java Nio中的ByteBuffer的具体实现。
* NettyManagedBuff，数据在内存中，可以将字符数组表示为NettyManagedBuffer，是对Netty Nio ByteBuf的实现。
* FileSegmentManagedBuff，在Shuffle BlockManager模块需要在内存中维护本地executor生成的shuffle-map输出的文件引用，从而可以提供给ShuffleFetch进行远程读取，调用nioByteBuffer和createInputStream函数从文件中读取为Bytes，并进行后面的网络传输。

# 3.Message

Message是Spark Network Common的应用层通信的基础，提供了应用层通信的数据表示，以及解码、编码能力，如下图所示：



Message的特征如下：

1. 请求响应可以划分为RequestMessage和ResponseMessage两种，
2. 对于Response，根据划分结果可以划分为Failure和Success两种类型。
3. 根据功能的不同，Request和Response主要划分为Stream,ChunkFetch,Rpc

具体类型见Message.Type:



**1）Stream Message**

Stream消息指ManagedBuffer中的Stream流，在Spark内部，SparkContext的adddFile操作会在Driver中针每一个add进来的file/jar分配一个streamId。Worker通过该StreamID向Driver发起一个StreamRequest请求，Driver将文件转换为FileSegmentManagerBuffer返回给Worker。Stream Message用于Shuffle中获取文件：对应类OneForOneBlockFetcher

*使用场景：为了Spark Job的执行，在每个节点添加一个文件，例如Job运行依赖的jar包。也可以是Job中共享的文件，可以是本地文件、HDFS文件或者HTTP、FTP等URI对应的文件。*

**2）ChunkFetch Message**

ChunkFetch类似Stream的概念，CheckFetch的对象是一个内存中的Iterator[ManagedBuffer]，即一组Buffer，每一个Buffer对应一个ChunkIndex，整个Iterator[ManagerBuffer]由一个StreamId标识。Client每次ChunkFetch请求都是由<StreamId,chunkIndex>组成的StreamChunkId，Server端根据StreamChunkId获取一个Buffer并返回给Client。

不管是Stream还是ChunkFetch，在Server的内存中都需要管理一组有StreamID与资源之间的映射，即StreamManager类，提供getChunk和openStream两个接口来响应ChunkFetch和Stream两种操作，并且针对Server的ChunkFetch提供一个registerStream接口来注册一组Buffer。例如将BlockManager中的一组BlockID对应的的Iterator[ManagedBuffer]注册到StreamManager，从而支持远程的Block Fetch操作

*使用场景：ExternalShuffleService（单独的Shuffle服务进程，对其他计算节点提供本节点的所有Shuffle Map输出），为远程Executor提供OpenBlocks的RPC接口，根据请求的appId，executorid,blockId对应的本地目录，从本地磁盘中加载一组FileSegementManagedBuffer到内存，并返回加载后的streamId返回给客户端，从而支持ChunkFetch操作。*

**3）RpcMessage**

在Spark中各组件的通信系统，都是使用RpcEnv作为核心，其负责RpcEndpoint整个生命周期的管理，其通过解析TCP层的数据包并反序列化，封装成RpcMessage，并且路由到指定的Endpoint，调用相应的业务逻辑。

例如Driver与Executor之间的心跳交互过程，如下图所示：

1） 在heartbeatReceiverRef.askSync中将HeartBeart消息，封装到RequestMessage中，代码如下所示：

*val response = heartbeatReceiverRef.askSync[HeartbeatResponse](*

*message, RpcTimeout(conf, "spark.executor.heartbeatInterval", "10s"))*

2）调用NettyRpcEndpointRef中的ask方法，将Heartbeat消息封装到RequestMessage中：

*override def ask[T: ClassTag](message: Any, timeout: RpcTimeout): Future[T] = {*

*nettyEnv.ask(new RequestMessage(nettyEnv.address, this, message), timeout)*

*}*

3）RequestMessage的构造过程中，将message序列化成ByteBuffer。在传输的过程中，同时rpcAddress及receiverAddress等都会序列化进去，该RequestMessage仅是RpcEnv层面的ByteBuff。并该消息发送到Outbox中，等待处理。

1. RpcEnv将该消息传输给Server端时，会调用TransportClient.sendRpc方法，如下：

*channel.writeAndFlush(new RpcRequest(requestId, new NioManagedBuffer(message))*

最终转换为RpcRequest

1. Server的handler在接收RpcRequest以后，获取request的类型RequestMessage后，进行处理（处理过程需要看MessageDecoder的handler处理）

*public void channelRead(ChannelHandlerContext ctx, Object request) throws Exception {*

*if (request instanceof RequestMessage) {*

*requestHandler.handle((RequestMessage) request);*

*}}*

1. 将RequestMessage封装再次成RpcMessage后传入Inbox

*def postRemoteMessage(message: RequestMessage, callback: RpcResponseCallback): Unit = {*

*val rpcCallContext =*

*new RemoteNettyRpcCallContext(nettyEnv, callback, message.senderAddress)*

*val rpcMessage = RpcMessage(message.senderAddress, message.content, rpcCallContext)*

*postMessage(message.receiver.name, rpcMessage, (e) => callback.onFailure(e))*

*}*

1. 调用Inbox中的MessageLoop进行处理

*message match {*

*case RpcMessage(\_sender, content, context) =>*

*try {*

*endpoint.receiveAndReply(context).applyOrElse[Any, Unit](content, { msg =>*

*throw new SparkException(s"Unsupported message $message from ${\_sender}")*

*})*

*}*

1. 调用HeartbeatReceiver的Endpoint处理如下

*override def receiveAndReply(context: RpcCallContext): PartialFunction[Any, Unit]{*

*case heartbeat @ Heartbeat(executorId, accumUpdates, blockManagerId)*

*}*

9） 整个过程封装步骤如下：

Client端：

*HeartBeat -> RequestMessage -> RpcMessage->*

Channel:

*ByteBuffer*

Server端：

*RpcMessage->RequestMessage->HeartBeat*

其中RpcMessage是RequestMessage的实现类。

# 4.Netty Handler

上述类中，NettyRpcHandler是对RpcHandler的继承，用于处理RPC消息的处理。RpcHandler位于network-common中，NettyRpcHandler主要是将接收的消息交给Dispatcher进行处理。Netty处理RPC类型请求依赖TransportChannelHandler，在TransportServer初始化时添加到pipeline中，如下所示：

*public TransportChannelHandler initializePipeline(*

*SocketChannel channel,*

*RpcHandler channelRpcHandler) {*

*try {*

*TransportChannelHandler channelHandler = createChannelHandler(channel, channelRpcHandler);*

*channel.pipeline()*

*.addLast("encoder", ENCODER)*

*.addLast(TransportFrameDecoder.HANDLER\_NAME, NettyUtils.createFrameDecoder())*

*.addLast("decoder", DECODER)*

*.addLast("idleStateHandler", new IdleStateHandler(0, 0, conf.connectionTimeoutMs() / 1000))*

*.addLast("handler", channelHandler);*

*return channelHandler;*

*} …….*

*}*

相关类如下所示：



在createChannelHandler中添加了TransportResponseHandler和TransportRequestHandler。

## 4.1 MessageCoder/Decoder

针对网络包到Message的编码和解码，MessageCoder继承MessageToMessageEnconder<Message>抽象类，编码如下：

*@Override*

*public void encode(ChannelHandlerContext ctx, Message in, List<Object> out) throws Exception {*

*Object body = null;*

*long bodyLength = 0;*

*boolean isBodyInFrame = false;*

*if (in.body() != null) {*

*try {*

*bodyLength = in.body().size(); //message的长度*

*body = in.body().convertToNetty();*

*isBodyInFrame = in.isBodyInFrame();*

*} catch (Exception e) {}*

*Message.Type msgType = in.type();*

*int headerLength = 8 + msgType.encodedLength() + in.encodedLength(); //Header比特长度*

*long frameLength = headerLength + (isBodyInFrame ? bodyLength : 0);*

*ByteBuf header = ctx.alloc().heapBuffer(headerLength); header.writeLong(frameLength); //Message Type的长度*

*msgType.encode(header); //写入Message Type*

*in.encode(header);*

*if (body != null) {*

*out.add(new MessageWithHeader(in.body(), header, body, bodyLength)); //body*

*} else {*

*out.add(header);*

*}*

*}*

MessageDecoder继承MessageToMessageDecoder<ByteBuf>，将ByteBuf转换成Message对象

*private Message decode(Message.Type msgType, ByteBuf in) {*

*switch (msgType) {*

*case ChunkFetchRequest:*

*return ChunkFetchRequest.decode(in);*

*case ChunkFetchSuccess:*

*return ChunkFetchSuccess.decode(in);*

*case ChunkFetchFailure:*

*return ChunkFetchFailure.decode(in);*

*case RpcRequest:*

*return RpcRequest.decode(in);*

*case RpcResponse:*

*return RpcResponse.decode(in);*

*case RpcFailure:*

*return RpcFailure.decode(in);*

*case OneWayMessage:*

*return OneWayMessage.decode(in);*

*case StreamRequest:*

*return StreamRequest.decode(in);*

*case StreamResponse:*

*return StreamResponse.decode(in);*

*case StreamFailure:*

*return StreamFailure.decode(in);*

*default:*

*throw new IllegalArgumentException("Unexpected message type: " + msgType);*

*}*

*}*

## 4.2 TransportRequestHandler和TransportResponseHandler

TrasnportRequestHandler和TransportResponseHandler是Spark Network Common的核心，封装了所有的请求/响应处理。TransportChannelHandler内部实现仅是封装了RequestHandler和ResponseHandler，当从Netty中读取一个Message以后，会根据路由给相应的responseHandler和requestHandler，如下所示：

*public void channelRead(ChannelHandlerContext ctx, Object request) throws Exception {*

*if (request instanceof RequestMessage) {*

*requestHandler.handle((RequestMessage) request);*

*} else if (request instanceof ResponseMessage) {*

*responseHandler.handle((ResponseMessage) request);*

*} else {*

*ctx.fireChannelRead(request);*

*}}*

Server提供的RPC、ChunkFetch和Stream的功能都是依赖TransportRequestHandler来实现的，从原理上看，RPC与ChunkFetch/Stream都有很大的不同，其中RPC对于TransportRequestHandler来说是功能依赖，而ChunkFetch/Stream来说只是数据依赖。

1）ChunckFetch和Stream

TransportRequestHandler中已经提供ChunkFetch/Stream的实现，只需要在构造的时候，向handler提供StreamManager，告诉RequestHandler从哪里可以读取到chunk或者Stream，如下所示：

*public TransportRequestHandler(Channel channel,TransportClient reverseClient,*

*RpcHandler rpcHandler,Long maxChunksBeingTransferred) {*

*this.channel = channel;this.reverseClient = reverseClient;*

*this.rpcHandler = rpcHandler;*

*this.streamManager = rpcHandler.getStreamManager();*

*this.maxChunksBeingTransferred = maxChunksBeingTransferred; }*

2) RPC

对于Rpc的处理需要向TransportRequestHandler注册一个rpcHandler，针对每个RPC接口进行功能实现，同时RCP与ChunkFetch/Stream都会有同一个StreamManager的依赖，一次StreamManager也是依赖rpcHandler（rpcHandler中提供了RPC功能实现和StreamManager的数据依赖）

*private final RpcHandler rpcHandler*

其中rpcHandler的实现类为NettyRpcHandler，receiver处理为：

*override def receive(*

*client: TransportClient,*

*message: ByteBuffer,*

*callback: RpcResponseCallback): Unit = {*

*val messageToDispatch = internalReceive(client, message)*

*dispatcher.postRemoteMessage(messageToDispatch, callback)*

*}*

其中internalReceive的实现如下:

*private def internalReceive(client: TransportClient, message: ByteBuffer): RequestMessage = {*

*val addr = client.getChannel().remoteAddress().asInstanceOf[InetSocketAddress]*

*assert(addr != null)*

*val clientAddr = RpcAddress(addr.getHostString, addr.getPort)*

*val requestMessage = RequestMessage(nettyEnv, client, message)*

*if (requestMessage.senderAddress == null) {*

*new RequestMessage(clientAddr, requestMessage.receiver, requestMessage.content)*

*} else {*

*// The remote RpcEnv listens to some port, we should also fire a RemoteProcessConnected for*

*// the listening address*

*val remoteEnvAddress = requestMessage.senderAddress*

*if (remoteAddresses.putIfAbsent(clientAddr, remoteEnvAddress) == null) {*

*dispatcher.postToAll(RemoteProcessConnected(remoteEnvAddress))*

*}*

*requestMessage*

*}*

将接收到的消息转换成RequestMessage，然后交给Dispatcher进行处理。

## 4.3 NettyRpcHandler

处理Rpc消息，如下所示：

*override def receive(*

*client: TransportClient,*

*message: ByteBuffer,*

*callback: RpcResponseCallback): Unit = {*

*val messageToDispatch = internalReceive(client, message)*

*dispatcher.postRemoteMessage(messageToDispatch, callback)*

*}*

提交给Inbox，Inbox的MessageLoop对消息的处理如下：

*def process(dispatcher: Dispatcher): Unit = {*

*safelyCall(endpoint) {*

*message match {*

*case RpcMessage(\_sender, content, context) =>*

*try {*

*endpoint.receiveAndReply(context).applyOrElse[Any, Unit](content, { msg =>*

*throw new SparkException(s"Unsupported message $message from ${\_sender}")*

*})*

*}....*

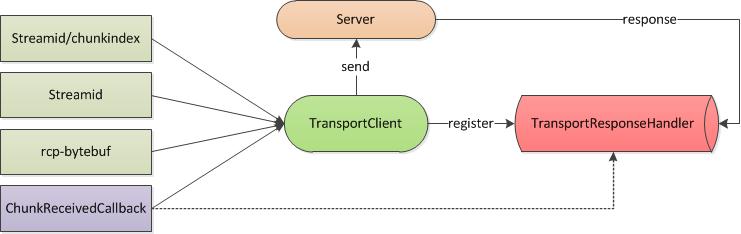
当RpcEndpoint启动后，将rpcEndpoint注册到Dispatcher中：

*if (endpoints.putIfAbsent(name, new EndpointData(name, endpoint, endpointRef))*

# 5.Client与Server

Server通过监听一个端口，注入rpcHandler和streamMananger从而对外提供Rpc、ChunkFetch和Stream的服务。而Client即为一个客户端，通过该类，可以将streamId和chunkeId对应的CheckFetch请求，StreamId对应的Stream请求，以及Rpc数据包对应的Rpc请求发送给服务端，并监听和处理来自服务端的响应。

客户端类和TransportResponseHandler负责处理来自服务端的响应，如下所示：



TrannsportClient将用户的RPC、ChunkFetch、Stream的请求进行打包并发送给Server端，同时将用户提供的回调函数注册到TransportResponseHnadler，判断为响应包以后，将包数据路由到TransportResponseHandler，如下所示:

*public void handle(ResponseMessage message) throws Exception {*

*if (message instanceof ChunkFetchSuccess) {*

*ChunkFetchSuccess resp = (ChunkFetchSuccess) message;*

*ChunkReceivedCallback listener = outstandingFetches.get(resp.streamChunkId);*

*if (listener == null) {*

*logger.warn("Ignoring response for block {} from {} since it is not outstanding",*

*resp.streamChunkId, getRemoteAddress(channel));*

*resp.body().release();*

*} else {*

*outstandingFetches.remove(resp.streamChunkId);*

*listener.onSuccess(resp.streamChunkId.chunkIndex, resp.body());*

*resp.body().release();*

*}*

*}*

*......*

*}*

使用Netty实现Rpc:

https://github.com/neoremind/navi-pbrpc/wiki/Tutorials

Spark Network-Comon

https://github.com/ColZer/DigAndBuried/blob/master/spark/spark-network-netty.md

Spark Network-common模块设计原理

http://blog.csdn.net/ggz631047367/article/details/78122605

Spark RPC:

http://geek.csdn.net/news/detail/234515

https://gist.github.com/baymaxium/4905db44099c63a79c0c1bded0c7954f

http://blog.csdn.net/qq\_21383435/article/details/78976051