Spark RpcEnv

RpcEnv是Spark集群RPC通信的基础服务环境，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）。

Spark RPC主要在两个模块中：

1. Spark Core，和Scala语言融合封装了Server和Client，依赖模块spark-network-common
2. Spark-network-common中，用Java语言编写，基于Netty4开发，提供全双工、多路复用I/O的Socket I/O

# 1.Spark RPC使用示例

kraps-rpc将Spark RPC抽离出来的单独项目，在maven项目中添加依赖如下：

*<dependency>*

*<groupId>net.neoremind</groupId>*

*<artifactId>kraps-rpc\_2.11</artifactId>*

*<version>1.0.0</version>*

*</dependency>*

要开发一个Hello服务，客户端传输String，服务端响应hi或者bye，并echo回去输入的string。

* 1. **创建Endpoint**，定义HelloEndpoint（RpcServer的业务逻辑），继承自RpcEndpoint可以并发调用该服务，如果继承ThreadSafeRpcEndpoint则不允许并发，源码如下：

*class HelloEndpoint(override val rpcEnv: RpcEnv) extends RpcEndpoint {*

*override def onStart(): Unit = {*

*println("start hello endpoint")*

*}*

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

*case SayHi(msg) => {*

*println(s"receive $msg")*

*context.reply(s"hi, $msg")*

*}*

*case SayBye(msg) => {*

*println(s"receive $msg")*

*context.reply(s"bye, $msg")*

*}*

*}*

*override def onStop(): Unit = {*

*println("stop hello endpoint")*

*}*

*}*

*case class SayHi(msg: String)*

*case class SayBye(msg: String)*

* 1. **运行Server**

提供HelloEndpoint服务的RPC Server要实现以下几个步骤：

1. 创建RpcEnvServerConfig，RpcConf中可以定义Server的参数，下面的示例中：

hello-server，Server的名称

host和port，主机和端口号，如果不能使用配置的端口，则retry，端口自增

1. 创建RpcEnv，通过Tpc Socket：localhost:52345，启动Server，使用NettyRpcEnvFactory这个工厂方法。
2. 创建HelloEndpoint，使用标识：HelloService，客户端使用HelloService来调用并路由到correct service
3. awaitTermination，阻塞线程，让server不退出jvm的情况下一直运行，源码如下：

*import net.neoremind.kraps.RpcConf*

*import net.neoremind.kraps.rpc.\_*

*import net.neoremind.kraps.rpc.netty.NettyRpcEnvFactory*

*object HelloworldServer {*

*def main(args: Array[String]): Unit = {*

*val config = RpcEnvServerConfig(new RpcConf(), "hello-server", "localhost", 52345)*

*val rpcEnv: RpcEnv = NettyRpcEnvFactory.create(config)*

*val helloEndpoint: RpcEndpoint = new HelloEndpoint(rpcEnv)*

*rpcEnv.setupEndpoint("HelloService", helloEndpoint)*

*rpcEnv.awaitTermination()*

*}*

*}*

* 1. **Client Call**

1. 异步调用

创建RpcEnv，使用setupEndpointRef来创建调stub，可以访问到标识为HelloSevice的Remote Serve，源码如下：

*import net.neoremind.kraps.RpcConf*

*import net.neoremind.kraps.rpc.{RpcAddress, RpcEndpointRef, RpcEnv, RpcEnvClientConfig}*

*import net.neoremind.kraps.rpc.netty.NettyRpcEnvFactory*

*import scala.concurrent.{Await, Future}*

*import scala.concurrent.duration.Duration*

*import scala.concurrent.ExecutionContext.Implicits.global*

*object HelloworldClient {*

*def main(args: Array[String]): Unit = {*

*import scala.concurrent.ExecutionContext.Implicits.global*

*val rpcConf = new RpcConf()*

*val config = RpcEnvClientConfig(rpcConf, "hello-client")*

*val rpcEnv: RpcEnv = NettyRpcEnvFactory.create(config)*

*val endPointRef: RpcEndpointRef = rpcEnv.setupEndpointRef(RpcAddress("localhost", 52345), "HelloService")*

*val future: Future[String] = endPointRef.ask[String](SayHi("neo"))*

*future.onComplete {*

*case scala.util.Success(value) => println(s"Got the result = $value")*

*case scala.util.Failure(e) => println(s"Got error: $e")*

*}*

*Await.result(future, Duration.apply("30s"))*

*}*

*}*

1. 同步调用

和异步调用步骤相同，不同的地方是使用askWithRetry而不是ask方法来调动server，如下所示：

*object HelloworldClient {*

*def main(args: Array[String]): Unit = {*

*import scala.concurrent.ExecutionContext.Implicits.global*

*val rpcConf = new RpcConf()*

*val rpcConf = new RpcConf()*

*val config = RpcEnvClientConfig(rpcConf, "hello-client")*

*val rpcEnv: RpcEnv = NettyRpcEnvFactory.create(config)*

*val endPointRef: RpcEndpointRef = rpcEnv.setupEndpointRef(RpcAddress("localhost", 52345), "HelloService")*

*val result = endPointRef.askWithRetry[String](SayBye("neo"))*

*println(result)*

*}*

*}*

在Spark中RpcConf的实现类为SparkConf，参数配置如下：

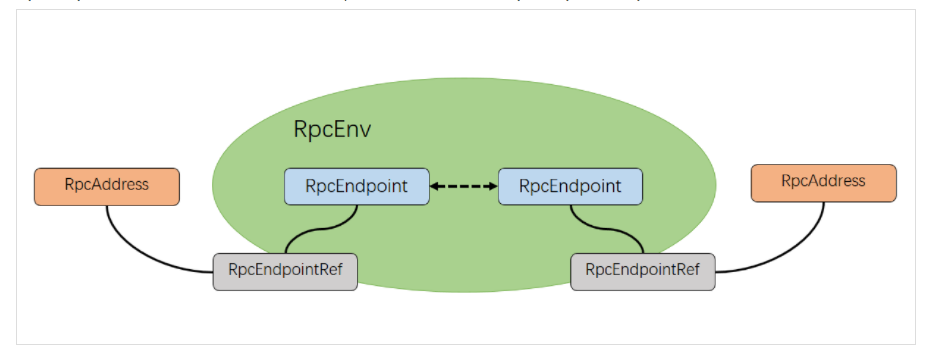
*-Dspark.rpc.netty.dispatcher.numThreads=16 -Dspark.rpc.io.threads=8*

其他参数如下：

|  |  |
| --- | --- |
| 参数 | 描述 |

# 2.Spark RPC相关类分析

当前Spark使用是基于Netty的Rpc框架，各个组件可以认为是独立的实体，之间通过消息来进行通信，关系图如下所示：



1. **RpcEndpoint**，需要通信的个体（如master,worker,driver等），根据接收到的消息进行对应的处理。RpcEndpoint经历的过程依次是：构建—>onStart—>receive—>onStop。其中onStart在接收任务消息前调用，receive和receiveAndReply分别用来接收另一个RpcEndpoint send和ask过来的消息。核心方法为：

a）receive，接收RpcEndpointRef.send和RpcCallContext.reply传过来的消息

*def receive: PartialFunction[Any, Unit] = {*

*case \_ => throw new SparkException(self + " does not implement 'receive'")*

*}*

当遇到不匹配类型的消息，则抛出SparkException。该方法是单向方式的，可以比作UDP。

b）receiveAndReply，接收RpcEndpointRef.ask传过来的类型消息，消息不匹配则抛出SparkException

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

*case \_ => context.sendFailure(new SparkException(self + " won't reply anything"))*

*}*

该方法是应答方式的，比作TCP。

在类中出现的RpcCallContext用来分离核心业务逻辑和底层传输的桥接方法，可以看出RPC的多用组合、聚合以及回调callback的设计模式来做OO抽象，这样可以剥离业务逻辑—> RPC封装（Spark-Core）—> 底层通信（spark-network-common）三者。RpcCallContext提供两个方法，用于回复正常的响应和错误异常，例如：

*reply(response: Any)，回复一个message，可以是一个case class*

*sendFailure(e: Throwable)，回复一个异常，可以是Exception子类，由于Spark Rpc默认采用JAVA序列化，所以异常在完整在客户端还原并作为cause throw出去*

在RpcEndpoint中提供以下一系列回调函数，如下：

1. onError
2. onConnected
3. onNetworkError
4. onStart
5. onStop
6. stop

需要注意的是，RpcEndpoint的子类是ThreadSafeRpcEndpoint，Spark Rpc框架中对这种Endpoint不做并发处理，也就是同一时间只允许一个线程做调用。还有一个默认的RpcEndpoint叫做RpcEndpointVerifier，每一个RpcEnv初始化的时候都会注册上这个Endpoint，因为客户端的调用每次都需要先询问服务端是否存在某一个Endpoint。

1. **RpcEndpointRef**，对远程RpcEndpoint的引用，给远程Endpoint发送消息时需要获取该RpcEndpoint的引用，然后通过该应用发送消息，可以是同步或者异步，其映射为一个地址，主要的实现类为NettyRpcEndpointRef，其通信端为client:TranspointClient。RpcEndpoingRef是客户端发起请求的入口。
2. **RpcAddress，**用来表示主机名和端口号，其伴生对象用于从URL等构造RpcAddress对象
3. **RpcEnv（**类似于ActorySystem对象），为RpcEndpoint提供处理消息的环境，所有的RpcEndpoint需要注册到该对象中用于接受消息，注册时指定name。RpcEnf负载RpcEndpoint整个生命周期的管理，包括注册endpoint、endpoint之间消息的路由以及停止endpoint。

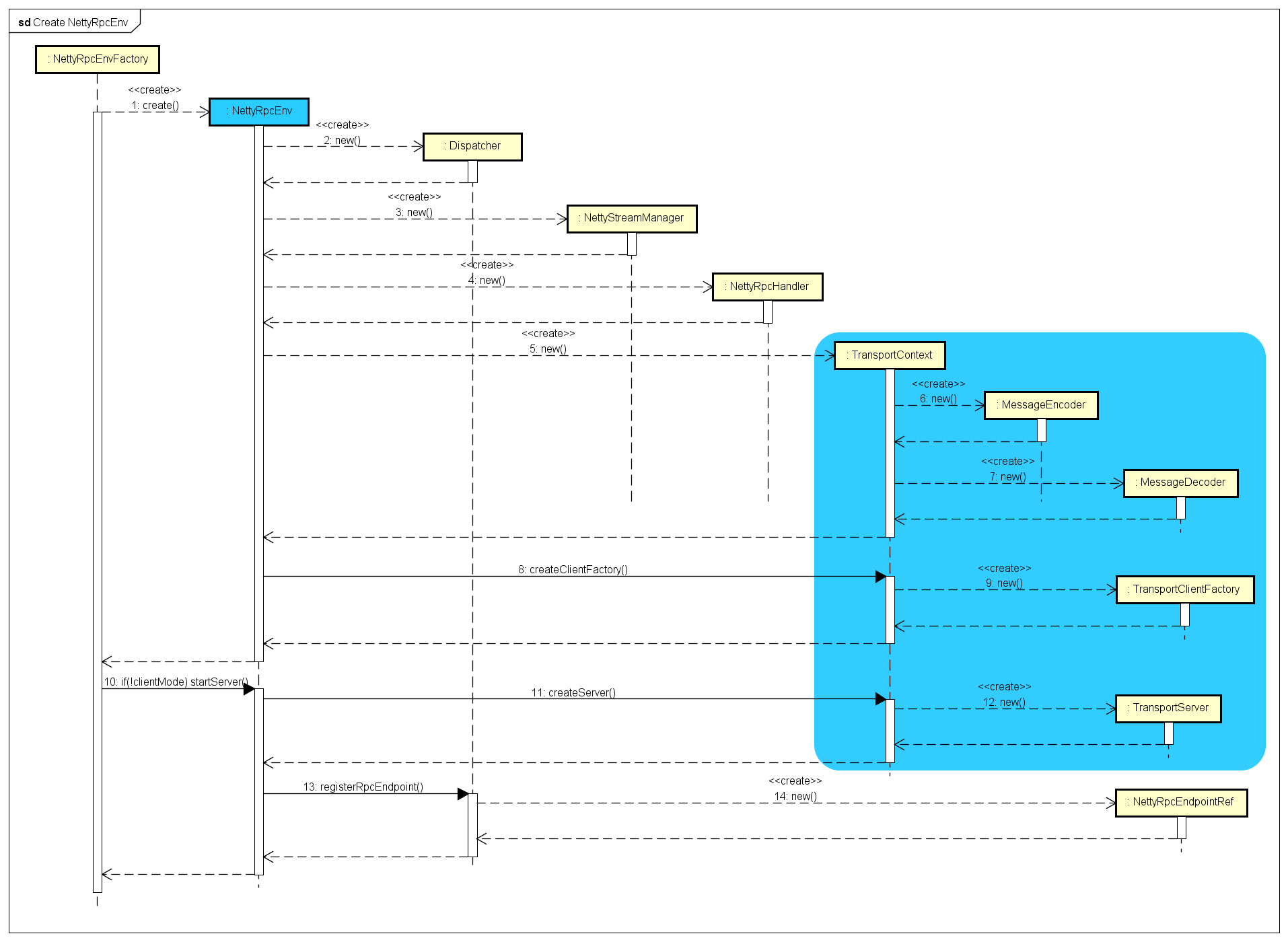
RpcEnv将会处理从RpcEndpointRef和远程节点发送过来的消息，然后发送给消息的Endpoint处理，对于接收到的异常使用RpcCallContext处理。其实现类为NettyRpcEnv，该类中包含Netty Server及对消息的处理逻辑，NettyRpcEnvFactory用于创建一个RpcEnv。

Spark RPC实现相关类之间的关系图如下所示：



## 2.1 NettyRpcEnv

创建NettyRpcEnv对象，会创建很多用来处理底层RPC通信的线程和数据结构，创建过程如下所示：



具体描述如下：

1. 创建一个NettyRpcEnv对象，通过NettyRpcEnvFactory创建
2. Dispatcher负责RPC消息的路由，能够将消息路由到对应的RpcEndpoint进行处理
3. NettyStreamManager负责提供文件服务，包括文件,jar文件及目录等
4. NettyRpcHandler负责处理网络IO事件，接收RPC调用请求，并通过Dispatcher派发消息
5. TransportContext负责管理网络传输上下文信息，创建MessageEncoder、MessageDecoder、TransportClientFactory和TransportServer
6. TransportServer配置并启动一个Rpc Server服务

NettyRpcEnv主要用于Endpoint的注册、启动TransportServer，获得RPCEndpointRef及创建客户端等，主要成员是dispatcher和TransportContext。

## 2.2 Dispatcher

Dispatcher的主要作用是保存注册的RpcEndpoint，分发相应的Message到RpcEndpoint中进行处理。在创建RpcEnv对象后，会通过setupEndpoint注册到Dispatcher中，同时返回RpcEndpointRef引用对象。下面介绍类中的成员对象及其作用：



1. EndpointData，定义如下：

*private class EndpointData(*

*val name: String,*

*val endpoint: RpcEndpoint,*

*val ref: NettyRpcEndpointRef) {*

*val inbox = new Inbox(ref, endpoint)*

*}*

Dispatcher的内部类，维护一个Endpoint的信息及inbox，Dispatcher会将消息发送到InBox中。

其中：*receivers:LinkedBlockingQueue[EndpointData]，*维护一个Blocking的队列，保存拥有消息的EndpointData。注册Endpoint、发送消息、停止RpcEnv、取消注册的Endpoint时，都会在receivers中添加相应的EndpointData。

1. *endpoints：ConcurrentHashMap[String,EndpointData]*，保存name与EndpointData的关系
2. *endpointRefs:ConcurrentHashMap[String,RpcEndpointRef]*，保存RpcEndpoint与RpcEndpointRef的关系

核心成员方法及其使用：

1. registerRpcEndpoint，在RpcEnv上进行注册

*def registerRpcEndpoint(name: String, endpoint: RpcEndpoint): NettyRpcEndpointRef = {*

*//根据NettyEnv的address和参数name,创建RpcEndpointAddress*

*val addr = RpcEndpointAddress(nettyEnv.address, name)*

*//创建对应的NettyRpcEndpointRef*

*val endpointRef = new NettyRpcEndpointRef(nettyEnv.conf, addr, nettyEnv)*

*synchronized {*

*//新建一个EndpointData，包含一个Inbox成员，将其添加到endpoints中*

*if (endpoints.putIfAbsent(name, new EndpointData(name, endpoint, endpointRef)) != null) {*

*throw new IllegalArgumentException(s"There is already an RpcEndpoint called $name")*

*}*

*val data = endpoints.get(name)*

*//将endpoint和对应的endpointRef添加到对应的endpointRefs中*

*endpointRefs.put(data.endpoint, data.ref)*

*//在receivers中添加新创建的endpointData*

*receivers.offer(data) // for the OnStart message*

*}*

*endpointRef //返回对应的EndpointRef}*

1. postToAll，向所有已经注册的RpcEndpoint发送消息，用于向所有的endpoints发送events，例如a new node connected

*def postToAll(message: InboxMessage): Unit = {*

*val iter = endpoints.keySet().iterator()*

*while (iter.hasNext) {*

*val name = iter.next*

*postMessage(name, message, (e) => { ……*

*}}*

*)}*

}

1. postMessage，将消息发送给特定的endpoint进行处理，参数1

*private def postMessage(*

*endpointName: String, //endpoint的名字*

*message: InboxMessage, //要传递的消息*

*callbackIfStopped: (Exception) => Unit): Unit = { //当endpoint停止时调用回调函数*

*val error = synchronized {*

*val data = endpoints.get(endpointName)//根据endpontName获得对应的endpointData*

*if ….else {*

*data.inbox.post(message) //将message添加到该endpointData的inbox的message中*

*receivers.offer(data) //将endpointData添加到receives中*

*None*

*}*

*}*

*// We don't need to call `onStop` in the `synchronized` block*

*error.foreach(callbackIfStopped)*

*}*

Dispatcher的主要职责：

1）内部使用集合endpoints和endpointRefs维护Endpoint,EndpointRef，对外通过registerRpcEndpoint、removeRpcEndpointRef等方法提供Endpoint注册删除和获取EndpointRef服务

2）利用EndpoingData和Inbox结构完成消息的存储

3）创建线程池threadPool，执行MessageLoop线程，处理消息

## 2.3 Inbox

Dispatcher在分发Message到相应endpointData中进行处理时，最重要的成员就是Inbox。定义如下：



Inbox的作用：

1. 内部维护了messages的LinkedList，用于存储消息，同时维护该消息对应的消费者Endpoint
2. 提供了post和process两个方法，分别用户添加消息到messages和消费消息，process方法在MessageLoop中被调用

可以处理的消息类型为InboxMessage，实现类为RpcMessage、OneWayMessage等。

## 2.4 Outbox

Outbox作用于Client端，当RpcEndpointRef请求RpcEndpoint时，若RpcEndpointRef和RpcEndpoint位于同一机器时，使用Inbox的逻辑，不在同一台机器，则RpcEndpointRef将信息发送到Outbox，类如下所示：

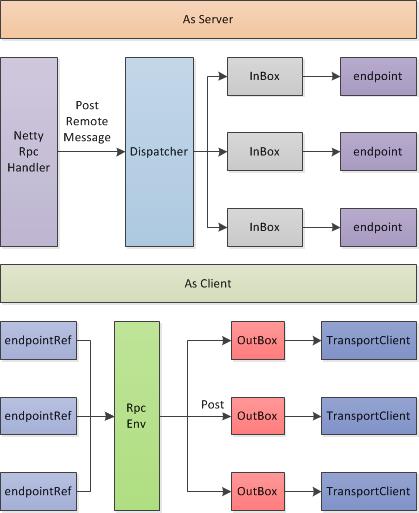


和Inbox类似，Outbox内部也维护了messages用于存储消息，此外还有TransportClient，用于和相应的netty server通信，方法send用于将消息添加到messages中并调用drainOutbox方法消费消息。

在以上的类中Dispatcher,Inbox作用于Server端，分发并处理EndpointRef发送的消息。Outbox作用与Client端，当访问远程Server时，使用TransportClient发送消息，如果Server在同一个程序中直接交给Dispatcher,Inbox，不需要OutBox。

## 2.5 RpcEnv内部实现

RpcEnv是按照MailBox的设计思路来实现，Server端及客户端的实现如下图所示：



RpcEnv即充当Server，也为Client内部实现

1）上半部分AS Server，RpcEnv会初始化一个Server，并注册NettyRpcServer。上面所述RpcHandler的receive接口负责对每一个请求进行处理，一般情况下，简单业务可以在RpcHandler直接完成请求的处理。但是RpcEnv的Server上会挂载很多RpcEndpoint，每个RpcEndpoint的Rpc请求不可控，因此需要Dispatcher分发器对一定的分发机制和队列来维护这些请求，其中Inbox即为请求队列。

在将RpcEndpoint注册到RpcEnv的过程中，也间接的将RpcEnv注册到Dispatcher分发器中，Dispatcher针对每个RpcEndpoint维护一个Inbox，在Dispatcher维持一个线程池，线程针对每个Inbox里面的请求进行处理。实际的处理过程由RpcEndpoint来完成

2） 下半部分Rpc Client的功能实现，RpcEndpointRef是以RpcEndpoint为单位，如果一个进程需要和远程机器的N个RpcEndpoint服务进行通信，就对应N个RpcEndpointRef，当调用一个RpcEndpointRef的ask/send等接口，会将消息内容+RpcEndpointRef+本地地址一起打包成RequestMessage，交由RpcEnv进行发送。

和发送端一样，在RpcEnv中，针对每个remote端的host:port维护一个队列，即Outbox，RpcEnv的发送仅仅是把消息放入到相应的队列中，但是在Outbox中没有维护一个线程池来定时清理OutBox，而是通过一堆synchronized来实现。

# 3.Spark Rpc流程分析

NettyRpcEnv处理消息的主要流程：

1. 创建Dispatcher，后声明线程组来分发消息，如下所示：

*private val threadpool: ThreadPoolExecutor = {*

*val availableCores =*

*if (numUsableCores > 0) numUsableCores else Runtime.getRuntime.availableProcessors()*

*val numThreads = nettyEnv.conf.getInt("spark.rpc.netty.dispatcher.numThreads",*

*math.max(2, availableCores))*

*val pool = ThreadUtils.newDaemonFixedThreadPool(numThreads, "dispatcher-event-loop")*

*for (i <- 0 until numThreads) {*

*pool.execute(new MessageLoop)*

*}*

*pool*

*}*

线程组监控receivers是否有新的EndpointData，执行的线程为MessageLoop，如下所示：

*private class MessageLoop extends Runnable {*

*override def run(): Unit = {*

*try {*

*while (true) {*

*try {*

*val data = receivers.take()*

*if (data == PoisonPill) {*

*// Put PoisonPill back so that other MessageLoops can see it.*

*receivers.offer(PoisonPill)*

*return*

*}*

*data.inbox.process(Dispatcher.this)*

*} .....*

*}*

*}*

1. 如果有消息，并且不为PoisonPill，则调用EndpointData的Inbox的process方法进行消息处理，依次从相应的EndpointData的Inbox的messages中获取第一个元素，匹配消息并调用对应的Endpoint的相应方法进行处理

*message match {*

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

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

*case OneWayMessage(\_sender, content) =>*

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

*case OnStart =>*

*endpoint.onStart()*

*if (!endpoint.isInstanceOf[ThreadSafeRpcEndpoint]) {*

*inbox.synchronized {*

*if (!stopped) {*

*enableConcurrent = true}}}*

*case OnStop =>*

*val activeThreads = inbox.synchronized { inbox.numActiveThreads }*

*dispatcher.removeRpcEndpointRef(endpoint)*

*endpoint.onStop()*

*case RemoteProcessConnected(remoteAddress) =>*

*endpoint.onConnected(remoteAddress)*

*case RemoteProcessDisconnected(remoteAddress) =>*

*endpoint.onDisconnected(remoteAddress)*

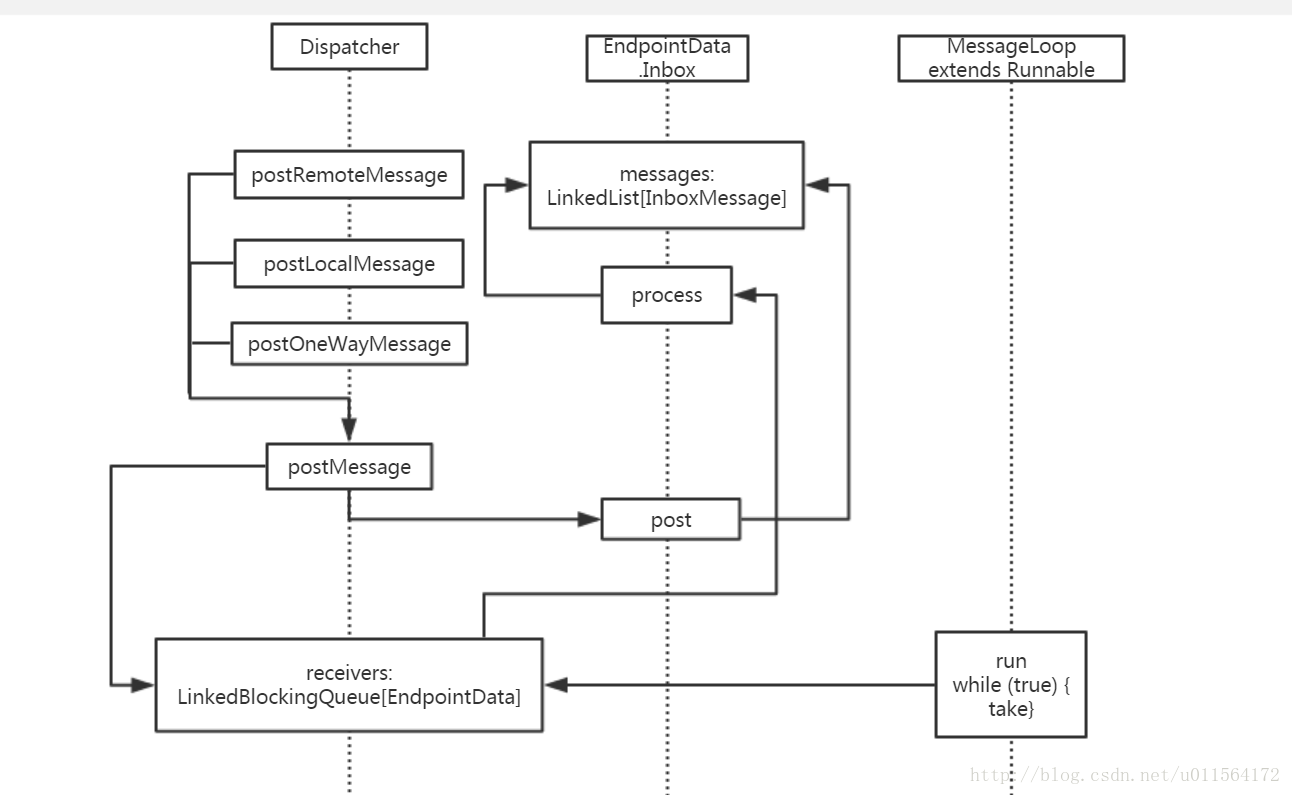
*case RemoteProcessConnectionError(cause, remoteAddress) =>*

*endpoint.onNetworkError(cause, remoteAddress)*

*}*

1. 如果没有消息，则阻塞等待
2. 如果有消息，但是为PoisonPill，则将其继续添加到receivers中，然后停止该线程

Dispatcher处理流程图如下所示：



## 3.1 Spark RPC之Netty启动

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，下图是RpcEnv的几个主要类：



1. RpcEnv（类似于ActorySystem对象），RPC环境，所有的RpcEndpoint需要注册到该对象中用于接受消息，注册时指定name,RpcEnv将会处理从RpcEndpointRef和远程节点发送过来的消息，然后发送给消息的Endpoint处理，对于接收到的异常使用RpcCallContext处理。其实现类为NettyRpcEnv，该类中包含Netty Server及对消息的处理逻辑，NettyRpcEnvFactory用于创建一个RpcEnv。
2. RpcAddress，用来表示主机名和端口号，其伴生对象用于从URL等构造RpcAddress对象
3. RpcTimeout，超时时间，其中awaitResult在规定的时间内返回结果对象，伴生对象主要是配置文件中获取时间值后生成该对象
4. RpcEnvConfig，用于构建RpcEnv的配置对象
5. RpcEndpoint，进程间调用的Endpoint，当一个消息到来时
6. RpcEndpointRef，一个远程RpcEndpoint的引用，通过它可以给远程Endpoint发送消息，可以是同步或者异步，其映射为一个地址，主要的实现类为NettyRpcEndpointRef，其通信端为client:TranspointClient

其中NettyRpcEnv的create，方法如下所示：

*def create(config: RpcEnvConfig): RpcEnv = {*

*val sparkConf = config.conf*

*val javaSerializerInstance = //实例化Java序列化工具*

*new JavaSerializer(sparkConf).newInstance().asInstanceOf[JavaSerializerInstance]*

*val nettyEnv =*

*new NettyRpcEnv(sparkConf, javaSerializerInstance, config.advertiseAddress,*

*config.securityManager, config.numUsableCores)*

*if (!config.clientMode) {*

*val startNettyRpcEnv: Int => (NettyRpcEnv, Int) = { actualPort =>*

*nettyEnv.startServer(config.bindAddress, actualPort) //启动Netty server*

*(nettyEnv, nettyEnv.address.port)*

*}*

*try {*

*Utils.startServiceOnPort(config.port, startNettyRpcEnv, sparkConf, config.name).\_1*

*} ……*

*nettyEnv*

*}*

通过NettyRpcEnv的startServer启动RPC Server，相关类如下所示：

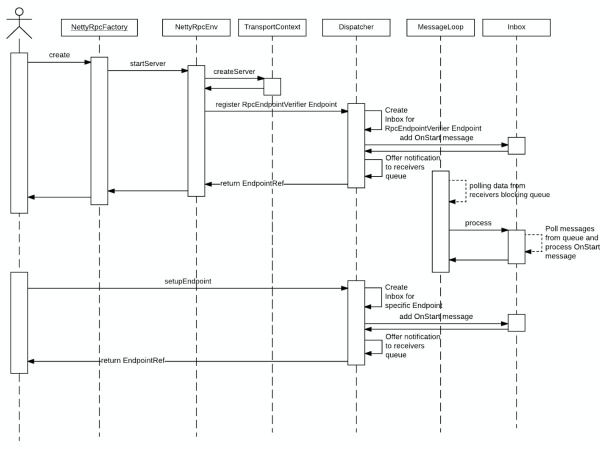


在spark-network-common模块中：

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。

NettyRpcEnv的启动时序图如下所示：



## 3.2 RpcRequest处理请求

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;*

*} …….*

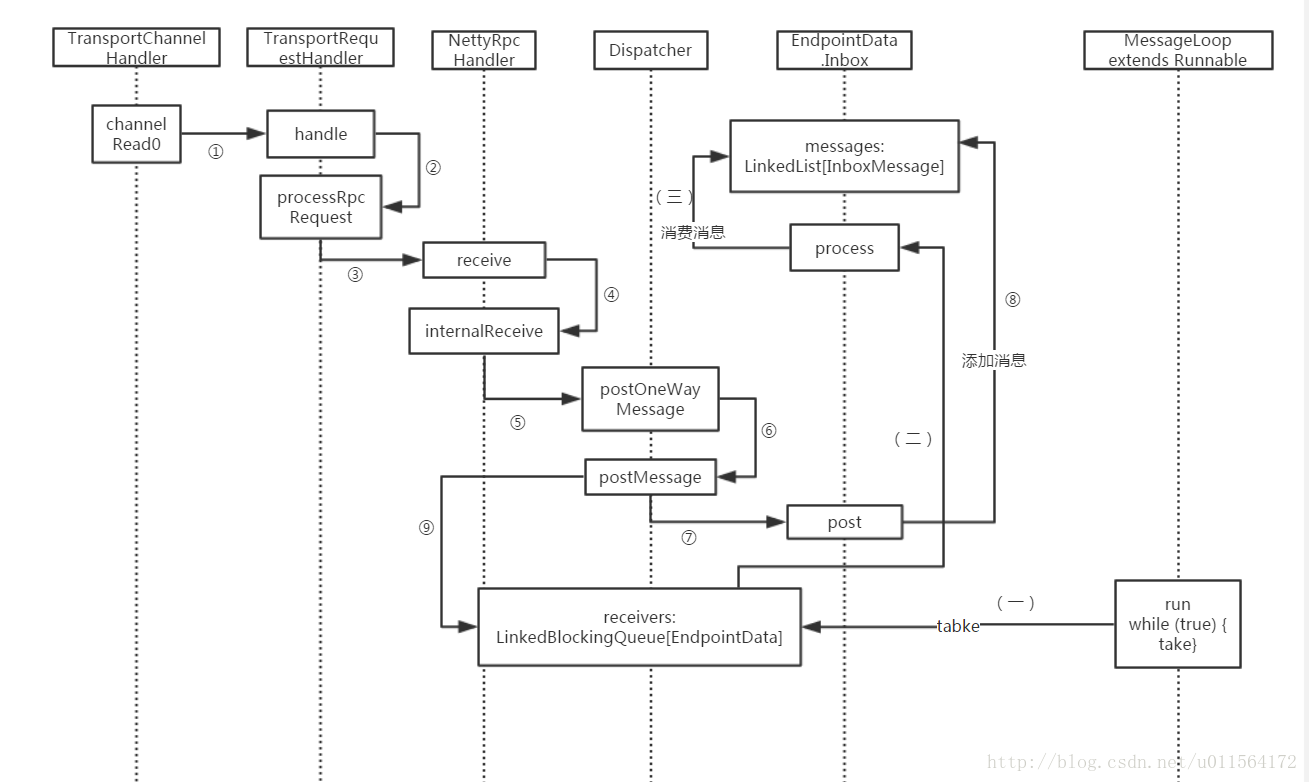
*}*

Handler类图如下所示：



### 3.2.1 RpcEndpointRef和RpcEndpoint同台机器

当RpcEndpointRef和RpcEndpoint不再同一台机器上时，RcpRequest请求处理流程如下所示：



步骤如下：

1. 如上节所述，创建RpcEnv时启动Netty Server，同时将TransportChannelHanlder添加到pipeline中
2. TransportChannel处理Netty接收到数据后，交给TransportRequestHandler后调用processRpcRequest方法，将消息传递给NettyRpcHandler
3. NettyRpcHandler接收到消息后，发送给Dispatcher，进行消息调度，如下所示：

*override def receive(client: TransportClient,message: ByteBuffer,callback: RpcResponseCallback): Unit = {*

*val messageToDispatch = internalReceive(client, message)*

*dispatcher.postRemoteMessage(messageToDispatch, callback)*

*}*

1. Dispatcher进行消息的调度处理，源码如下所示：

*private def postMessage(endpointName: String,message: InboxMessage,*

*callbackIfStopped: (Exception) => Unit): Unit = {*

*val error = synchronized {*

*val data = endpoints.get(endpointName)*

*if (stopped) {*

*Some(new RpcEnvStoppedException())*

*} else if (data == null) {*

*Some(new SparkException(s"Could not find $endpointName."))*

*} else {*

*data.inbox.post(message)*

*receivers.offer(data)*

*None*

*}} }*

1. Dispatcher属于NettyRpcEnv中的对象，其是一直执行的服务，当有消息要处理时调用Inbox处理消息，如下：

*safelyCall(endpoint) {*

*message match {*

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

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

*……)}*

1. Inbox调用RpcEndpoint处理消息，如下所示:

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

*……*

*case \_ => context.sendFailure(new SparkException(self + " won't reply anything"))*

*}*

1. Inbox获取根据RpcEndpointRef中传过来的endpointName进行message的路由处理，流程图如下所示：



1. 首先RpcEnv在创建时将Endpoint注册到Dispatcher中代码分析如下：

*def registerRpcEndpoint(name: String, endpoint: RpcEndpoint): NettyRpcEndpointRef = {*

*val addr = RpcEndpointAddress(nettyEnv.address, name)*

*val endpointRef = new NettyRpcEndpointRef(nettyEnv.conf, addr, nettyEnv)*

*synchronized {*

*if (endpoints.putIfAbsent(name, new EndpointData(name, endpoint, endpointRef)) != null) {}*

*val data = endpoints.get(name)*

*endpointRefs.put(data.endpoint, data.ref)*

*receivers.offer(data) // for the OnStart message*

*}*

*endpointRef*

*}*

b）客户端在与NettyServer通信时，会传输Endpoint的name，如下所示：

*val endPointRef: RpcEndpointRef = rpcEnv.setupEndpointRef(RpcAddress("localhost", 52345), "HelloService")*

*val future: Future[String] = endPointRef.ask[String](SayHi("neo"))*

将endpointName封装到RpcEndpointAddress中：

*def setupEndpointRef(address: RpcAddress, endpointName: String): RpcEndpointRef = {*

*setupEndpointRefByURI(RpcEndpointAddress(address, endpointName).toString)*

*}*

RpcEndpointRef在ask进行通信时，将message:Any与Endpoint name封装到RequestMessage中，如NettyRpcEndpointRef#ask所示：

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

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

*}*

对RequestMessage进行序列化后形成ByteBuffer后，传给NettyRpcEnv的dispatcher。

1. Dispatcher接收到TransportRequesthandler发送过来的消息后，对ByteBuffer进行反序列化后Endpoint的name及Message，NettyRpcEnv中对ByteBuffer进行反序列化：

*override def receive(*

*client: TransportClient,*

*message: ByteBuffer,*

*callback: RpcResponseCallback): Unit = {*

*val messageToDispatch = internalReceive(client, message)*

*dispatcher.postRemoteMessage(messageToDispatch, callback)*

*}*

最终执行的操作是NettyRpcEnv#RequestMessage.apply

*def apply(nettyEnv: NettyRpcEnv, client: TransportClient, bytes: ByteBuffer): RequestMessage = {*

*val bis = new ByteBufferInputStream(bytes)*

*val in = new DataInputStream(bis)*

*try {*

*val senderAddress = readRpcAddress(in)*

*val endpointAddress = RpcEndpointAddress(readRpcAddress(in), in.readUTF())*

*val ref = new NettyRpcEndpointRef(nettyEnv.conf, endpointAddress, nettyEnv)*

*ref.client = client*

*new RequestMessage(*

*senderAddress,*

*ref,*

*// The remaining bytes in `bytes` are the message content.*

*nettyEnv.deserialize(client, bytes))*

*} finally {*

*in.close()*

*}*

*}*

1. 首先根据name获取对应的Endpoint，如下所示：

*val data = endpoints.get(endpointName)*

*if (stopped) {*

*Some(new RpcEnvStoppedException())*

*} else if (data == null) {*

*Some(new SparkException(s"Could not find $endpointName."))*

*} else {*

*data.inbox.post(message)*

*receivers.offer(data)*

*None*

*}*

1. Dispatcher最终会调用对应的Endpoint进行Message的处理，Inbox调用RpcEndpoint处理消息，如下所示:

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

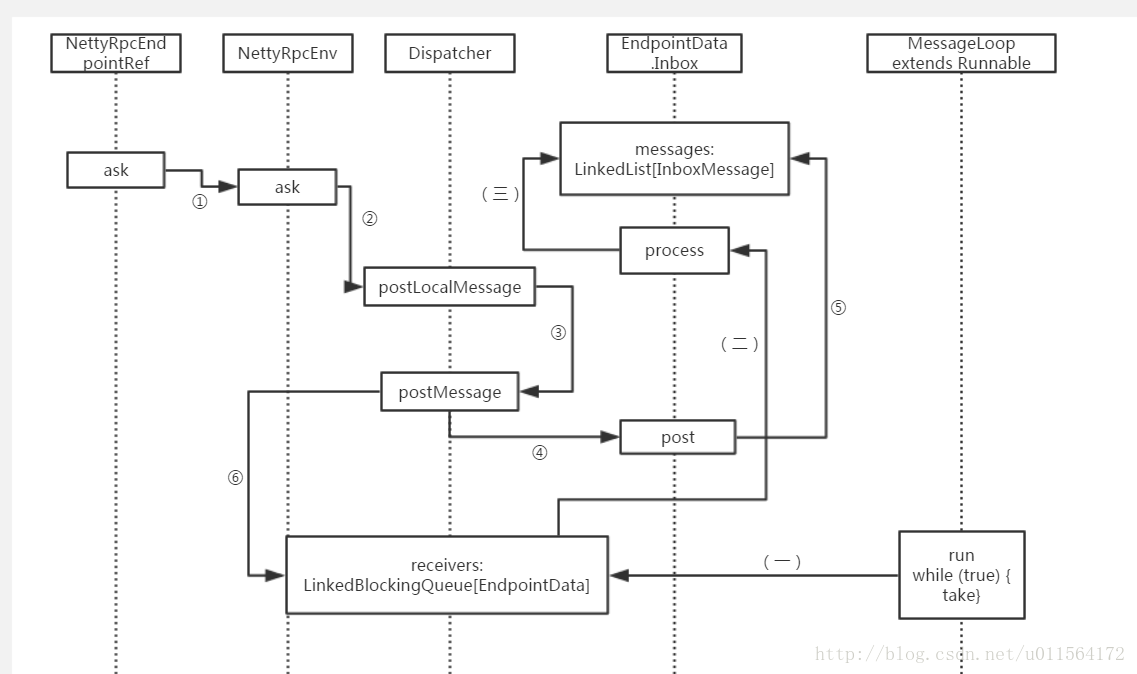
*……*

*case \_ => context.sendFailure(new SparkException(self + " won't reply anything"))*

*}*

### 3.2.2 RpcEndpointRef和RpcEndpoint同机器

当RpcEndpointRef和RpcEndpoing位于同一机器上时，访问过程如下所示：



前面的处理过程基本相同，不同处是NettyRpcEnv在接收到消息后，判断是否是本地消息，如下所示：

*if (remoteAddr == address) {*

*val p = Promise[Any]()*

*p.future.onComplete {*

*case Success(response) => onSuccess(response)*

*case Failure(e) => onFailure(e)*

*}(ThreadUtils.sameThread)*

*dispatcher.postLocalMessage(message, p)*

*}*

与远程EndpointRef不同的是调用postLocalMessage进行处理，不需要对消息进行封装，处理过程如下：

*def postLocalMessage(message: RequestMessage, p: Promise[Any]): Unit = {*

*val rpcCallContext =*

*new LocalNettyRpcCallContext(message.senderAddress, p)*

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

*postMessage(message.receiver.name, rpcMessage, (e) => p.tryFailure(e))*

*}*

其余过程相同，不再介绍。

## 3.3 Spark RPC客户端处理流程

以上分析均是NettyRpcServer接收到RpcRequest请求后进行的处理流程，客户端需要先建立RpcEnv，然后获取RpcEndpointRef，执行流程如下所示：



1）客户端创建RpcEnv，获取远端要通信的EndpointRef

2）调用EndpointRef.ask或者send向远程RpcServer发送请求，将请求封装成RequestOutboxMessage，序列化后发送到指定地址的Outbox中

3）Outbox如果发现未建立连接则建立连接，调用TransportClient发送数据，完成后即可返回（requestID），返回一个Future后，客户端可以阻塞或者做其他操作

4）TransportResponseHandler接收到远程的响应后，先反序列化，然后回调前面阶段的Future，完成调用。

# 4.Spark RpcEndpoint

Spark各组件（Master、Worker、Driver及Executor等）之间的交互都是通过Rpc来实现，其中核心是RpcEndpoint，类关系如下所示：



1. Master和Worker，为Spark Standalone模式下Master与Slave之间的交互
2. BlockManagerMasterEndpoint和BlockManagerSlaveEndpoint，以远程消息通信的方式管理所有节点的BlockManager
3. ClientEndpoint，客户端与Driver传递消息
4. DriverEndpoint，代表DriverEndpoint，负责Driver与Executor之间的交互
5. HeartbeatReceiver，用于Driver从Executor之间心跳交互
6. CoarseGrainedExecutorBackend，用于Executor与其它组件的交互
7. ReceiverTrackerEndpoint，Spark Streaming中从Receivers中接收消息

RpcEndpoint与Spark各组件之间的关系如下图所示：



## 4.1 Master与Worker

Spark在基于Standalone模式下，Master和多Worker构成Spark集群。Master和Worker之间的通信是基于Rpc，在Master端，RpcEndpoint的代码如下所示：

*def startRpcEnvAndEndpoint(*

*host: String,port: Int,webUiPort: Int,*

*conf: SparkConf): (RpcEnv, Int, Option[Int]) = {*

*val securityMgr = new SecurityManager(conf)*

*val rpcEnv = RpcEnv.create(SYSTEM\_NAME, host, port, conf, securityMgr)*

*val masterEndpoint = rpcEnv.setupEndpoint(ENDPOINT\_NAME,*

*new Master(rpcEnv, rpcEnv.address, webUiPort, securityMgr, conf))*

*val portsResponse = masterEndpoint.askSync[BoundPortsResponse](BoundPortsRequest)*

*(rpcEnv, portsResponse.webUIPort, portsResponse.restPort)*

*}*

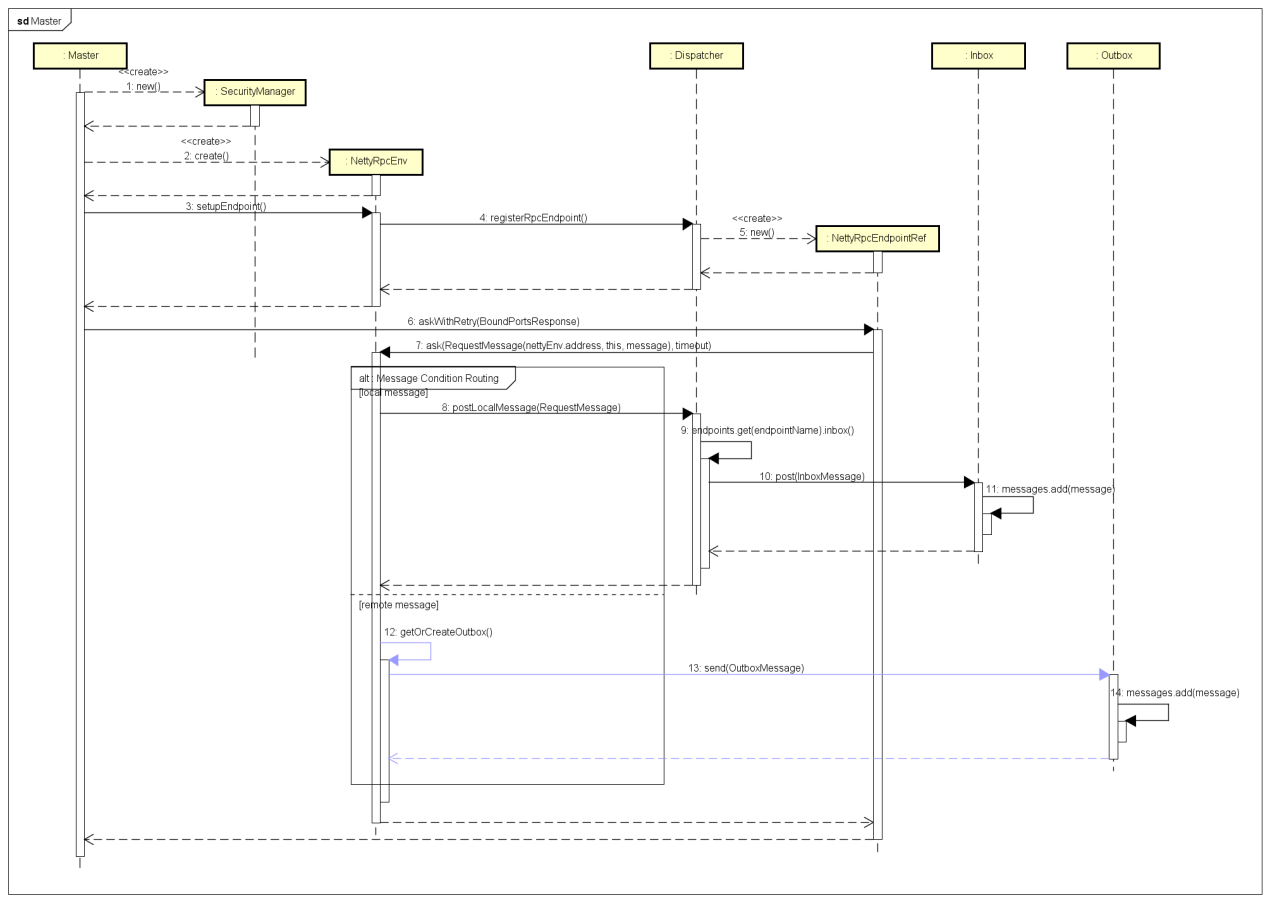
Master RpcEndpoint接收到BoudPortsRequest后，调用receiveAndReply方法：

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

*case BoundPortsRequest =>*

*context.reply(BoundPortsResponse(address.port, webUi.boundPort, restServerBoundPort))*

发送BoundPortsRequest消息，具体的消息路由过程如下图所示：



本地消息和远程消息派发的流程，最主要的区别是在接收消息时走的是Inbox，发送消息走的是Outbox。

### 4.1.1 本地消息路由

Master启动后，调用startRpcEnvAndEndpoint，将Endpoint注册到NettyRpcEnv的dispatcher中，在启动后发送BoundPortsRequest消息，走的是本地消息路由，直接放到对应的Inbox中，对应的代码逻辑如下所示：

*private def postMessage(endpointName: String,message: InboxMessage,*

*callbackIfStopped: (Exception) => Unit): Unit = {*

*val error = synchronized {*

*val data = endpoints.get(endpointName)*

*if (stopped) {*

*Some(new RpcEnvStoppedException())*

*} else if (data == null) {*

*Some(new SparkException(s"Could not find $endpointName."))*

*} else {*

*data.inbox.post(message)*

*receivers.offer(data)*

*None*

*}*

*}*

上面通过data.inbox派发消息，然后将data:EndpointData放入到receivers队列，触发Dispatcher内部的MessageLoop线程去消费：

*private de private class MessageLoop extends Runnable {*

*override def run(): Unit = {*

*try {*

*while (true) {*

*try {*

*val data = receivers.take()*

*if (data == PoisonPill) {*

*receivers.offer(PoisonPill)*

*return*

*}*

*data.inbox.process(Dispatcher.this)*

*} ……*

*}*

这里调用Inbox的process方法来派发消息到指定的RpcEndpoint，根据上面的时序图，原始消息被封装为一个RpcMessage，该消息在Inbox的process方法中处理派发逻辑，如下所示：

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

*try {*

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

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

*})*

*} catch {......*

*}*

消息已经发送给对应的RpcEndpoint的receiveAndReply方法，实际上是Master实现类，这里的消息解包后为content：BoundPortsRequest，Master的receiveAndReply方法处理该本地消息，代码如下所示：

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

*......*

*case BoundPortsRequest =>*

*context.reply(BoundPortsResponse(address.port, webUi.boundPort, restServerBoundPort))*

*......*

*}*

Master对于BoundPortsResponse返回Master端的几个端口数据。

### 4.1.2 远程消息路由

Worker启动时，会首先获取到一个Master的RpcEndpointRef远程引用，通过该对象引用能够与Master进行RPC通信，如下所示：

*def startRpcEnvAndEndpoint(*

*host: String, port: Int,webUiPort: Int,*

*cores: Int, memory: Int,*

*masterUrls: Array[String],*

*workDir: String,workerNumber: Option[Int] = None,conf: SparkConf = new SparkConf): RpcEnv = {*

*val systemName = SYSTEM\_NAME + workerNumber.map(\_.toString).getOrElse("")*

*val securityMgr = new SecurityManager(conf)*

*val rpcEnv = RpcEnv.create(systemName, host, port, conf, securityMgr)*

*val masterAddresses = masterUrls.map(RpcAddress.fromSparkURL(\_))*

*rpcEnv.setupEndpoint(ENDPOINT\_NAME, new Worker(rpcEnv, webUiPort, cores, memory,*

*masterAddresses, ENDPOINT\_NAME, workDir, conf, securityMgr))*

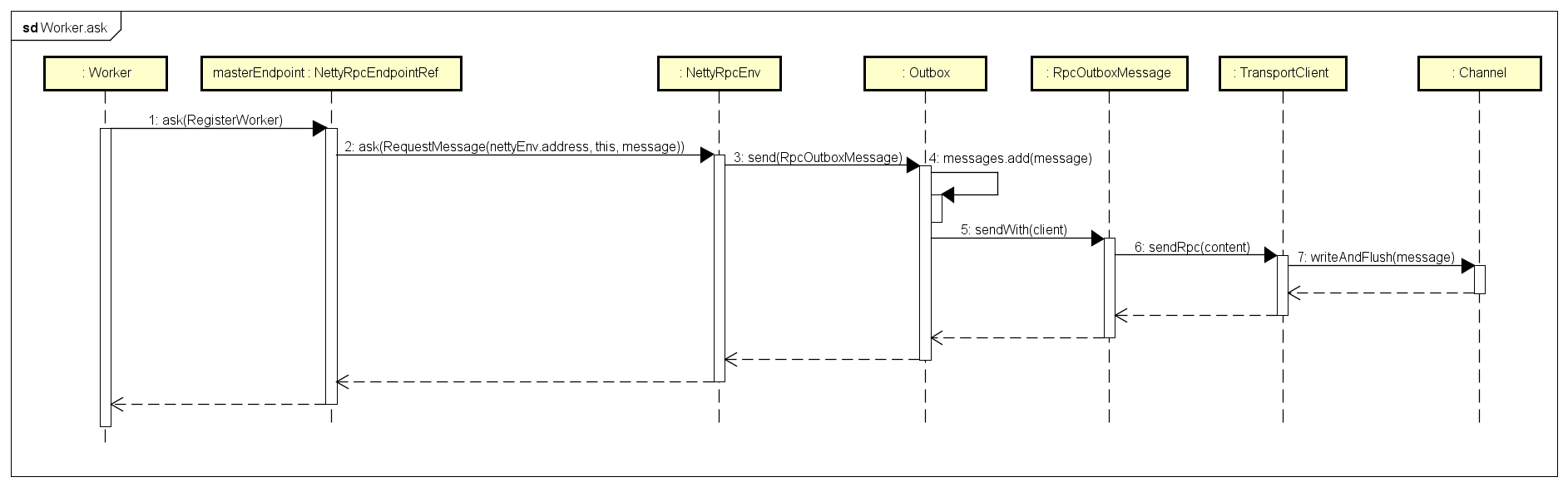
*rpcEnv*

*}*

获取Master的RpcEndpointRef远程引用代码如下：

*val masterEndpoint = rpcEnv.setupEndpointRef(masterAddress, Master.ENDPOINT\_NAME)*

Worker启动后通过sendRegisterMessageToMaster会向Master注册，过程如下图所示：



Worker调用send(RegisterWorker)方法，与Master进行通信。Outbox调用sendWith将消息发送给Master：

*override def sendWith(client: TransportClient): Unit = {*

*client.send(content)*

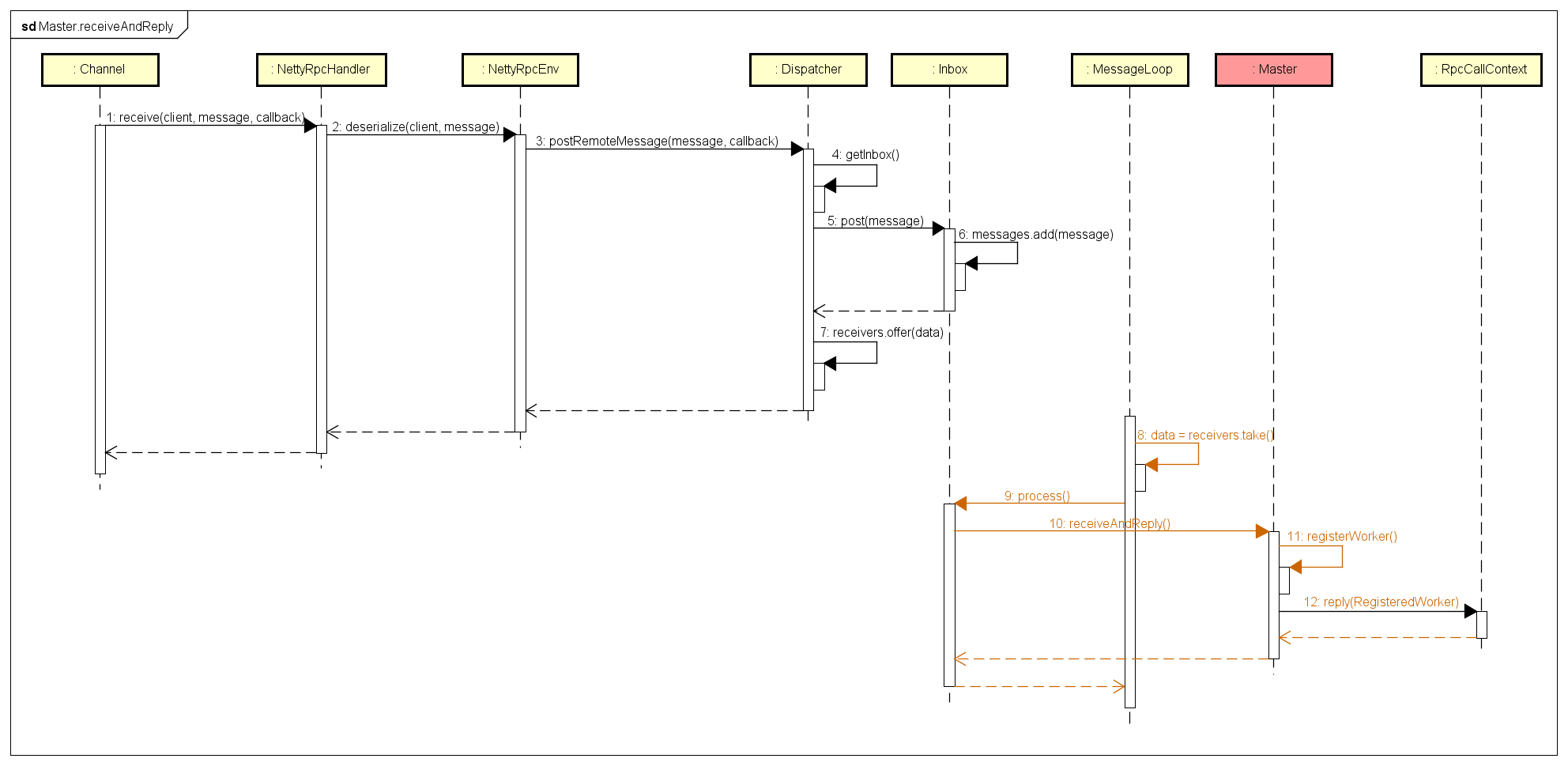
*}*

*TransportClient#send(ByteBuffer message) {*

*channel.writeAndFlush(new OneWayMessage(new NioManagedBuffer(message)));*

*}*

终通过Netty的channel将消息发送给远程Master端，RegisterWorker最终会被路由到Master对应的Inbox中，然后派发给Master进行处理，下面是Master端接收并处理消息的过程，如下图所示：



上图中分为两个部分：

1. 从远端接收消息RegisterWorker，将接收到的消息放入到Inbox中
2. 触发MessageLoop线程处理该消息，从而调用Inbox的process方法，继续调用RpcEndpoint(master)的receive方法，处理RegisterWorker，如下所示：

*case RegisterWorker(*

*id, workerHost, workerPort, workerRef, cores, memory, workerWebUiUrl, masterAddress) =>*

*if (state == RecoveryState.STANDBY) {*

*workerRef.send(MasterInStandby)*

*} else if (idToWorker.contains(id)) {*

*workerRef.send(RegisterWorkerFailed("Duplicate worker ID"))*

*} else {*

*val worker = new WorkerInfo(id, workerHost, workerPort, cores, memory,*

*workerRef, workerWebUiUrl)*

*if (registerWorker(worker)) {*

*persistenceEngine.addWorker(worker)*

*workerRef.send(RegisteredWorker(self, masterWebUiUrl, masterAddress))*

*schedule()*

*} else {*

*val workerAddress = worker.endpoint.address*

*workerRef.send(RegisterWorkerFailed("Attempted to re-register worker at same address: "*

*+ workerAddress))*

*}*

*}*

如果Worker注册成功，则Master会通过WorkerRef#send回复Worker响应：

*workerRef.send(RegisteredWorker(self, masterWebUiUrl, masterAddress))*

Worker收到Master回复后，调用handleRegisterResponse

*private def handleRegisterResponse(msg: RegisterWorkerResponse): Unit = synchronized {*

*msg match {*

*case RegisteredWorker(masterRef, masterWebUiUrl, masterAddress) =>*

*registered = true*

*changeMaster(masterRef, masterWebUiUrl, masterAddress)*

*forwordMessageScheduler.scheduleAtFixedRate(new Runnable {*

*override def run(): Unit = Utils.tryLogNonFatalError {*

*self.send(SendHeartbeat)*

*}*

*}, 0, HEARTBEAT\_MILLIS, TimeUnit.MILLISECONDS)*

*val execs = executors.values.map { e =>*

*new ExecutorDescription(e.appId, e.execId, e.cores, e.state)*

*}*

*masterRef.send(WorkerLatestState(workerId, execs.toList, drivers.keys.toSeq))*

*case RegisterWorkerFailed(message) =>*

*if (!registered) {*

*logError("Worker registration failed: " + message)*

*System.exit(1)*

*}*

*case MasterInStandby =>*

*// Ignore. Master not yet ready.*

*}*

*}*

## 4.2 Driver与Executor之间心跳交互（HeartbeatReceiver）

Driver通过HeartbeatReceiver与Executor进行心跳交互，注册如下：

*\_heartbeatReceiver = env.rpcEnv.setupEndpoint(*

*HeartbeatReceiver.ENDPOINT\_NAME, new HeartbeatReceiver(this))*

Executor启动后获取HeartbeatReceiverRef，如下所示：

*private val heartbeatReceiverRef =*

*RpcUtils.makeDriverRef(HeartbeatReceiver.ENDPOINT\_NAME, conf, env.rpcEnv)*

启动HeartbeatTask后，线程启动如下：

*private def startDriverHeartbeater(): Unit = {*

*val intervalMs = conf.getTimeAsMs("spark.executor.heartbeatInterval", "10s")*

*val initialDelay = intervalMs + (math.random \* intervalMs).asInstanceOf[Int]*

*val heartbeatTask = new Runnable() {*

*override def run(): Unit = Utils.logUncaughtExceptions(reportHeartBeat())*

*}*

*heartbeater.scheduleAtFixedRate(heartbeatTask, initialDelay, intervalMs, TimeUnit.MILLISECONDS)*

*}*

执行流程如下所示：



Executor调用askSync(message:Heartbeat)将消息发送给Driver ，代码如下所示：

*val message = Heartbeat(executorId, accumUpdates.toArray,env.blockManager.blockManagerId)*

*try {*

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

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

*if (response.reregisterBlockManager) {*

*logInfo("Told to re-register on heartbeat")*

*env.blockManager.reregister()*

*}*

*heartbeatFailures = 0*

*}*

通过Netty的channel将消息发送给远程Driver端，Heartbeat最终会被路由到Driver对应的Inbox中，然后派发给Driver进行处理，

Driver收到Executor的Heartbeat消息后，进行注册：

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

*if (scheduler != null) {*

*if (executorLastSeen.contains(executorId)) {*

*executorLastSeen(executorId) = clock.getTimeMillis()*

*eventLoopThread.submit(new Runnable {*

*override def run(): Unit = Utils.tryLogNonFatalError {*

*val unknownExecutor = !scheduler.executorHeartbeatReceived(*

*executorId, accumUpdates, blockManagerId)*

*val response = HeartbeatResponse(reregisterBlockManager = unknownExecutor)*

*context.reply(response)*

*}*

*})*

*} else {*

*context.reply(HeartbeatResponse(reregisterBlockManager = true))*

*}*

*}*

## 4.3 ClientEndpoint，Standalone模式

ClientEndpoint用于客户端与Master之间的消息交互，用于在Standalone模式下启动或者停止drivers，在Client中初始化如下：

*val rpcEnv =*

*RpcEnv.create("driverClient", Utils.localHostName(), 0, conf, new SecurityManager(conf))*

*val masterEndpoints = driverArgs.masters.map(RpcAddress.fromSparkURL).*

*map(rpcEnv.setupEndpointRef(\_, Master.ENDPOINT\_NAME))*

*rpcEnv.setupEndpoint("client", new ClientEndpoint(rpcEnv, driverArgs, masterEndpoints, conf))*

启动时，根据driverArgs.cmd进行Driver的操作，如下所示：

override def onStart(): Unit = {

*driverArgs.cmd match {*

*case "launch" =>*

*val mainClass = "org.apache.spark.deploy.worker.DriverWrapper"*

*val classPathConf = "spark.driver.extraClassPath"*

*val classPathEntries = sys.props.get(classPathConf).toSeq.flatMap { cp =>*

*cp.split(java.io.File.pathSeparator)*

*}*

*val libraryPathConf = "spark.driver.extraLibraryPath"*

*val libraryPathEntries = sys.props.get(libraryPathConf).toSeq.flatMap { cp =>*

*cp.split(java.io.File.pathSeparator)*

*}*

*val extraJavaOptsConf = "spark.driver.extraJavaOptions"*

*val extraJavaOpts = sys.props.get(extraJavaOptsConf)*

*.map(Utils.splitCommandString).getOrElse(Seq.empty)*

*val sparkJavaOpts = Utils.sparkJavaOpts(conf)*

*val javaOpts = sparkJavaOpts ++ extraJavaOpts*

*val command = new Command(mainClass,*

*Seq("{{WORKER\_URL}}", "{{USER\_JAR}}", driverArgs.mainClass) ++ driverArgs.driverOptions,*

*sys.env, classPathEntries, libraryPathEntries, javaOpts)*

*val driverDescription = new DriverDescription(*

*driverArgs.jarUrl,*

*driverArgs.memory,*

*driverArgs.cores,*

*driverArgs.supervise,*

*command)*

*ayncSendToMasterAndForwardReply[SubmitDriverResponse](*

*RequestSubmitDriver(driverDescription))*

*case "kill" =>*

*val driverId = driverArgs.driverId*

*ayncSendToMasterAndForwardReply[KillDriverResponse](RequestKillDriver(driverId))*

*}*

Master接收到RequestSubmitDriver或者RequestKillDriver后，

*case RequestSubmitDriver(description) =>*

*if (state != RecoveryState.ALIVE) {*

*context.reply(SubmitDriverResponse(self, false, None, msg))*

*} else {*

*val driver = createDriver(description)*

*persistenceEngine.addDriver(driver)*

*waitingDrivers += driver*

*drivers.add(driver)*

*schedule()*

*context.reply(SubmitDriverResponse(self, true, Some(driver.id),*

*s"Driver successfully submitted as ${driver.id}"))*

*}*

https://www.cnblogs.com/yourarebest/p/5313006.html

## 4.4 DriverEndpoint，YARN模式

SparkContext中CoarseGrainedSchedulerBackend和CM交互，请求和kill掉Executor，当前仅支持YARN 模式。当CoarseGrainedExecutorBackend在实例化后调用onStart后向DriverEndpoint发送RegisterExecutor进行注册，DriverEndpoint收到注册信息在Driver中进行保存从而获取计算资源。CoarseGrainedSchedulerBackend启动后，获取DriverEndpoint，如下：

*rpcEnv.asyncSetupEndpointRefByURI(driverUrl).flatMap { ref =>*

*driver = Some(ref)*

*ref.ask[Boolean](RegisterExecutor(executorId, self, hostname, cores, extractLogUrls))*

*}*

流程如下所示：



### 4.4.1 Executor与DriverEndpoint

CoarseGrainedExecutorBackend在调用driver.ask[RegisterExecutor]后，将消息发送到DriverEndpoint<CoarseGrainedSchedulerBackend>，执行代码如下：

*case RegisterExecutor(executorId, executorRef, hostname, cores, logUrls) =>*

*if (executorDataMap.contains(executorId)) {*

*executorRef.send(RegisterExecutorFailed("Duplicate executor ID: " + executorId))*

*context.reply(true)*

*} else if (scheduler.nodeBlacklist != null &&*

*scheduler.nodeBlacklist.contains(hostname)) {*

*executorRef.send(RegisterExecutorFailed(s"Executor is blacklisted: $executorId"))*

*context.reply(true)*

*} else {*

*val executorAddress = if (executorRef.address != null) {*

*executorRef.address*

*} else {*

*context.senderAddress*

*}*

*logInfo(s"Registered executor $executorRef ($executorAddress) with ID $executorId")*

*addressToExecutorId(executorAddress) = executorId*

*totalCoreCount.addAndGet(cores)*

*totalRegisteredExecutors.addAndGet(1) //将Executor添加到Driver中*

*val data = new ExecutorData(executorRef, executorRef.address, hostname,*

*cores, cores, logUrls)*

*CoarseGrainedSchedulerBackend.this.synchronized {*

*executorDataMap.put(executorId, data)*

*if (currentExecutorIdCounter < executorId.toInt) {*

*currentExecutorIdCounter = executorId.toInt*

*}*

*if (numPendingExecutors > 0) {*

*numPendingExecutors -= 1*

*logDebug(s"Decremented number of pending executors ($numPendingExecutors left)")*

*}*

*}*

*executorRef.send(RegisteredExecutor)*

*// Note: some tests expect the reply to come after we put the executor in the map*

*context.reply(true)*

*listenerBus.post(*

*SparkListenerExecutorAdded(System.currentTimeMillis(), executorId, data))*

*makeOffers()*

*}*

### 4.4.2 TaskScheduler与DriverEndpoint

在Driver内部，调用createDriverEndpointRef创建driverEndpoint，如下所示：

*protected def createDriverEndpointRef(*

*properties: ArrayBuffer[(String, String)]): RpcEndpointRef = {*

*rpcEnv.setupEndpoint(ENDPOINT\_NAME, createDriverEndpoint(properties))*

*}*

TaskScheduler调用reviveOffers申请资源并执行task，调用CoarseGrainedSchedulerBackend，如下所示：

*override def reviveOffers() {*

*driverEndpoint.send(ReviveOffers)*

*}*

DriverEndpoint中对ReviveOffers的处理如下：

*override def receive: PartialFunction[Any, Unit] = {*

*case ReviveOffers =>*

*makeOffers()*

*}*

makeOffers在所有的Executors上构建resource offers，如下所示：

*private def makeOffers() {*

*// Make sure no executor is killed while some task is launching on it*

*val taskDescs = CoarseGrainedSchedulerBackend.this.synchronized {*

*// Filter out executors under killing*

*val activeExecutors = executorDataMap.filterKeys(executorIsAlive)*

*val workOffers = activeExecutors.map {*

*case (id, executorData) =>*

*new WorkerOffer(id, executorData.executorHost, executorData.freeCores)*

*}.toIndexedSeq*

*scheduler.resourceOffers(workOffers)*

*}*

*if (!taskDescs.isEmpty) {*

*launchTasks(taskDescs)*

*}*

*}*

## 4.5 BlockManagerMaster与BlockManagerSlave

在Spark中Block是数据处理的基本单位，BlockManager对整个集群的Block数据进行管理。BlockManagerMasterEndpoint本身是一个消息循环体，负责通过远程消息通信的方式去管理所有节点的BlockManager，在SparkEnv中BlockManagerMasterEndpoint的实例如下所示：

*val blockManagerMaster = new BlockManagerMaster(registerOrLookupEndpoint(*

*BlockManagerMaster.DRIVER\_ENDPOINT\_NAME,*

*new BlockManagerMasterEndpoint(rpcEnv, isLocal, conf, listenerBus)),*

*conf, isDriver)*

在BlockManagerMaster实例化时产生，并开始接受其他BlockManager向其发送消息。在ExecutorBackend启动时会实例化BlockManager，并通过远程通信的方式注册给BlockManagerMaster，如下所示：

*val env = SparkEnv.createExecutorEnv(*

*driverConf, executorId, hostname, cores, cfg.ioEncryptionKey, isLocal = false)*

在SparkEnv中会创建BlockManager

*val blockManagerMaster = new BlockManagerMaster(registerOrLookupEndpoint(*

*BlockManagerMaster.DRIVER\_ENDPOINT\_NAME,*

*new BlockManagerMasterEndpoint(rpcEnv, isLocal, conf, listenerBus)),*

*conf, isDriver)*

*val blockManager = new BlockManager(executorId, rpcEnv, blockManagerMaster,*

*serializerManager, conf, memoryManager, mapOutputTracker, shuffleManager,*

*blockTransferService, securityManager, numUsableCores)*

在BlockManager中的BlockManagerMaster的初始化过程中会获取Driver端的BlockManagerMaster：

*def registerOrLookupEndpoint(*

*name: String, endpointCreator: => RpcEndpoint):*

*RpcEndpointRef = {*

*if (isDriver) {*

*logInfo("Registering " + name)*

*rpcEnv.setupEndpoint(name, endpointCreator)*

*} else {*

*RpcUtils.makeDriverRef(name, conf, rpcEnv)*

*}*

*}*

BlockManager启动后，会将BlockManagerSlaveEndpoint注册到NettyRpcEnv的Dispatcher中：

*private val slaveEndpoint = rpcEnv.setupEndpoint(*

*"BlockManagerEndpoint" + BlockManager.ID\_GENERATOR.next,*

*new BlockManagerSlaveEndpoint(rpcEnv, this, mapOutputTracker))*

### 4.5.1 BlockManager的注册

BlockManager在初始化后向BlockManagerMaster发送RegisterBlockManager，流程图如下所示：



### 4.5.2 BlockManager向BlockManagerMaster汇报Block信息

Executor与Driver发送心跳的过程中，触发BlockManager向BlockManagerMaster汇报Block信息，register如下所示：

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

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

*if (response.reregisterBlockManager) {*

*env.blockManager.reregister()*

*}*

BlockManager.reregister过程如下：

*def reregister(): Unit = {*

*master.registerBlockManager(blockManagerId, maxOnHeapMemory, maxOffHeapMemory, slaveEndpoint)*

*reportAllBlocks()*

*}*

在reportAllBlocks中，向BlockManagerMaster发送UpdateBlockInfo消息，如下所示：

*def updateBlockInfo(*

*blockManagerId: BlockManagerId,*

*blockId: BlockId,*

*storageLevel: StorageLevel,*

*memSize: Long,*

*diskSize: Long): Boolean = {*

*val res = driverEndpoint.askSync[Boolean](*

*UpdateBlockInfo(blockManagerId, blockId, storageLevel, memSize, diskSize))*

*res*

*}*

BlockManagerMasterEndpoint接收到UpdateBlockInfo后，进行Block Info的更新，处理如下：

*case \_updateBlockInfo @*

*UpdateBlockInfo(blockManagerId, blockId, storageLevel, deserializedSize, size) =>*

*context.reply(updateBlockInfo(blockManagerId, blockId, storageLevel, deserializedSize, size))*

*listenerBus.post(SparkListenerBlockUpdated(BlockUpdatedInfo(\_updateBlockInfo)))*

## 4.6 MapOutputTracker

MapOutputTracker用于跟踪map阶段任务的输出状态，此状态便于reduce阶段获取地址及中间输出结果。MapOutputTracker的启动如下所示：

*val mapOutputTracker = if (isDriver) {*

*new MapOutputTrackerMaster(conf, broadcastManager, isLocal)*

*} else {*

*new MapOutputTrackerWorker(conf)*

*}*

*mapOutputTracker.trackerEndpoint = registerOrLookupEndpoint(MapOutputTracker.ENDPOINT\_NAME,*

*new MapOutputTrackerMasterEndpoint(*

*rpcEnv, mapOutputTracker.asInstanceOf[MapOutputTrackerMaster], conf))*

在Driver中mapOutputTracker实现类为MapOutputTrackerMaster，并将其注册到Dispatcher中：

*mapOutputTracker.trackerEndpoint = registerOrLookupEndpoint(MapOutputTracker.ENDPOINT\_NAME,*

*new MapOutputTrackerMasterEndpoint(*

*rpcEnv, mapOutputTracker.asInstanceOf[MapOutputTrackerMaster], conf))*

在Executor中，获取Driver中的MapOutputTracker RcpEndpointRef，调用的代码如上述相同，但是根据是否是Driver，进行的流程不同，如下所示：

*def registerOrLookupEndpoint(*

*name: String, endpointCreator: => RpcEndpoint):*

*RpcEndpointRef = {*

*if (isDriver) {*

*rpcEnv.setupEndpoint(name, endpointCreator)*

*} else {*

*RpcUtils.makeDriverRef(name, conf, rpcEnv)*

*}*

*}*

在Task结束后，DAGScheduler将Task的输出注册到shuffleStatus中，如下：

*event.reason match {*

*case Success =>*

*task match {*

*......*

*case smt: ShuffleMapTask =>*

*mapOutputTracker.registerMapOutput(*

*shuffleStage.shuffleDep.shuffleId, smt.partitionId, status）*

*shuffleStage.pendingPartitions -= task.partitionId*

*}*

*}*

Executor在执行过程中通过MapOutputTrackerMasterEndpoint从Driver中获取map输出信息，如下所示：

*val fetchedBytes = askTracker[Array[Byte]](GetMapOutputStatuses(shuffleId))*

*fetchedStatuses = MapOutputTracker.deserializeMapStatuses(fetchedBytes)*

*mapStatuses.put(shuffleId, fetchedStatuses)*

在MapOutputTrackerMaster中处理如下：

*case GetMapOutputStatuses(shuffleId: Int) =>*

*val hostPort = context.senderAddress.hostPort*

*val mapOutputStatuses = tracker.post(new GetMapOutputMessage(shuffleId, context))*

## 4.6 ReceiveTracker

ReceiverTracker负责启动、管理各个Receiver及从Receiver接收到的数据。在ReceiverTracker启动时，会初始化ReceiverTrackeEndpoint，用来处理来自ReceiverTracker和Receiver接收到的消息，如下所示：

*endpoint = ssc.env.rpcEnv.setupEndpoint(*

*"ReceiverTracker", new ReceiverTrackerEndpoint(ssc.env.rpcEnv))*

*if (!skipReceiverLaunch) launchReceivers()*

*trackerState = Started*

其中的launchReceivers将各个receivers分发到Executors上。

Receivers的运行进程为ReceiverSupervisorImpl，运行过程中将接收到的数据Block通过ReceiverTrackeEndpointRef发送给Driver，如下所示：

*def pushAndReportBlock(*

*receivedBlock: ReceivedBlock,*

*metadataOption: Option[Any],*

*blockIdOption: Option[StreamBlockId]*

*) {*

*val blockId = blockIdOption.getOrElse(nextBlockId)*

*val time = System.currentTimeMillis*

*val blockStoreResult = receivedBlockHandler.storeBlock(blockId, receivedBlock)*

*val numRecords = blockStoreResult.numRecords*

*val blockInfo = ReceivedBlockInfo(streamId, numRecords, metadataOption, blockStoreResult)*

*trackerEndpoint.askSync[Boolean](AddBlock(blockInfo))*

*}*

在这里不再详述。

Spark Stream：http://blog.csdn.net/javastart/article/details/50965696

http://blog.csdn.net/u011564172/article/details/60143168

http://shiyanjun.cn/archives/1536.html

http://blog.csdn.net/u011564172/article/details/60875013

https://github.com/neoremind/kraps-rpc

http://blog.csdn.net/zhongqi2513/article/details/77745450

https://github.com/ColZer/DigAndBuried/tree/master/spark

http://www.raincent.com/content-85-9645-2.html

http://www.ppvke.com/Blog/archives/42322

http://shiyanjun.cn/archives/1536.html

http://blog.csdn.net/u011564172/article/details/59113617

http://netty.io/wiki/user-guide-for-4.x.html