## 5.3 RPC环境

RpcEnv是Spark 2.x.x版本中新出现的组件，那么它是不是给Spark带来了新的能力呢？对于有些组件是这样的，但是RpcEnv组件肩负着另一项历史使命——替代Spark 2.x.x以前版本中采用的Akka。Spark移除了对Akka的使用，这令我感到困惑，“Akka是一款很不错的分布式消息系统，为什么要替代？正如3.2节所说的——可以让用户使用任何版本的Akka来编程”。Akka具有分布式集群下的消息发送、远端同步调用、远端异步调用、路由、持久化、监管、At Least Once Delivery（至少投递一次）等能力，一旦Akka被替代，那就意味着RpcEnv必须也能支持这些机制。

现在我们来看看SparkEnv中的RpcEnv是如何被创建的，SparkEnv中创建RpcEnv的代码如下。

val systemName = if (isDriver) driverSystemName else executorSystemName  
val rpcEnv = RpcEnv.create(systemName, bindAddress, advertiseAddress, port, conf,  
 securityManager, clientMode = !isDriver)

这段代码首先生成系统名称systemName，如果当前应用为Driver（即SparkEnv位于Driver中），那么systemName为sparkDriver，否则（即SparkEnv位于Executor中），systemName为sparkExecutor。然后调用RpcEnv的create方法创建RpcEnv。create方法中只有两条语句。

val config = RpcEnvConfig(conf, name, bindAddress, advertiseAddress, port, securityManager,  
 clientMode) // RpcEnvConfig中保存了RpcEnv的配置信息  
 new NettyRpcEnvFactory().create(config) // 创建RpcEnv

这里的RpcEnvConfig实际是一个样例类，用于保存RpcEnv的配置信息。实际创建RpcEnv的动作发生在NettyRpcEnvFactory的create方法中，如代码清单5-1所示。

代码清单5-1 创建NettyRpcEnv

def create(config: RpcEnvConfig): RpcEnv = {  
 val sparkConf = config.conf  
 val javaSerializerInstance =  
 new JavaSerializer(sparkConf).newInstance().asInstanceOf[JavaSerializerInstance]  
 val nettyEnv =  
 new NettyRpcEnv(sparkConf, javaSerializerInstance, config.advertiseAddress,  
 config.securityManager)  
 if (!config.clientMode) {  
 val startNettyRpcEnv: Int => (NettyRpcEnv, Int) = { actualPort =>  
 nettyEnv.startServer(config.bindAddress, actualPort)  
 (nettyEnv, nettyEnv.address.port)  
 }  
 try {  
 Utils.startServiceOnPort(config.port, startNettyRpcEnv, sparkConf, config.name).\_1  
 } catch {  
 case NonFatal(e) =>  
 nettyEnv.shutdown()  
 throw e  
 }  
 }  
 nettyEnv  
}  
}

代码清单5-1中创建NettyRpcEnv的步骤如下。

1）创建javaSerializerInstance。此实例将用于RPC传输对象的序列化。

2）创建NettyRpcEnv。创建NettyRpcEnv其实就是对内部各个子组件TransportConf、Dispatcher、TransportContext、TransportClientFactory、TransportServer的实例化过程，这些内容都将在后面一一介绍。

3）启动NettyRpcEnv。这一步首先定义了一个偏函数startNettyRpcEnv，其函数实际为执行NettyRpcEnv的startServer方法（startServer方法的具体内容将在5.3.7节进行介绍），最后在启动NettyRpcEnv之后返回NettyRpcEnv及服务最终使用的端口。这里使用了Utils的startServiceOnPort方法，startServiceOnPort实际上是调用了作为参数的偏函数startNettyRpcEnv，有关startServiceOnPort的具体介绍可以参阅附录A中的内容。

由于抽象类RpcEnv只有一个实现子类NettyRpcEnv，所以本章不打算展示RpcEnv的接口定义，而是直接从介绍NettyRpcEnv入手。在正式介绍NettyRpcEnv的构造过程之前，我们需要先做一些准备。RpcEndpoint和RpcEndpointRef都是NettyRpcEnv中的重要概念，我们需要首先理解它们，再来看NettyRpcEnv的构造过程。

# 读累了记得休息一会哦~

公众号：古德猫宁李

* 电子书搜索下载
* 书单分享
* 书友学习交流

网站：[沉金书屋 https://www.chenjin5.com](https://www.chenjin5.com)

* 电子书搜索下载
* 电子书打包资源分享
* 学习资源分享

### 5.3.1 RPC端点RpcEndpoint

RPC端点是对Spark的RPC通信实体的统一抽象，所有运行于RPC框架之上的实体都应该继承RpcEndpoint。Spark早期版本节点间的消息通信主要采用Akka的Actor，从Spark 2.0.0版本开始移除了对Akka的依赖，这就意味着Spark需要Actor的替代品，RPC端点RpcEndpoint由此而生。RpcEndpoint是对能够处理RPC请求，给某一特定服务提供本地调用及跨节点调用的RPC组件的抽象。

1.RPC端点RpcEndpoint的定义

特质RpcEndpoint的定义如代码清单5-2所示。

代码清单5-2 RPC端点定义

private[spark] trait RpcEndpoint {  
 val rpcEnv: RpcEnv  
 final def self: RpcEndpointRef = {  
 require(rpcEnv != null, "rpcEnv has not been initialized")  
 rpcEnv.endpointRef(this)  
 }  
 def receive: PartialFunction[Any, Unit] = {  
 case \_ => throw new SparkException(self + " does not implement 'receive'")  
 }  
 def receiveAndReply(context: RpcCallContext): PartialFunction[Any, Unit] = {  
 case \_ => context.sendFailure(new SparkException(self + " won't reply anything"))  
 }  
 def onError(cause: Throwable): Unit = {  
 throw cause  
 }  
 def onConnected(remoteAddress: RpcAddress): Unit = {  
 }  
 def onDisconnected(remoteAddress: RpcAddress): Unit = {  
 }  
 def onNetworkError(cause: Throwable, remoteAddress: RpcAddress): Unit = {  
 }  
 def onStart(): Unit = {  
 }  
 def onStop(): Unit = {  
 }  
 final def stop(): Unit = {  
 val \_self = self  
 if (\_self != null) {  
 rpcEnv.stop(\_self)  
 }  
 }  
}

这里对代码清单5-2中所展示的RpcEndpoint的各个成员进行介绍。

·rpcEnv：当前RpcEndpoint所属的RpcEnv。

·self：获取RpcEndpoint相关联的RpcEndpointRef。从代码实现看到，其实现实际调用了RpcEnv的endpointRef方法。由于RpcEnv并未实现此方法，所以需要RpcEnv的子类来实现。以NettyRpcEnv为例，其endpointRef方法的实现如代码清单5-40所示。

·receive：接收消息并处理，但不需要给客户端回复。

·receiveAndReply：接收消息并处理，需要给客户端回复。回复是通过RpcCall-Context来实现的。

·onError：当处理消息发生异常时调用，可以对异常进行一些处理。

·onConnected：当客户端与当前节点连接上之后调用，可以针对连接进行一些处理。

·onDisconnected：当客户端与当前节点的连接断开之后调用，可以针对断开连接进行一些处理。

·onNetworkError：当客户端与当前节点之间的连接发生网络错误时调用，可以针对连接发生的网络错误进行一些处理。

·onStart：在RpcEndpoint开始处理消息之前调用，可以在RpcEndpoint正式工作之前做一些准备工作。

·onStop：在停止RpcEndpoint时调用，可以在RpcEndpoint停止的时候做一些收尾工作。

·stop：用于停止当前RpcEndpoint。从代码实现看到，其实现实际调用了RpcEnv的stop方法。由于RpcEnv并未实现此方法，所以需要其子类来实现。以NettyRpcEnv为例，其stop方法的实现如5.3.9节中代码清单5-45所示。

对Akka的Actor编程模型熟悉的读者，一定觉得RpcEndpoint中的一些接口非常类似于Actor，笔者认为Spark 2.0.0版本在去掉了Akka的同时，自身实现RpcEnv时肯定参考了Actor的一些思想。

2.特质RpcEndpoint的继承体系

由于RpcEndpoint只是一个特质，除了对接口的定义，并没有任何实现逻辑，所以我们需要看看有哪些子类实现了RpcEndpoint。RpcEndpoint的继承体系如图5-2所示。

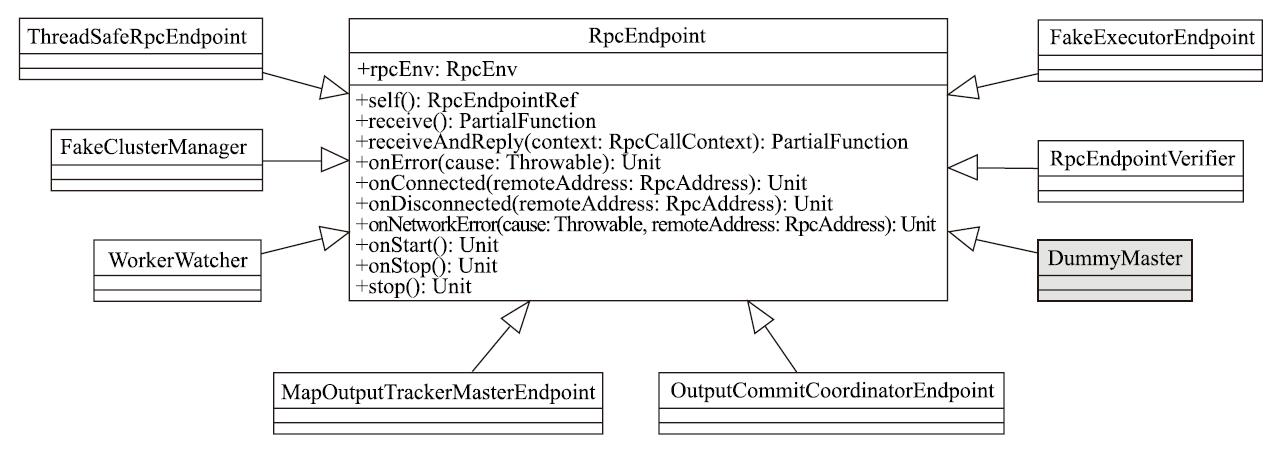


图5-2 RpcEndpoint的继承体系

图5-2中灰色的子类型DummyMaster（Dummy意为虚拟的、假的或傀儡的）正如其名字一样，不是NettyRpcEnv中具有真正用途的RpcEndpoint，而只用于测试。图中显示的RpcEndpoint的子类，除了ThreadSafeRpcEndpoint之外，都将在具体遇到时再作详细分析。ThreadSafeRpcEndpoint是继承自RpcEndpoint的特质，主要用于对消息的处理，必须是线程安全的场景。ThreadSafeRpcEndpoint对消息的处理都是串行的，即前一条消息处理完才能接着处理下一条消息。ThreadSafeRpcEndpoint的继承体系如图5-3所示。

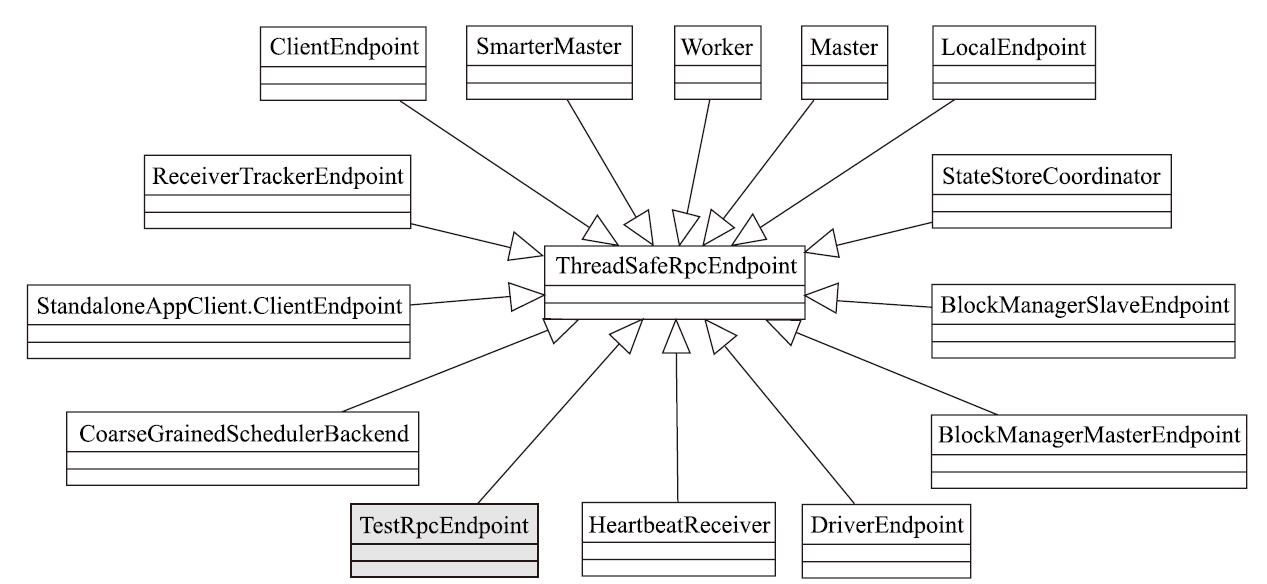


图5-3 ThreadSafeRpcEndpoint的继承体系

图5-3中的TestRpcEndpoint用于测试，其余实现类都在NettyRpcEnv中发挥着各自的作用，这些RpcEndpoint将在本书后续的各个章节中需要时再作具体介绍。

### 5.3.2 RPC端点引用RpcEndpointRef

如果说RpcEndpoint是Akka中Actor的替代产物，那么RpcEndpointRef就是Actor-Ref的替代产物。在Akka中只要你持有了一个Actor的引用ActorRef，那么你就可以使用此ActorRef向远端的Actor发起请求。RpcEndpointRef也具有同等的效用，要向一个远端的RpcEndpoint发起请求，你就必须持有这个RpcEndpoint的RpcEndpointRef。RpcEndpoint与RpcEndpointRef之间的关系如图5-4所示。

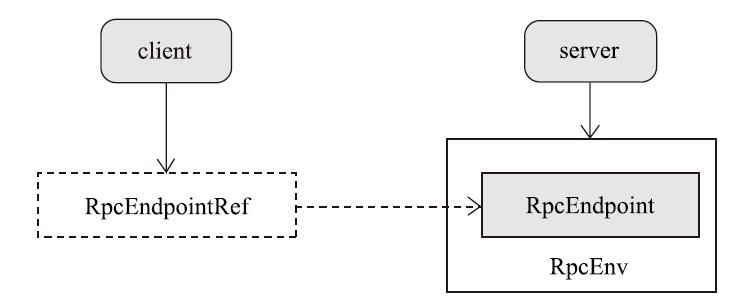


图5-4 RpcEndpoint与RpcEndpointRef的关系

在正式介绍RpcEndpointRef之前，先要理解什么是消息投递规则。

1.消息投递规则

一般而言，消息投递有下面3种情况。

·at-most-once： 意味着每条应用了这种机制的消息会被投递0次或1次。可以说这条消息可能会丢失。

·at-least-once： 意味着每条应用了这种机制的消息潜在地存在多次投递尝试并保证至少会成功一次。就是说这条消息可能会重复但是不会丢失。

·exactly-once： 意味着每条应用了这种机制的消息只会向接收者准确地发送一次。换言之，这种消息既不会丢失，也不会重复。

at-most-once的成本最低且性能最高，因为它在发送完消息后不会尝试去记录任何状态，然后这条消息将被抛之脑后。at-least-once需要发送者必须认识它所发送过的消息，并对没有收到回复的消息进行发送重试。这就要求接收者对消息的处理必须是幂等的，否则可能会带来新的问题。exactly-once的成本是三者中最高，性能却又是三者中最差的。它除了要求发送者有记忆和重试能力，还要求接收者能够认识接收过的消息并能过滤出那些重复的消息投递。

2.RPC端点引用RpcEndpointRef的定义

有了对消息投递规则的理解，我们现在开始介绍RpcEndpointRef。抽象类RpcEndpoint-Ref定义了所有RpcEndpoint引用的属性与接口，如代码清单5-3所示。

代码清单5-3 RpcEndpointRef的接口定义

private[spark] abstract class RpcEndpointRef(conf: SparkConf)  
 extends Serializable with Logging {  
 private[this] val maxRetries = RpcUtils.numRetries(conf)  
 private[this] val retryWaitMs = RpcUtils.retryWaitMs(conf)  
 private[this] val defaultAskTimeout = RpcUtils.askRpcTimeout(conf)  
 def address: RpcAddress  
 def name: String  
 def send(message: Any): Unit  
 def ask[T: ClassTag](message: Any, timeout: RpcTimeout): Future[T]  
 def ask[T: ClassTag](message: Any): Future[T] = ask(message, defaultAskTimeout)  
 def askWithRetry[T: ClassTag](message: Any): T = askWithRetry(message, defaultAskTimeout)  
 def askWithRetry[T: ClassTag](message: Any, timeout: RpcTimeout): T = {  
 var attempts = 0  
 var lastException: Exception = null  
 while (attempts < maxRetries) {  
 attempts += 1  
 try {  
 val future = ask[T](message, timeout)  
 val result = timeout.awaitResult(future)  
 if (result == null) {  
 throw new SparkException("RpcEndpoint returned null")  
 }  
 return result  
 } catch {  
 case ie: InterruptedException => throw ie  
 case e: Exception =>  
 lastException = e  
 logWarning(s"Error sending message [message = $message] in $attempts attempts", e)  
 }  
  
 if (attempts < maxRetries) {  
 Thread.sleep(retryWaitMs)  
 }  
 }  
  
 throw new SparkException(  
 s"Error sending message [message = $message]", lastException)  
 }  
  
}

从代码清单5-3中看到，抽象类RpcEndpointRef中有3个属性，分别如下。

·maxRetries：RPC最大重新连接次数。可以使用spark.rpc.numRetries属性进行配置，默认为3次。

·retryWaitMs：RPC每次重新连接需要等待的毫秒数。可以使用spark.rpc.retry.wait属性进行配置，默认值为3秒。

·defaultAskTimeout：RPC的ask操作的默认超时时间。可以使用spark.rpc.askTimeout或者spark.network.timeout属性进行配置，默认值为120秒。spark.rpc.askTimeout属性的优先级更高。

这三个属性都是通过调用工具类RpcUtils的方法得到的，想了解RpcUtils的介绍，可以参阅附录H。

接着来看一看RpcEndpointRef中各个方法的功能。

·address：返回当前RpcEndpointRef对应RpcEndpoint的RPC地址（RpcAddress）。

·name：返回当前RpcEndpointRef对应RpcEndpoint的名称。

·send：发送单向异步的消息。所谓“单向”就是发送完后就会忘记此次发送，不会有任何状态要记录，也不会期望得到服务端的回复。send采用了at-most-once的投递规则。RpcEndpointRef的send方法非常类似于Akka中Actor的tell方法。

·ask[T:ClassTag](message:Any)：以默认的超时时间作为timeout参数，调用ask[T:ClassTag](message:Any,timeout:RpcTimeout)方法。

·askWithRetry[T:ClassTag](message:Any,timeout:RpcTimeout)：发送同步的请求，此类请求将会被RpcEndpoint接收，并在指定的超时时间内等待返回类型为T的处理结果。当此方法抛出SparkException时，将会进行请求重试，直到超过了默认的重试次数为止。由于此类方法会重试，因此要求服务端对消息的处理是幂等的。此方法也采用了at-least-once的投递规则。此方法也非常类似于Akka中采用了at-least-once机制的Actor的ask方法。

·askWithRetry[T:ClassTag](message:Any)：以默认的超时时间作为timeout参数，调用askWithRetry[T:ClassTag](message:Any,timeout:RpcTimeout)。

小贴士： Akka的Actor模型一般提供的消息都属于at-most-once，那是因为大多数场景不需要有状态的消息投递，如Web服务器请求。当你有强一致性需求时，才应该启用Akka的at-least-once机制。

### 5.3.3 创建传输上下文TransportConf

根据3.2.1节的介绍，我们知道TransportConf是RPC框架中的配置类。由于RPC环境RpcEnv的底层也依赖于数据总线，因此需要创建传输上下文TransportConf。创建TransportConf是构造NettyRpcEnv的过程中的第一步，代码如下。

private[netty] val transportConf = SparkTransportConf.fromSparkConf(  
 conf.clone.set("spark.rpc.io.numConnectionsPerPeer", "1"),  
 "rpc",  
 conf.getInt("spark.rpc.io.threads", 0))

这里用到了3.2.1节介绍的SparkTransportConf，可以看到在调用SparkTransportConf的fromSparkConf方法之前，首先对SparkConf进行了克隆，然后设置了spark.rpc.io.numConnectionsPerPeer，此属性的格式在3.2.2节分析TransportClientFactory的成员变量时介绍过，由于当前模块名为rpc，所以为spark.rpc.io.numConnectionsPerPeer。我们还看到这里通过spark.rpc.io.threads属性来设置Netty传输线程数。

### 5.3.4 消息调度器Dispatcher

创建消息调度器Dispatcher是有效提高NettyRpcEnv对消息异步处理并最大提升并行处理能力的前提。Dispatcher负责将RPC消息路由到要该对此消息处理的RpcEndpoint（RPC端点）。创建Dispatcher的代码如下。

private val dispatcher: Dispatcher = new Dispatcher(this)

1.消息调度器Dispatcher的概述

想要把Dispatcher解释得足够明白，最好的方式是先弄明白其中涉及的概念。Dispatcher中的概念包括以下几个。

·RpcEndpoint：RPC端点，即RPC分布式环境中一个具体的实例，其可以对指定的消息进行处理。由于RpcEndpoint是一个特质，所以需要提供RpcEndpoint的实现类。特质RpcEndpoint已在前文详细介绍，此处不再赘述。

·RpcEndpointRef：RPC端点引用，即RPC分布式环境中一个具体实体的引用，所谓引用实际是“spark://host:port/name”这种格式的地址。其中，host为端点所在RPC服务所在的主机IP，port是端点所在RPC服务的端口，name是端点实例的名称。抽象类RpcEndpointRef已在前文详细介绍，此处不再赘述。

·InboxMessage：Inbox盒子内的消息。InboxMessage是一个特质，所有类型的RPC消息都继承自InboxMessage。InboxMessage的继承体系可以用图5-5表示。

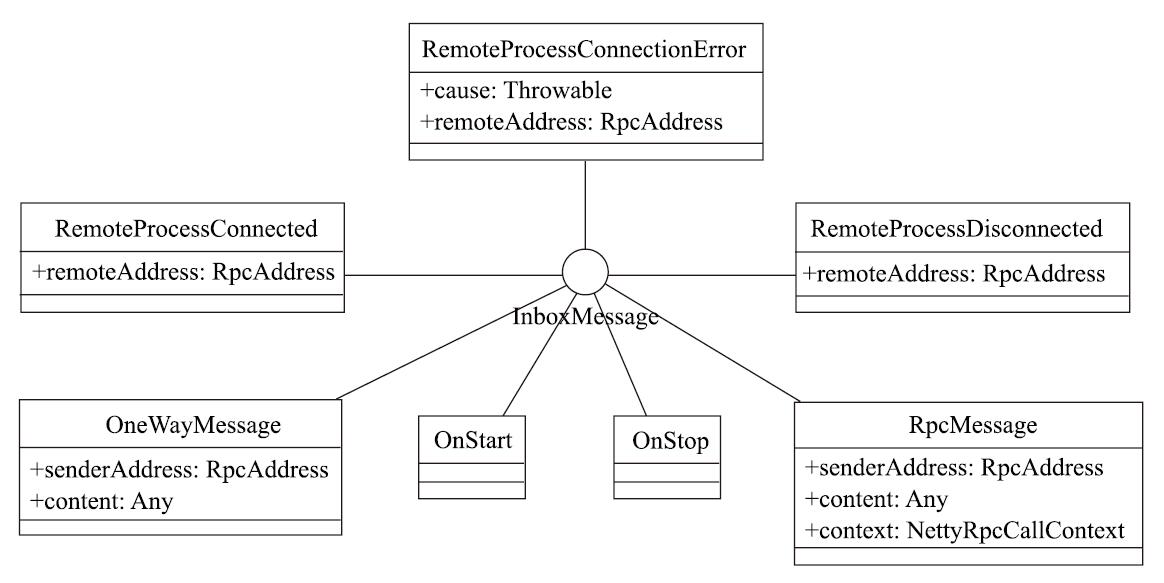


图5-5 InboxMessage的继承体系

图5-5展示了InboxMessage的多种实现类，其作用分别如下。

·OneWayMessage：RpcEndpoint处理此类型的消息后不需要向客户端回复信息。

·RpcMessage：RPC消息，RpcEndpoint处理完此消息后需要向客户端回复信息。

·OnStart：用于Inbox实例化后，再通知与此Inbox相关联的RpcEndpoint启动。

·OnStop：用于Inbox停止后，通知与此Inbox相关联的RpcEndpoint停止。

·RemoteProcessConnected：此消息用于告诉所有的RpcEndpoint，有远端的进程已经与当前RPC服务建立了连接。

·RemoteProcessDisconnected：此消息用于告诉所有的RpcEndpoint，有远端的进程已经与当前RPC服务断开了连接。

·RemoteProcessConnectionError：此消息用于告诉所有的RpcEndpoint，与远端某个地址之间的连接发生了错误。

·Inbox：端点内的盒子。每个RpcEndpoint都有一个对应的盒子，这个盒子里有个存储InboxMessage消息的列表messages。所有的消息将缓存在messages列表里面，并由RpcEndpoint异步处理这些消息。

·EndpointData：RPC端点数据，它包括了RpcEndpoint、NettyRpcEndpointRef及Inbox等属于同一个端点的实例。Inbox与RpcEndpoint、NettyRpcEndpointRef通过此EndpointData相关联。

有了对以上概念的介绍，现在来看看Dispatcher中的一些成员变量。

·endpoints：端点实例名称与端点数据EndpointData之间映射关系的缓存。有了这个缓存，就可以使用端点名称从中快速获取或删除EndpointData了。

·endpointRefs：端点实例RpcEndpoint与端点实例引用RpcEndpointRef之间映射关系的缓存。有了这个缓存，就可以使用端点实例从中快速获取或删除端点实例引用了。

·receivers：存储端点数据EndpointData的阻塞队列。只有Inbox中有消息的EndpointData才会被放入此阻塞队列。

·stopped：Dispatcher是否停止的状态。

·threadpool：用于对消息进行调度的线程池。此线程池运行的任务都是MessageLoop（将在代码清单5-4、代码清单5-5中详细介绍）。

有了对Dispatcher涉及概念及属性的了解，我们就可以更容易理解Dispatcher的内存模型了，如图5-6所示。

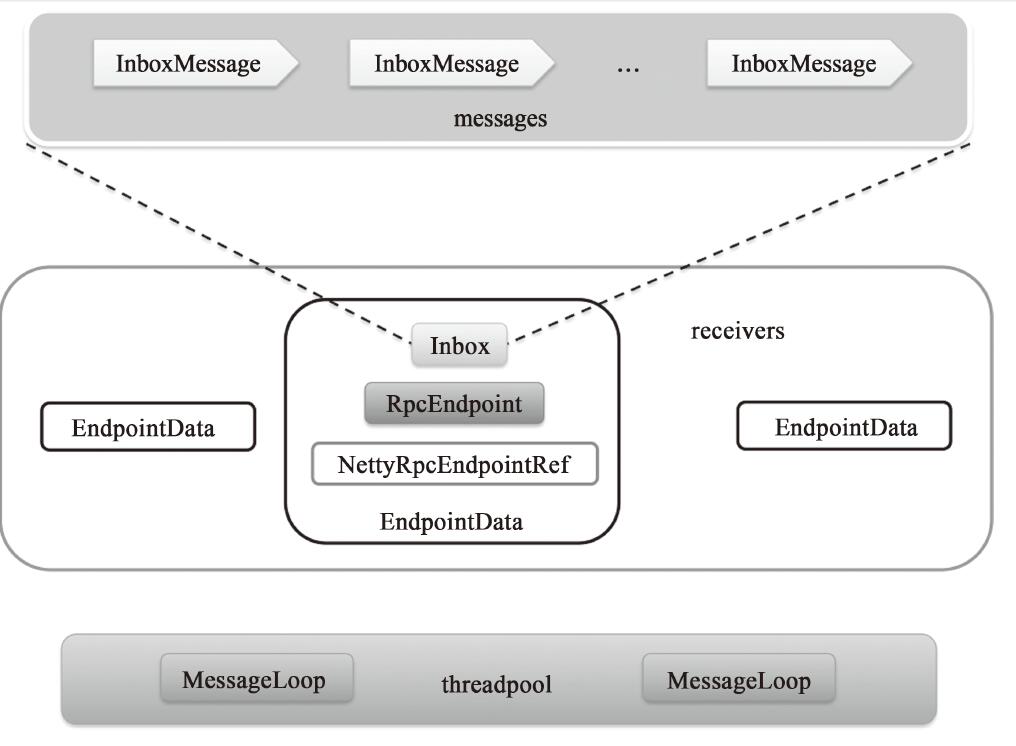


图5-6 Dispatcher的内存模型

2.Dispatcher的调度原理

在创建Dispatcher的最后会创建对消息进行调度的线程池threadpool，如代码清单5-4所示。

代码清单5-4 创建Dispatcher中的线程池

private val threadpool: ThreadPoolExecutor = {  
 val numThreads = nettyEnv.conf.getInt("spark.rpc.netty.dispatcher.numThreads",  
 math.max(2, Runtime.getRuntime.availableProcessors()))  
 val pool = ThreadUtils.newDaemonFixedThreadPool(numThreads, "dispatcher-event-loop")  
 for (i <- 0 until numThreads) {  
 pool.execute(new MessageLoop)  
 }  
 pool  
}

根据代码清单5-4，可以看到创建threadpool线程池的步骤如下。

1）获取此线程池的大小numThreads。此线程池的大小默认为2与当前系统可用处理器数量之间的最大值，也可以使用spark.rpc.netty.dispatcher.numThreads属性配置。

2）创建线程池。此线程池是固定大小的线程池，并且启动的线程都以后台线程方式运行，且线程名以dispatcher-event-loop为前缀。

3）启动多个运行MessageLoop任务的线程，这些线程的数量与threadpool线程池的大小相同。

4）返回此线程池的引用。

MessageLoop实现了Java的Runnable接口，其实现如代码清单5-5所示。

代码清单5-5 MessageLoop的实现

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)  
 } catch {  
 case NonFatal(e) => logError(e.getMessage, e)  
 }  
 }  
 } catch {  
 case ie: InterruptedException => // exit  
 }  
 }  
}

根据代码清单5-5，我们看到MessageLoop果如其名，在循环过程中不断对新的消息进行处理。每次循环中的逻辑如下。

1）从receivers中获取EndpointData。receivers中的EndpointData，其Inbox的messages列表中肯定有了新的消息。换言之，只有Inbox的messages列表中有了新的消息，此EndpointData才会被放入receivers中。由于receivers是个阻塞队列，所以当receivers中没有EndpointData时，MessageLoop线程会被阻塞。

2）如果取到的EndpointData是“毒药”（PoisonPill），那么此MessageLoop线程将退出（通过return语句）。这里有个动作就是将PoisonPill重新放入到receivers中，这是因为threadpool线程池极有可能不止一个MessageLoop线程，为了让大家都“毒发身亡”，还需要把“毒药”放回到receivers中，这样其他“活着”的线程就会再次误食“毒药”，达到所有MessageLoop线程都结束的效果。

3）如果取到的EndpointData不是“毒药”，那么调用EndpointData中Inbox的process方法对消息进行处理。

上文的MessageLoop任务实际是将消息交给EndpointData中Inbox的process方法处理的，但在正式介绍process之前，先来看看Inbox中的一些成员属性，这样有助于之后对process方法的理解。Inbox中的成员属性如下。

·messages：消息列表。用于缓存需要由对应RpcEndpoint处理的消息，即与Inbox在同一EndpointData中的RpcEndpoint。Inbox构造的最后是一些有些古怪的代码，如代码清单5-6所示。

代码清单5-6 Inbox向自身的messages列表中放入OnStart消息

inbox.synchronized {  
 messages.add(OnStart)  
}

上述代码将向Inbox自身的messages列表中放入OnStart消息，为什么会有这么一段代码？此处先不解释这段代码的具体用途。

·stopped：Inbox的停止状态。

·enableConcurrent：是否允许多个线程同时处理messages中的消息。

·numActiveThreads：激活线程的数量，即正在处理messages中消息的线程数量。

介绍完Inbox中的成员属性，我们一起来看看其process方法的实现，如代码清单5-7所示。

代码清单5-7 Inbox中的消息处理

def process(dispatcher: Dispatcher): Unit = {  
 var message: InboxMessage = null  
 inbox.synchronized {  
 if (!enableConcurrent && numActiveThreads != 0) {  
 return  
 }  
 message = messages.poll()  
 if (message != null) {  
 numActiveThreads += 1  
 } else {  
 return  
 }  
 }  
 while (true) {  
 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}")  
 })  
 } catch {  
 case NonFatal(e) =>  
 context.sendFailure(e)  
 // Throw the exception -- this exception will be caught by the safelyCall function.  
 // The endpoint's onError function will be called.  
 throw e  
 }  
  
 case OneWayMessage(\_sender, content) =>  
 endpoint.receive.applyOrElse[Any, Unit](content, { msg =>  
 throw new SparkException(s"Unsupported message $message from ${\_sender}")  
 })  
  
 case OnStart =>  
 endpoint.onStart()  
 if (!endpoint.isInstanceOf[ThreadSafeRpcEndpoint]) {  
 inbox.synchronized {  
 if (!stopped) {  
 enableConcurrent = true  
 }  
 }  
 }  
  
 case OnStop =>  
 val activeThreads = inbox.synchronized { inbox.numActiveThreads }  
 assert(activeThreads == 1,  
 s"There should be only a single active thread but found $activeThreads threads.")  
 dispatcher.removeRpcEndpointRef(endpoint)  
 endpoint.onStop()  
 assert(isEmpty, "OnStop should be the last message")  
  
 case RemoteProcessConnected(remoteAddress) =>  
 endpoint.onConnected(remoteAddress)  
  
 case RemoteProcessDisconnected(remoteAddress) =>  
 endpoint.onDisconnected(remoteAddress)  
  
 case RemoteProcessConnectionError(cause, remoteAddress) =>  
 endpoint.onNetworkError(cause, remoteAddress)  
 }  
 }  
  
 inbox.synchronized {  
 // "enableConcurrent" will be set to false after 'onStop' is called, so we should check it  
 // every time.  
 if (!enableConcurrent && numActiveThreads != 1) {  
 // If we are not the only one worker, exit  
 numActiveThreads -= 1  
 return  
 }  
 message = messages.poll()  
 if (message == null) {  
 numActiveThreads -= 1  
 return  
 }  
 }  
 }  
}

根据代码清单5-7，Inbox对消息处理的步骤如下。

1）进行线程并发检查。具体是，如果不允许多个线程同时处理messages中的消息（enableConcurrent为false），并且当前激活线程数（numActiveThreads）不为0，这说明已经有线程在处理消息，所以当前线程不允许再去处理消息（使用return返回）。

2）从messages中获取消息。如果有消息未处理，则当前线程需要处理此消息，因而算是一个新的激活线程（需要将numActiveThreads加1）。如果messages中没有消息了（一般发生在多线程情况下），则直接返回。

3）根据消息类型进行匹配，并执行对应的逻辑。这里有个小技巧值得借鉴，那就是匹配执行的过程中也许会发生错误，当发生错误的时候，我们希望当前Inbox所对应RpcEndpoint的错误处理方法onError可以接收到这些错误信息。Inbox的safelyCall方法给我们提供了这方面的实现，如代码清单5-8所示。

代码清单5-8 safelyCall的实现

private def safelyCall(endpoint: RpcEndpoint)(action: => Unit): Unit = {  
 try action catch {  
 case NonFatal(e) =>  
 try endpoint.onError(e) catch {  
 case NonFatal(ee) => logError(s"Ignoring error", ee)  
 }  
 }  
}

4）对激活线程数量进行控制。当第3）步对消息处理完毕后，当前线程作为之前已经激活的线程是否还有存在的必要呢？这里有两个判断：如果不允许多个线程同时处理messages中的消息并且当前激活的线程数多于1个，那么需要当前线程退出并将numActiveThreads减1；如果messages已经没有消息要处理了，这说明当前线程无论如何也该返回并将numActiveThreads减1。

Image00062.jpg 注意 第1）步、第2）步及第4）步位于Inbox的锁保护之下，是因为messages是普通的java.util.LinkedList，LinkedList本身不是线程安全的，所以为了增加并发安全性，需要通过同步保护。

3.Inbox的消息来源

MessageLoop线程的执行逻辑是不断地消费各个EndpointData中Inbox里的消息，但是EndpointData是何时放入receivers中的？Inbox里的消息来自哪里？Dispatcher中有很多完成这些功能的方法，让我们一起看看Dispatcher中与此相关的一些方法。

（1）注册RpcEndpoint

Dispatcher的registerRpcEndpoint方法用于注册RpcEndpoint，这个方法的副作用便可以将EndpointData放入receivers，其实现如代码清单5-9所示。

代码清单5-9 注册RpcEndpoint

def registerRpcEndpoint(name: String, endpoint: RpcEndpoint): NettyRpcEndpointRef = {  
 val addr = RpcEndpointAddress(nettyEnv.address, name)  
 val endpointRef = new NettyRpcEndpointRef(nettyEnv.conf, addr, nettyEnv)  
 synchronized {  
 if (stopped) {  
 throw new IllegalStateException("RpcEnv has been stopped")  
 }  
 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)  
 endpointRefs.put(data.endpoint, data.ref)  
 receivers.offer(data) // for the OnStart message  
 }  
 endpointRef  
}

从代码清单5-9可以看到，注册RpcEndpoint的步骤如下。

1）使用当前RpcEndpoint所在NettyRpcEnv的地址和RpcEndpoint的名称创建RpcEndpointAddress对象。

2）创建RpcEndpoint的引用对象——NettyRpcEndpointRef。

3）创建EndpointData，并放入endpoints缓存。

4）将RpcEndpoint与NettyRpcEndpointRef的映射关系放入endpointRefs缓存。

5）将EndpointData放入阻塞队列receivers的队尾。MessageLoop线程异步获取到此EndpointData，并处理其Inbox中刚刚放入的OnStart消息，最终调用RpcEndpoint的OnStart方法在RpcEndpoint开始处理消息之前做一些准备工作。

小贴士： 由于EndpointData是新创建的，所以在构造EndpointData的过程中也会构造它内部的Inbox。因此Inbox在构造的最后会执行代码清单5-6所示的逻辑，向Inbox自身的messages列表放入OnStart消息。随后在注册RpcEndpoint的第5）步，当前Inbox所属的EndpointData就被放入receivers中。最后MessageLoop线程就会取出此EndpointData处理，进而调用当前Inbox的process方法处理OnStart。Inbox的process方法处理OnStart的效果是在Inbox自身初始化完毕后，再启动与此Inbox相关联的RpcEndpoint（可以参见代码清单5-7中对OnStart消息匹配后的执行过程）。

6）返回NettyRpcEndpointRef。

（2）对RpcEndpoint去注册

Dispatcher的stop方法（见代码清单5-10）用于对RpcEndpoint的注册。

代码清单5-10 Dispatcher的stop方法

def stop(rpcEndpointRef: RpcEndpointRef): Unit = {  
 synchronized {  
 if (stopped) {  
 return  
 }  
 unregisterRpcEndpoint(rpcEndpointRef.name)  
 }  
}

读者可能对stop这个方法的命名有些怀疑，不应该是停止Dispatcher的吗？阅读了代码清单5-10中stop方法的实现后，你就会释然。代码清单5-10中stop方法的执行步骤为：首先判断Dispatcher是否已经停止，如果Dispatcher未停止，则调用Dispatcher的unregisterRpcEndpoint方法对RpcEndpoint去注册。

Dispatcher的unregisterRpcEndpoint方法用于对RpcEndpoint去注册，这个方法的副作用也可以将EndpointData放入receivers，其实现如代码清单5-11所示。

代码清单5-11 对RpcEndpoint去注册

private def unregisterRpcEndpoint(name: String): Unit = {  
 val data = endpoints.remove(name)  
 if (data != null) {  
 data.inbox.stop()  
 receivers.offer(data) // for the OnStop message  
 }  
}

从代码清单5-11可以看到，去注册的步骤如下。

1）从endpoints中移除EndpointData。

2）调用EndpointData中Inbox的stop方法停止Inbox。

3）将EndpointData重新放入receivers中。

代码本身很简单，可是奇怪的是为什么EndpointData从endpoints中移除后，最后还要放入receivers？EndpointData虽然移除了，但是对应的RpcEndpointRef并没有从endpointRefs缓存中移除，这又是为什么？这里先不直接回答这两个问题。

当要移除一个EndpointData时，其Inbox可能正在对消息进行处理，所以不能贸然停止。这里采用了更平滑的停止方式，即调用了Inbox的stop方法来平滑过渡，stop方法的实现如代码清单5-12所示。

代码清单5-12 Inbox的stop方法

def stop(): Unit = inbox.synchronized {  
 if (!stopped) {  
 enableConcurrent = false  
 stopped = true  
 messages.add(OnStop)  
 }  
}

Inbox的stop方法的处理步骤如下。

1）根据之前的分析，MessageLoop有“允许并发运行”和“不允许并发运行”两种情况。对于允许并发的情况，为了确保安全，应该将enableConcurrent设置为false。

2）设置当前Inbox为停止状态。

3）向messages中添加OnStop消息。根据代码清单5-5中对MessageLoop的分析，为了能够处理OnStop消息，只有Inbox所属的EndpointData放入receivers中，其messages列表中的消息才会被处理，这回答了刚才提出的第一个问题。为了实现平滑停止，OnStop消息最终将匹配代码清单5-7，调用Dispatcher的removeRpcEndpointRef方法，将RpcEndpoint与RpcEndpointRef的映射从缓存endpointRefs中移除，这回答了刚才的第二个问题。在匹配执行OnStop消息的最后，将调用RpcEndpoint的onStop方法对RpcEndpoint停止。

（3）将消息提交给指定的RpcEndpoint

Dispatcher的postMessage用于将消息提交给指定的RpcEndpoint，其实现如代码清单5-13所示。

代码清单5-13 postMessage的实现

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  
 }  
 }  
 // We don't need to call 'onStop' in the 'synchronized' block  
 error.foreach(callbackIfStopped)  
}

从代码清单5-13可以看出，postMessage方法的处理步骤如下。

1）根据端点名称endpointName从缓存endpoints中获取EndpointData。

2）如果当前Dispatcher没有停止并且缓存endpoints中确实存在名为endpointName的EndpointData，那么将调用EndpointData对应Inbox的post方法将消息加入Inbox的消息列表中，因此还需要将EndpointData推入receivers，以便MessageLoop处理此Inbox中的消息。Inbox的post方法的实现如代码清单5-14所示，其逻辑为Inbox未停止时向messages列表加入消息。

代码清单5-14 Inbox的post方法

def post(message: InboxMessage): Unit = inbox.synchronized {  
 if (stopped) {  
 // We already put "OnStop" into "messages", so we should drop further messages  
 onDrop(message)  
 } else {  
 messages.add(message)  
 false  
 }  
}

此外，Dispatcher中还有一些方法间接使用了Dispatcher的postMessage方法，如代码清单5-15所示。

代码清单5-15 对postMessage方法的间接使用

def postToAll(message: InboxMessage): Unit = {  
 val iter = endpoints.keySet().iterator()  
 while (iter.hasNext) {  
 val name = iter.next  
 postMessage(name, message, (e) => logWarning(s"Message $message dropped. ${e.getMessage}"))  
 }  
}  
  
/\*\* Posts a message sent by a remote endpoint. \*/  
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))  
}  
  
/\*\* Posts a message sent by a local endpoint. \*/  
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))  
}  
  
/\*\* Posts a one-way message. \*/  
def postOneWayMessage(message: RequestMessage): Unit = {  
 postMessage(message.receiver.name, OneWayMessage(message.senderAddress, message. content),  
 (e) => throw e)  
}

代码清单5-15中的方法除了postOneWayMessage方法，其他需要回复的消息中都封装了RpcCallContext。RpcCallContext是用于回调客户端的上下文，其定义如代码清单5-16所示。

代码清单5-16 RpcCallContext的定义

private[spark] trait RpcCallContext {  
 def reply(response: Any): Unit  
 def sendFailure(e: Throwable): Unit  
 def senderAddress: RpcAddress  
}

代码清单5-16中的RpcCallContext一共定义了三个接口。

·reply：用于向发送者回复信息。

·sendFailure：用于向发送者发送失败信息。

·senderAddress：用于获取发送者的地址。

特质RpcCallContext的继承体系如图5-7所示。

（4）停止Dispatcher

Dispatcher的stop方法用来停止Dispatcher，其实现如代码清单5-17所示。

代码清单5-17 停止Dispatcher

def stop(): Unit = {  
 synchronized {  
 if (stopped) {  
 return  
 }  
 stopped = true  
 }  
 // Stop all endpoints. This will queue all endpoints for processing by the message loops.  
 endpoints.keySet().asScala.foreach(unregisterRpcEndpoint)  
 // Enqueue a message that tells the message loops to stop.  
 receivers.offer(PoisonPill)  
 threadpool.shutdown()  
}

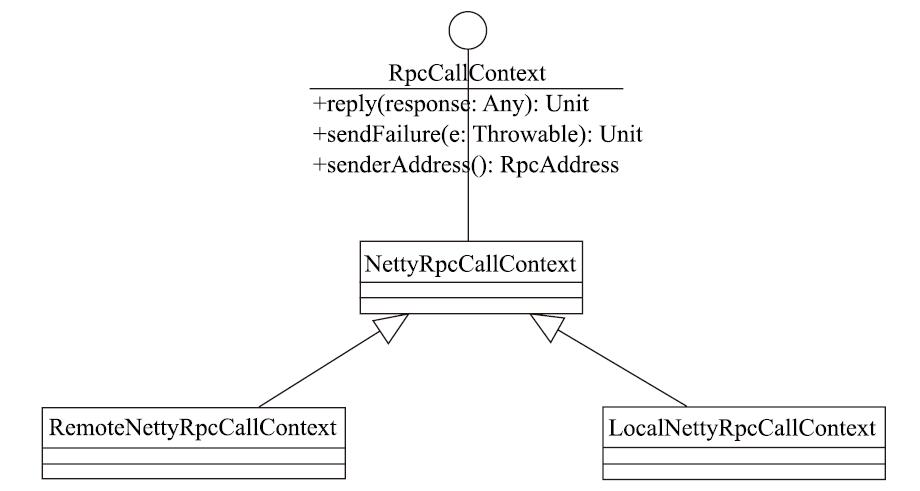


图5-7 RpcCallContext的继承体系

代码清单5-17中stop方法的执行步骤如下。

1）如果Dispatcher还未停止，则将自身状态修改为已停止。

2）对endpoints中的所有EndpointData去注册。这里通过调用unregisterRpcEndpoint方法（见代码清单5-11），将向endpoints中的每个EndpointData的Inbox里放置OnStop消息。

3）向receivers中投放“毒药”。

Image00045.jpg 提示 由于Dispatcher都停止了，所以Dispatcher中的所有MessageLoop线程也就没有存在的必要了。

4）关闭threadpool线程池。

经过对Dispatcher和MessageLoop的分析，我们现在可以对图5-6中展现的内容进行扩展，增加一些运行时的执行流程，如图5-8所示。

图5-8中的序号代表其中的简要步骤，具体解释如下。

·序号①表示调用Inbox的post方法将消息放入messages列表中。

·序号②表示将有消息的Inbox相关联的EndpointData放入receivers。

·序号③表示MessageLoop每次循环首先从receivers中获取EndpointData。

·序号④表示执行EndpointData中Inbox的process方法对消息进行具体处理。

### 5.3.5 创建传输上下文TransportContext

创建传输上下文TransportContext是NettyRpcEnv提供服务端与客户端能力的前提。创建TransportContext的代码如下。

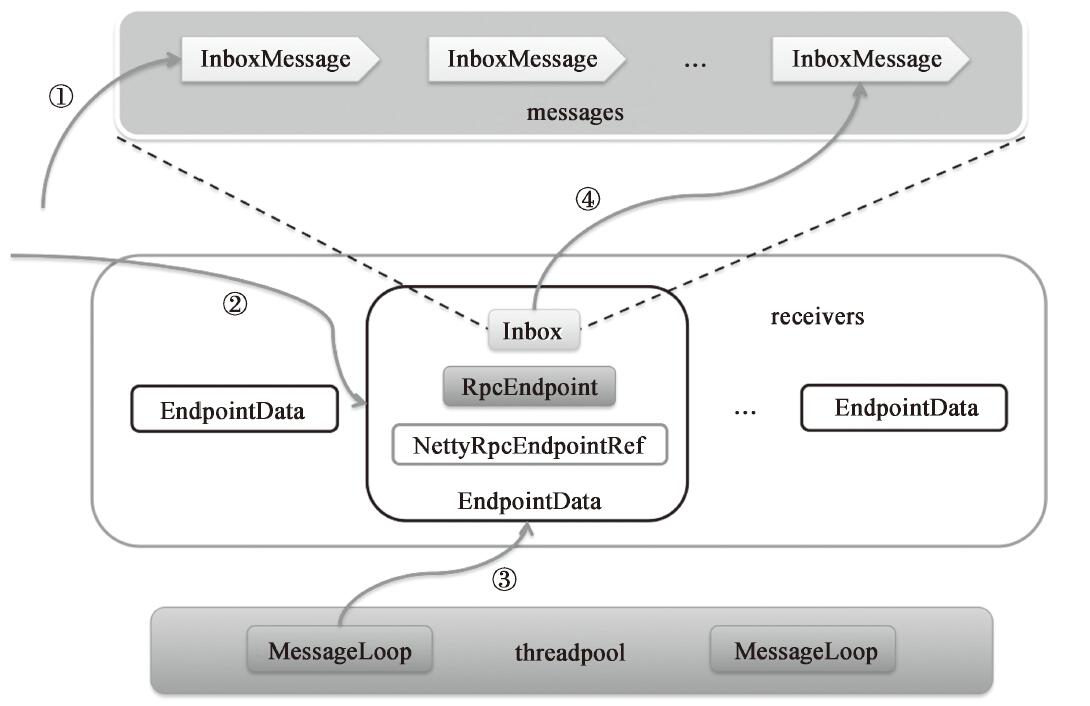


图5-8 Dispatcher中MessageLoop线程的执行流程

private val streamManager = new NettyStreamManager(this)  
private val transportContext = new TransportContext(transportConf,  
 new NettyRpcHandler(dispatcher, this, streamManager))

关于TransportContext，在3.2节已经有足够多的介绍，所以这里重点来看看其构造器中传入的RpcHandler参数。这里用于构造TransportContext的RpcHandler实际是其实现类NettyRpcHandler，NettyRpcHandler的构造器里则以刚刚新建的NettyStreamManager实例作为参数。本节将首先介绍NettyStreamManager，然后来看看NettyRpcHandler的实现。

1.NettyStreamManager详解

NettyStreamManager实现了StreamManager，专用于为NettyRpcEnv提供文件服务的能力。创建StreamManager的代码如下。

private val streamManager = new NettyStreamManager(this)

从上面代码可以看出，此处创建的streamManager的实际类型为NettyStreamManager。NettyStreamManager用于提供NettyRpcEnv的文件流服务，这替代了在Spark 1.x.x版本中使用Jetty实现的HttpFileServer。在3.2.6节的代码清单3-27中我们曾经介绍过，RpcHandler的getStreamManager方法返回的类型为StreamManager。NettyStreamManager正是继承了StreamManager。NettyStreamManager中定义了三个文件与目录缓存，如代码清单5-18所示。

代码清单5-18 NettyStreamManager中的文件与目录缓存

private val files = new ConcurrentHashMap[String, File]()  
private val jars = new ConcurrentHashMap[String, File]()  
private val dirs = new ConcurrentHashMap[String, File]()

根据这三个缓存结构，我们猜想NettyStreamManager应该可以提供对普通文件、Jar文件及目录的下载支持。NettyStreamManager中一共提供了两类方法，一类用于添加缓存，一类用于支持文件流的读取。添加缓存的三个方法（如代码清单5-19所示）。

代码清单5-19 添加缓存的方法

override def addFile(file: File): String = {  
 val existingPath = files.putIfAbsent(file.getName, file)  
 require(existingPath == null || existingPath == file,  
 s"File ${file.getName} was already registered with a different path " +  
 s"(old path = $existingPath, new path = $file")  
 s"${rpcEnv.address.toSparkURL}/files/${Utils.encodeFileNameToURIRawPath(file.getName())}"  
}  
  
override def addJar(file: File): String = {  
 val existingPath = jars.putIfAbsent(file.getName, file)  
 require(existingPath == null || existingPath == file,  
 s"File ${file.getName} was already registered with a different path " +  
 s"(old path = $existingPath, new path = $file")  
 s"${rpcEnv.address.toSparkURL}/jars/${Utils.encodeFileNameToURIRawPath(file.getName())}"  
}  
  
override def addDirectory(baseUri: String, path: File): String = {  
 val fixedBaseUri = validateDirectoryUri(baseUri)  
 require(dirs.putIfAbsent(fixedBaseUri.stripPrefix("/"), path) == null,  
 s"URI '$fixedBaseUri' already registered.")  
 s"${rpcEnv.address.toSparkURL}$fixedBaseUri"  
}

代码清单4-22中提到的fileServer实际就是NettyStreamManager。由于添加缓存的三个方法都比较简单，所以不再过多介绍。

NettyStreamManager提供的用于打开文件流的方法如代码清单5-20所示。

代码清单5-20 打开文件流

override def openStream(streamId: String): ManagedBuffer = {  
 val Array(ftype, fname) = streamId.stripPrefix("/").split("/", 2)  
 val file = ftype match {  
 case "files" => files.get(fname)  
 case "jars" => jars.get(fname)  
 case other =>  
 val dir = dirs.get(ftype)  
 require(dir != null, s"Invalid stream URI: $ftype not found.")  
 new File(dir, fname)  
 }  
 if (file != null && file.isFile()) {  
 new FileSegmentManagedBuffer(rpcEnv.transportConf, file, 0, file.length())  
 } else {  
 null  
 }  
}

openStream方法的逻辑很简单，从缓存中获取文件后，将TransportConf及File等信息封装为FileSegmentManagedBuffer并返回。我们之前在代码清单3-24、代码清单3-25及代码清单3-26中介绍过FileSegmentManagedBuffer的实现，我们知道FileSegmentManagedBuffer封装了对文件的流处理细节。由于NettyStreamManager只实现了Stream Manager的openStream方法，根据3.2.6节对TransportRequestHandler的handle方法和process StreamRequest方法的介绍，我们知道NettyStreamManager将只提供对StreamRequest类型消息的处理。各个Executor节点就可以使用Driver节点的RpcEnv提供的NettyStreamManager，从Driver将Jar包或文件下载到Executor节点上供任务执行。

2.NettyRpcHandler详解

上文谈到NettyRpcEnv中用于构造TransportContext的RpcHandler实际是其实现类NettyRpcHandler。结合代码清单3-27中对RpcHandler的定义，我们选择两个重载的receive方法来看看NettyRpcHandler是如何实现RpcHandler的。

·对客户端进行响应的receive方法（见代码清单5-21）的分析。

代码清单5-21 NettyRpcHandler的receive方法

override def receive(  
 client: TransportClient,  
 message: ByteBuffer,  
 callback: RpcResponseCallback): Unit = {  
 val messageToDispatch = internalReceive(client, message)  
 dispatcher.postRemoteMessage(messageToDispatch, callback)  
}

可以看到，代码清单5-21中receive方法的处理步骤如下。

1）调用internalReceive方法将ByteBuffer类型的message转换为RequestMessage。

2）调用Dispatcher的postRemoteMessage方法（见代码清单5-15）将消息转换为RpcMessage后放入Inbox的消息列表。根据代码清单5-7，MessageLoop将调用RpcEnd-Point实现类的receiveAndReply方法，即RpcEndPoint处理完消息后会向客户端进行回复。

internalReceive方法可以将ByteBuffer类型的message转换为RequestMessage，其实现如代码清单5-22所示。

代码清单5-22 internalReceive的实现

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 = nettyEnv.deserialize[RequestMessage](client, message)  
 if (requestMessage.senderAddress == null) {  
 RequestMessage(clientAddr, requestMessage.receiver, requestMessage.content)  
 } else {  
 val remoteEnvAddress = requestMessage.senderAddress  
 if (remoteAddresses.putIfAbsent(clientAddr, remoteEnvAddress) == null) {  
 dispatcher.postToAll(RemoteProcessConnected(remoteEnvAddress))  
 }  
 requestMessage  
 }  
}

根据代码清单5-22，internalReceive的执行步骤如下。

1）从TransportClient中获取远端地址RpcAddress。

2）调用NettyRpcEnv的deserialize方法对客户端发送的序列化后的消息（即ByteBuffer类型的消息）进行反序列化，根据deserialize的实现（见代码清单5-23），反序列化实际使用了javaSerializerInstance。javaSerializerInstance是通过NettyRpcEnv的构造参数传入的对象，类型为JavaSerializerInstance（请参阅代码清单5-1）。

代码清单5-23 反序列化ByteBuffer类型的消息

private[netty] def deserialize[T: ClassTag](client: TransportClient, bytes: ByteBuffer): T = {  
 NettyRpcEnv.currentClient.withValue(client) {  
 deserialize { () =>  
 javaSerializerInstance.deserialize[T](bytes)  
 }  
 }  
}

3）如果反序列化得到的请求消息requestMessage中没有发送者的地址信息，则使用从TransportClient中获取的远端地址RpcAddress、requestMessage的接收者（即RpcEndpoint）、requestMessage的内容，以构造新的RequestMessage。

4）如果反序列化得到的请求消息requestMessage中含有发送者的地址信息，则将从TransportClient中获取的远端地址RpcAddress与requestMessage中的发送者地址信息之间的映射关系放入缓存remoteAddresses。还将调用Dispatcher的postToAll方法（见代码清单5-15）向endpoints缓存的所有EndpointData的Inbox中放入RemoteProcessConnected消息 [[1]](#_1_35) 。最后将返回requestMessage。

·对客户端进行响应的receive重载方法（其实现见代码清单5-24）的分析。

代码清单5-24 NettyRpcHandler的receive重载方法

override def receive(  
 client: TransportClient,  
 message: ByteBuffer): Unit = {  
 val messageToDispatch = internalReceive(client, message)  
 dispatcher.postOneWayMessage(messageToDispatch)  
}

根据3.2.6节对RpcHandler实现的分析，我们知道此方法不会对客户端进行回复。此方法也调用了internalReceive方法，但是最后向EndpointData的Inbox投递消息使用了postOneWayMessage方法（见代码清单5-15），将消息转换为OneWayMessage后放入Inbox的消息列表。根据代码清单5-7，将调用RpcEndPoint实现类的receive方法，也就是说RpcEndPoint处理完消息后不会向客户端进行回复。

由于NettyRpcHandler实现了RpcHandler的两个receive方法，所以根据3.2.6节Transport-RequestHandler的handle方法、processRpcRequest方法及processOneWayMessage方法的介绍，我们知道NettyRpcHandler将提供对OneWayMessage和RpcRequest两种类型消息的处理。

NettyRpcHandler除实现了RpcHandler的两个receive方法，还实现了exception-Caught、channelActive与channelInactive等。exceptionCaught方法将会向Inbox中投递Remote ProcessConnectionError消息。channelActive将会向Inbox中投递RemoteProcess-Connected。channelInactive将会向Inbox中投递RemoteProcessDisconnected消息。这几个方法的处理都与receive方法类似，感兴趣的读者可以自行分析，此处不再赘述。

[[1]](#_1_34) 根据代码清单5-7中Inbox的process方法对RemoteProcessConnected消息的匹配， 将执行所有RpcEndpoint的onConnected方法。特质RpcEndpoint中的onConnected是个空方法，所以RpcEndpoint的具体实现类可以针对远端客户端连接到服务端设计一些定制化的功能。

### 5.3.6 创建传输客户端工厂TransportClientFactory

创建传输客户端工厂TransportClientFactory是NettyRpcEnv向远端服务发起请求的基础，Spark与远端RpcEnv进行通信都依赖于TransportClientFactory生产的TransportClient。NettyRpcEnv中共创建了两个TransportClientFactory，代码如下。

private def createClientBootstraps(): java.util.List[TransportClientBootstrap] = {  
 if (securityManager.isAuthenticationEnabled()) {  
 java.util.Arrays.asList(new SaslClientBootstrap(transportConf, "", securityManager,  
 securityManager.isSaslEncryptionEnabled()))  
 } else {  
 java.util.Collections.emptyList[TransportClientBootstrap]  
 }  
}  
private val clientFactory = transportContext.createClientFactory(createClientBootstraps())  
@volatile private var fileDownloadFactory: TransportClientFactory = \_

这里的clientFactory用于常规的发送请求和接收响应。fileDownloadFactory则用于文件下载。由于有些RpcEnv本身并不需要从远端下载文件，所以这里只声明了变量fileDownload-Factory，并未进一步对其初始化。需要下载文件的RpcEnv会调用downloadClient方法创建TransportClientFactory，并用此TransportClientFactory创建下载所需的传输客户端TransportClient。downloadClient的创建见代码清单5-25。

代码清单5-25 获取下载的TransportClient

private def downloadClient(host: String, port: Int): TransportClient = {  
 if (fileDownloadFactory == null) synchronized {  
 if (fileDownloadFactory == null) {  
 val module = "files"  
 val prefix = "spark.rpc.io."  
 val clone = conf.clone()  
  
 // Copy any RPC configuration that is not overridden in the spark.files namespace.  
 conf.getAll.foreach { case (key, value) =>  
 if (key.startsWith(prefix)) {  
 val opt = key.substring(prefix.length())  
 clone.setIfMissing(s"spark.$module.io.$opt", value)  
 }  
 }  
  
 val ioThreads = clone.getInt("spark.files.io.threads", 1)  
 val downloadConf = SparkTransportConf.fromSparkConf(clone, module, ioThreads)  
 val downloadContext = new TransportContext(downloadConf, new NoOpRpcHandler(), true)  
 fileDownloadFactory = downloadContext.createClientFactory(createClientBootstraps())  
 }  
 }  
 fileDownloadFactory.createClient(host, port)  
}

根据代码清单5-25，我们知道fileDownloadFactory与clientFactory使用的Spark-TransportConf内部代理的SparkConf都是从NettyRpcEnv的SparkConf克隆来的，不同之处在于clientFactory所属的模块（module变量）为rpc，fileDownloadFactory所属的模块为files。clientFactory中的读写线程数由spark.rpc.io.numConnectionsPerPeer属性控制，而fileDownloadFactory中的读写线程数由spark.files.io.threads属性控制。有关SparkTransportConf.fromSparkConf方法在3.2.1节已经介绍过，TransportContext的createClientFactory方法的内容也在3.2.2节详细说明，故此处不再赘述。

### 5.3.7 创建TransportServer

作为一个RPC环境，NettyRpcEnv不应该只具有向远端服务发起请求并接收响应的能力，也应当对外提供接收请求、处理请求、回复客户端的服务。NettyRpcEnv中创建TransportServer的代码如下 [[1]](#_1_37) 。

@volatile private var server: TransportServer = \_  
private val stopped = new AtomicBoolean(false)

可以看到，TransportServer在此时并未实例化，那么它是何时实例化的？在代码清单5-1中我们介绍过启动RpcEnv的偏函数startNettyRpcEnv，startNettyRpcEnv将负责回调NettyRpcEnv的startServer方法，startServer的实现如代码清单5-26所示。

代码清单5-26 启动TransportServer

def startServer(bindAddress: String, port: Int): Unit = {  
 val bootstraps: java.util.List[TransportServerBootstrap] =  
 if (securityManager.isAuthenticationEnabled()) {  
 java.util.Arrays.asList(new SaslServerBootstrap(transportConf, securityManager))  
 } else {  
 java.util.Collections.emptyList()  
 }  
 server = transportContext.createServer(bindAddress, port, bootstraps)  
 dispatcher.registerRpcEndpoint(  
 RpcEndpointVerifier.NAME, new RpcEndpointVerifier(this, dispatcher))  
}

根据代码清单5-26，启动NettyRpcEnv的步骤如下。

1）创建TransportServer。这里使用了TransportContext的createServer方法，此方法已经在3.2.3节详细介绍，不再赘述。

2）向Dispatcher注册RpcEndpointVerifier。RpcEndpointVerifier用于校验指定名称的RpcEndpoint是否存在。RpcEndpointVerifier在Dispatcher中的注册名为endpoint-verifier，其实现如代码清单5-27所示。registerRpcEndpoint在代码清单5-9有详细分析，读者可以回顾。

代码清单5-27 RpcEndpointVerifier的实现

private[netty] class RpcEndpointVerifier(override val rpcEnv: RpcEnv, dispatcher: Dispatcher)  
 extends RpcEndpoint {  
  
 override def receiveAndReply(context: RpcCallContext): PartialFunction[Any, Unit] = {  
 case RpcEndpointVerifier.CheckExistence(name) => context.reply(dispatcher.verify(name))  
 }  
}  
  
private[netty] object RpcEndpointVerifier {  
 val NAME = "endpoint-verifier"  
 case class CheckExistence(name: String)  
}

可以看到RpcEndpointVerifier实现了RpcEndpoint的receiveAndReply方法。因此MessageLoop线程在处理RpcEndpointVerifier所关联的Inbox中的消息时，会匹配Rpc-Message调用RpcEndpointVerifier的receiveAndReply方法。RpcEndpointVerifier实现的receiveAndReply方法的处理步骤如下。

1）接收CheckExistence类型的消息，匹配出name参数，此参数代表要查询的Rpc-Endpoint的具体名称。

2）调用Dispatcher的verify方法。verify用于校验Dispatcher的endpoints缓存中是否存在名为name的RpcEndpoint，其实现如代码清单5-28所示。

3）调用RpcCallContext的reply方法回复客户端。RpcCallContext的定义如代码清单5-16所示。

代码清单5-28 Dispatcher的verify方法

def verify(name: String): Boolean = {  
 endpoints.containsKey(name)  
}

根据对RpcEndpointVerifier的实现分析，我们知道它对外提供了查询当前RpcEnd-pointVerifier所在RpcEnv的Dispatcher中是否存在请求中指定名称所对应的Rpc-Endpoint。

TransportServer初始化并且启动后，就可以利用NettyRpcHandler和NettyStreamManager对外提供服务了。

[[1]](#_1_36) 这里顺带列出了NettyRpcEnv的状态属性。

### 5.3.8 客户端请求发送

之前已经介绍了NettyRpcHandler和NettyStreamManager提供的服务端实现，现在我们来看看如何向远端RpcEndpoint发送消息。

当TransportClient发出请求之后，会等待获取服务端的回复，这就涉及超时问题。另外由于TransportClientFactory.createClient方法是阻塞式调用，所以需要一个异步的处理。NettyRpcEnv中实现这些需求的代码如下。

val timeoutScheduler = ThreadUtils.newDaemonSingleThreadScheduledExecutor("netty-rpc-env-timeout")  
private[netty] val clientConnectionExecutor = ThreadUtils.newDaemonCached-ThreadPool(  
 "netty-rpc-connection",  
conf.getInt("spark.rpc.connect.threads", 64))  
private val outboxes = new ConcurrentHashMap[RpcAddress, Outbox]()

上面代码中创建了与发送请求相关的三个组件，分别如下。

·timeoutScheduler：用于处理请求超时的调度器。timeoutScheduler的类型实际是ScheduledExecutorService，比起使用Timer组件，ScheduledExecutorService将比Timer更加稳定，比如线程挂掉后，ScheduledExecutorService会重启一个新的线程定时检查请求是否超时。

·clientConnectionExecutor：一个用于异步处理TransportClientFactory.createClient方法调用的线程池。这个线程池的大小默认为64，可以使用spark.rpc.connect.threads属性进行配置。

·outboxes：RpcAddress与Outbox的映射关系的缓存。每次向远端发送请求时，此请求消息首先放入此远端地址对应的Outbox，然后使用线程异步发送。

1.Outbox与OutboxMessage

上文提到的outboxes缓存了远端RPC地址与Outbox的关系，那么Outbox是什么？为了说明Outbox，不妨来看看Outbox中的成员属性。

·nettyEnv：当前Outbox所在节点上的NettyRpcEnv。

·address：Outbox所对应的远端NettyRpcEnv的地址。

·messages：向其他远端NettyRpcEnv上的所有RpcEndpoint发送的消息列表。

·client：当前Outbox内的TransportClient。消息的发送都依赖于此传输客户端。

·connectFuture：指向当前Outbox内连接任务的java.util.concurrent.Future引用。如果当前没有连接任务，则connectFuture为null。

·stopped：当前Outbox是否停止的状态。

·draining：表示当前Outbox内正有线程在处理messages列表中消息的状态。

消息列表messages中的消息类型为OutboxMessage，所有将要向远端发送的消息都会被封装成OutboxMessage类型。OutboxMessage作为一个特质，定义了所有向外发送消息的规范，其定义如代码清单5-29所示。

代码清单5-29 OutboxMessage的定义

private[netty] sealed trait OutboxMessage {  
 def sendWith(client: TransportClient): Unit  
 def onFailure(e: Throwable): Unit  
}

根据OutboxMessage的名称，我们很容易与Dispatcher中Inbox里的InboxMessage类型的消息关联起来。OutboxMessage在客户端使用，是对外发送消息的封装。InboxMessage在服务端使用，是对所接收消息的封装。OutboxMessage的继承体系如图5-9所示。

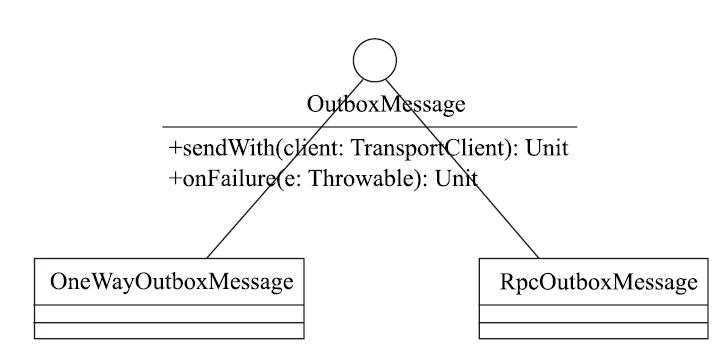


图5-9 OutboxMessage的继承体系

我们以RpcOutboxMessage为例，来看看OutboxMessage该如何实现，RpcOutbox-Message的实现如代码清单5-30所示。

代码清单5-30 RpcOutboxMessage的实现

private[netty] case class RpcOutboxMessage(  
 content: ByteBuffer,  
 \_onFailure: (Throwable) => Unit,  
 \_onSuccess: (TransportClient, ByteBuffer) => Unit)  
extends OutboxMessage with RpcResponseCallback with Logging {  
  
 private var client: TransportClient = \_  
 private var requestId: Long = \_  
  
 override def sendWith(client: TransportClient): Unit = {  
 this.client = client  
 this.requestId = client.sendRpc(content, this)  
 }  
  
 def onTimeout(): Unit = {  
 if (client != null) {  
 client.removeRpcRequest(requestId)  
 } else {  
 logError("Ask timeout before connecting successfully")  
 }  
 }  
  
 override def onFailure(e: Throwable): Unit = {  
 \_onFailure(e)  
 }  
  
 override def onSuccess(response: ByteBuffer): Unit = {  
 \_onSuccess(client, response)  
 }  
  
}

根据代码清单5-30，我们知道RpcOutboxMessage重写的sendWith方法正是利用了TransportClient的sendRpc方法（见3.2.8节）。TransportClient的sendRpc方法的第二个参数是RpcResponseCallback类型，RpcOutboxMessage本身也实现了RpcResponseCallback，所以调用的时候传递了RpcOutboxMessage的this引用。

有了对Outbox中各个属性的介绍，现在一起来认识下Outbox的各个方法。Outbox的发送消息的方法是最常用的方法，其实现如代码清单5-31所示。

代码清单5-31 Outbox的发送消息的方法

def send(message: OutboxMessage): Unit = {  
 val dropped = synchronized {  
 if (stopped) {  
 true  
 } else {  
 messages.add(message)  
 false  
 }  
 }  
 if (dropped) {  
 message.onFailure(new SparkException("Message is dropped because Outbox is stopped"))  
 } else {  
 drainOutbox()  
 }  
}

根据代码清单5-31，Outbox的send方法的执行步骤如下。

1）判断当前Outbox的状态是否已经停止。

2）如果Outbox已经停止，则向发送者发送SparkException异常。如果Outbox还未停止，则将OutboxMessage添加到messages列表中，并且调用drainOutbox方法处理messages中的消息。drainOutbox是一个私有方法，其实现如代码清单5-32所示。

代码清单5-32 drainOutbox的实现

private def drainOutbox(): Unit = {  
 var message: OutboxMessage = null  
 synchronized {  
 if (stopped) {  
 return  
 }  
 if (connectFuture != null) {  
 return  
 }  
 if (client == null) {  
 launchConnectTask()  
 return  
 }  
 if (draining) {  
 return  
 }  
 message = messages.poll()  
 if (message == null) {  
 return  
 }  
 draining = true  
 }  
 while (true) {  
 try {  
 val \_client = synchronized { client }  
 if (\_client != null) {  
 message.sendWith(\_client)  
 } else {  
 assert(stopped == true)  
 }  
 } catch {  
 case NonFatal(e) =>  
 handleNetworkFailure(e)  
 return  
 }  
 synchronized {  
 if (stopped) {  
 return  
 }  
 message = messages.poll()  
 if (message == null) {  
 draining = false  
 return  
 }  
 }  
 }  
}

根据代码清单5-32，drainOutbox的执行步骤如下。

1）如果当前Outbox已经停止或者正在连接远端服务，则返回。

2）如果Outbox中的TransportClient为null，这说明还未连接远端服务。此时需要调用launchConnectTask方法运行连接远端服务的任务，然后返回。

3）如果正有线程在处理（即发送）messages列表中的消息，则返回。

4）如果messages列表中没有消息要处理，则返回。否则取出其中的一条消息，并将draining状态置为true。

5）循环处理messages列表中的消息，即不断从messages列表中取出消息并调用OutboxMessage的sendWith方法发送消息。

在drainOutbox方法中调用launchConnectTask方法，运行连接远端服务的任务，其实现如代码清单5-33所示。

代码清单5-33 运行连接远端服务的任务

private def launchConnectTask(): Unit = {  
 connectFuture = nettyEnv.clientConnectionExecutor.submit(new Callable[Unit] {  
  
 override def call(): Unit = {  
 try {  
 val \_client = nettyEnv.createClient(address)  
 outbox.synchronized {  
 client = \_client  
 if (stopped) {  
 closeClient()  
 }  
 }  
 } catch {  
 case ie: InterruptedException =>  
 // exit  
 return  
 case NonFatal(e) =>  
 outbox.synchronized { connectFuture = null }  
 handleNetworkFailure(e)  
 return  
 }  
 outbox.synchronized { connectFuture = null }  
 // It's possible that no thread is draining now. If we don't drain here, we cannot send the  
 // messages until the next message arrives.  
 drainOutbox()  
 }  
 })  
}

根据代码清单5-33，launchConnectTask方法的执行步骤如下。

1）构造Callable的匿名内部类，此匿名类将调用NettyRpcEnv的createClient方法创建TransportClient，然后调用drainOutbox方法处理Outbox中的消息。

2）使用NettyRpcEnv中的clientConnectionExecutor提交Callable的匿名内部类。

NettyRpcEnv的createClient方法的实现如代码清单5-34所示。

代码清单5-34 创建TransportClient

private[netty] def createClient(address: RpcAddress): TransportClient = {  
 clientFactory.createClient(address.host, address.port)  
}

从代码清单5-34中看到，NettyRpcEnv的createClient方法实际使用了TransportClient-Factory的createClient方法（见3.2.2节）。

2.NettyRpcEndpointRef详解

在NettyRpcEnv中，要向远端RpcEndpoint发送请求，首先要持有RpcEndpoint的引用对象NettyRpcEndpointRef（类似于Akka中Actor的ActorRef）。NettyRpcEndpointRef是RpcEndpointRef的唯一子类。NettyRpcEndpointRef重写了RpcEndpointRef的部分方法，其实现如代码清单5-35所示。

代码清单5-35 NettyRpcEndpointRef的实现

private[netty] class NettyRpcEndpointRef(  
 @transient private val conf: SparkConf,  
 endpointAddress: RpcEndpointAddress,  
 @transient @volatile private var nettyEnv: NettyRpcEnv)  
 extends RpcEndpointRef(conf) with Serializable with Logging {  
  
 @transient @volatile var client: TransportClient = \_  
  
 private val \_address = if (endpointAddress.rpcAddress != null) endpointAddress else null  
 private val \_name = endpointAddress.name  
  
 override def address: RpcAddress = if (\_address != null) \_address.rpcAddress else null  
  
 private def readObject(in: ObjectInputStream): Unit = {  
 in.defaultReadObject()  
 nettyEnv = NettyRpcEnv.currentEnv.value  
 client = NettyRpcEnv.currentClient.value  
 }  
  
 private def writeObject(out: ObjectOutputStream): Unit = {  
 out.defaultWriteObject()  
 }  
  
 override def name: String = \_name  
  
 override def ask[T: ClassTag](message: Any, timeout: RpcTimeout): Future[T] = {  
 nettyEnv.ask(RequestMessage(nettyEnv.address, this, message), timeout)  
 }  
  
 override def send(message: Any): Unit = {  
 require(message != null, "Message is null")  
 nettyEnv.send(RequestMessage(nettyEnv.address, this, message))  
 }  
  
 override def toString: String = s"NettyRpcEndpointRef(${\_address})"  
  
 def toURI: URI = new URI(\_address.toString)  
  
 final override def equals(that: Any): Boolean = that match {  
 case other: NettyRpcEndpointRef => \_address == other.\_address  
 case \_ => false  
 }  
  
 final override def hashCode(): Int = if (\_address == null) 0 else \_address.hashCode()  
　}

根据代码清单5-35，NettyRpcEndpointRef包含了以下属性。

·client：类型为TransportClient（TransportClient的具体内容请参阅3.2.8节）。Netty-RpcEndpointRef将利用此TransportClient向远端的RpcEndpoint发送请求。

·\_address：远端RpcEndpoint的地址RpcEndpointAddress。

·\_name：远端RpcEndpoint的名称。

·下面来介绍NettyRpcEndpointRef中重写的方法。

·address：返回\_address属性的值，或返回null。

·name：返回\_name属性的值。

·ask[T:ClassTag](message:Any,timeout:RpcTimeout)：首先将message封装为Request Message，然后调用NettyRpcEnv的ask方法。

·send(message:Any)：首先将message封装为RequestMessage，然后调用NettyRpcEnv的send方法。

NettyRpcEndpointRef的ask方法和send方法分别调用了NettyRpcEnv的ask方法和send方法，下面将对它们进行详细介绍。

（1）询问

NettyRpcEnv重写了抽象类RpcEnv的ask方法，其实现如代码清单5-36所示。

代码清单5-36 询问方法ask的实现

private[netty] def ask[T: ClassTag](message: RequestMessage, timeout: RpcTimeout): Future[T] = {  
 val promise = Promise[Any]()  
 val remoteAddr = message.receiver.address  
  
 def onFailure(e: Throwable): Unit = {  
 if (!promise.tryFailure(e)) {  
 logWarning(s"Ignored failure: $e")  
 }  
 }  
  
 def onSuccess(reply: Any): Unit = reply match {  
 case RpcFailure(e) => onFailure(e)  
 case rpcReply =>  
 if (!promise.trySuccess(rpcReply)) {  
 logWarning(s"Ignored message: $reply")  
 }  
 }  
  
 try {  
 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)  
 } else {  
 val rpcMessage = RpcOutboxMessage(serialize(message),  
 onFailure,  
 (client, response) => onSuccess(deserialize[Any](client, response)))  
 postToOutbox(message.receiver, rpcMessage)  
 promise.future.onFailure {  
 case \_: TimeoutException => rpcMessage.onTimeout()  
 case \_ =>  
 }(ThreadUtils.sameThread)  
 }  
  
 val timeoutCancelable = timeoutScheduler.schedule(new Runnable {  
 override def run(): Unit = {  
 onFailure(new TimeoutException(s"Cannot receive any reply in ${timeout.duration}"))  
 }  
 }, timeout.duration.toNanos, TimeUnit.NANOSECONDS)  
 promise.future.onComplete { v =>  
 timeoutCancelable.cancel(true)  
 }(ThreadUtils.sameThread)  
 } catch {  
 case NonFatal(e) =>  
 onFailure(e)  
 }  
 promise.future.mapTo[T].recover(timeout.addMessageIfTimeout)(ThreadUtils.sameThread)  
}

根据代码清单5-36，NettyRpcEnv的ask方法的执行步骤如下。

1）如果请求消息的接收者的地址与当前NettyRpcEnv的地址相同（则说明处理请求的RpcEndpoint位于本地的NettyRpcEnv中），那么新建Promise对象，并且给Promise的future（类型为Future）设置完成时的回调函数（成功时调用onSuccess方法，失败时调用onFailure方法）。发送消息最终通过调用本地Dispatcher的postLocalMessage方法（见代码清单5-15）实现。

2）如果请求消息的接收者的地址与当前NettyRpcEnv的地址不同（则说明处理请求的RpcEndpoint位于其他节点的NettyRpcEnv中），那么将message序列化，并与onFailure、onSuccess方法一道封装为RpcOutboxMessage类型的消息。最后调用postToOutbox方法将消息投递出去。

3）使用timeoutScheduler设置一个定时器，用于超时处理。此定时器在等待指定的超时时间后将抛出TimeoutException异常。请求如果在超时时间内处理完毕，则会调用timeoutScheduler的cancel方法取消此超时定时器。

4）返回请求处理的结果。

（2）发送消息

NettyRpcEnv重写了抽象类RpcEnv的send方法，其实现如代码清单5-37所示。

代码清单5-37 send的实现

private[netty] def send(message: RequestMessage): Unit = {  
 val remoteAddr = message.receiver.address  
 if (remoteAddr == address) {  
 // Message to a local RPC endpoint.  
 try {  
 dispatcher.postOneWayMessage(message)  
 } catch {  
 case e: RpcEnvStoppedException => logWarning(e.getMessage)  
 }  
 } else {  
 // Message to a remote RPC endpoint.  
 postToOutbox(message.receiver, OneWayOutboxMessage(serialize(message)))  
 }  
}

根据代码清单5-37，NettyRpcEnv的send方法的执行步骤如下。

1）如果请求消息的接收者的地址与当前NettyRpcEnv的地址相同 [[1]](#_1_39) 。那么新建Promise对象，并且给Promise的future（类型为Future）设置完成时的回调函数（成功时调用onSuccess方法，失败时调用onFailure方法）。发送消息最终通过调用本地Dispatcher的postOneWayMessage方法（见代码清单5-15）实现。

2）如果请求消息的接收者的地址与当前NettyRpcEnv的地址不同 [[2]](#_2_7) ，那么将message序列化，并与onFailure、onSuccess方法一道封装为RpcOutboxMessage类型的消息。最后调用postToOutbox方法将消息投递出去。

NettyRpcEnv的ask和send方法都调用了私有方法postToOutbox，postToOutbox用于向远端节点上的RpcEndpoint发送消息，其实现如代码清单5-38所示。

代码清单5-38 postToOutbox的实现

private def postToOutbox(receiver: NettyRpcEndpointRef, message: OutboxMessage): Unit = {  
 if (receiver.client != null) {  
 message.sendWith(receiver.client)  
 } else {  
 require(receiver.address != null,  
 "Cannot send message to client endpoint with no listen address.")  
 val targetOutbox = {  
 val outbox = outboxes.get(receiver.address)  
 if (outbox == null) {  
 val newOutbox = new Outbox(this, receiver.address)  
 val oldOutbox = outboxes.putIfAbsent(receiver.address, newOutbox)  
 if (oldOutbox == null) {  
 newOutbox  
 } else {  
 oldOutbox  
 }  
 } else {  
 outbox  
 }  
 }  
 if (stopped.get) {  
 // It's possible that we put 'targetOutbox' after stopping. So we need to clean it.  
 outboxes.remove(receiver.address)  
 targetOutbox.stop()  
 } else {  
 targetOutbox.send(message)  
 }  
 }  
}

根据代码清单5-38，postToOutbox方法的执行步骤如下。

1）如果NettyRpcEndpointRef中的TransportClient不为空，则直接调用Outbox-Message的sendWith方法，否则进入第2）步。

2）获取NettyRpcEndpointRef的远端RpcEndpoint地址所对应的Outbox。首先从outboxes缓存中获取Outbox。如果outboxes中没有相应的Outbox，则需要新建Outbox并放入outboxes缓存中。

3）如果当前NettyRpcEnv已经处于停止状态，则将第2）步得到的Outbox从outboxes中移除，并且调用Outbox的stop方法停止Outbox。如果当前NettyRpcEnv还未停止，则调用第2）步得到的Outbox的send方法发送消息。

Outbox的stop方法用于停止Outbox，其实现如代码清单5-39所示。

代码清单5-39 Outbox的stop方法

def stop(): Unit = {  
 synchronized {  
 if (stopped) {  
 return  
 }  
 stopped = true  
 if (connectFuture != null) {  
 connectFuture.cancel(true)  
 }  
 closeClient()  
 }  
  
 var message = messages.poll()  
 while (message != null) {  
 message.onFailure(new SparkException("Message is dropped because Outbox is stopped"))  
 message = messages.poll()  
 }  
}

根据代码清单5-39，可以看到停止Outbox的效果包括如下几项。

·将Outbox的停止状态stopped置为true。

·关闭Outbox中的TransportClient。

·清空messages列表中的消息。

根据本节对客户端发送请求的分析，现在我们可以将此流程用图5-10表示。

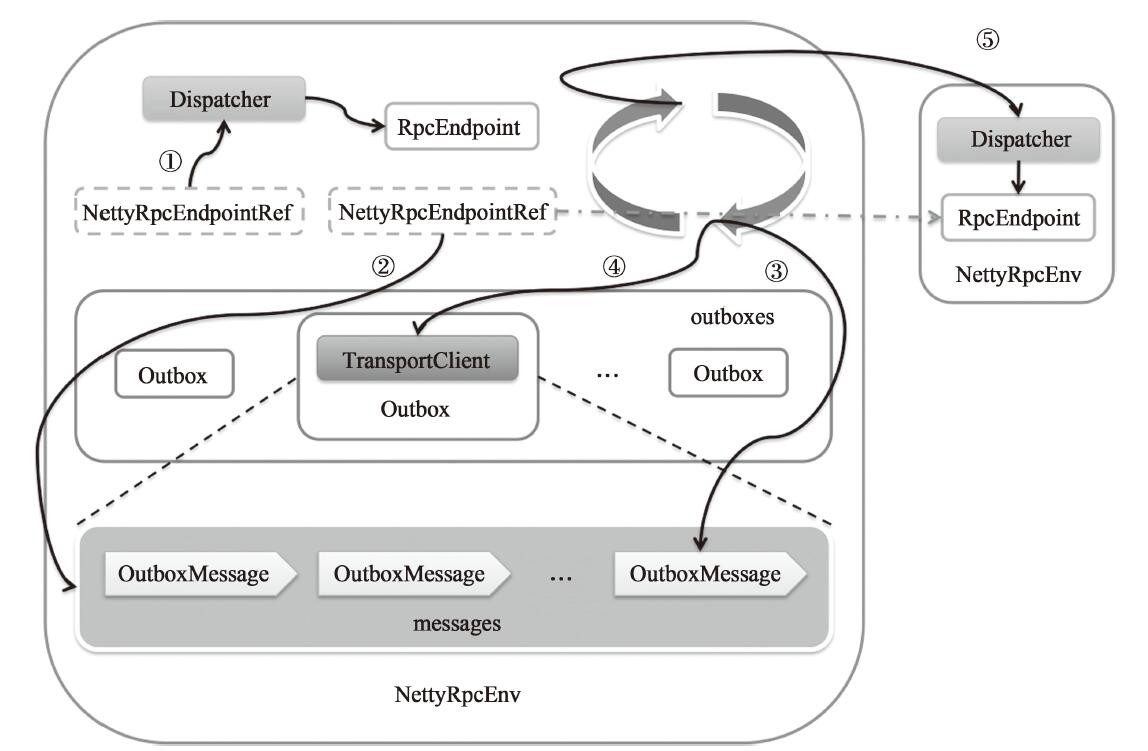


图5-10 RPC客户端发送请求示意图

图5-10展示了两个不同节点上的NettyRpcEnv。右边的NettyRpcEnv采用简略的表示方法，只展示了其内部的Dispatcher组件，实际上右边的NettyRpcEnv与左边的NettyRpcEnv在结构和组成上是一样的。除了Dispatcher组件，还展示了NettyRpcEnv内部的outboxes列表、outboxes列表内的Outbox及Outbox内部用于缓存OutboxMessage的messages列表。左边NettyRpcEnv中的NettyRpcEndpointRef和右边一个提供服务的RpcEndpoint之间有箭头的虚线表示NettyRpcEndpointRef引用RpcEndpoint，也就是说NettyRpcEndpointRef知道RpcEndpoint的地址信息。有了这些简短的陈述，我们可以对图5-10中各个序号的含义进行简单的说明。

序号①： 表示通过调用NettyRpcEndpointRef的send和ask方法向本地节点的Rpc-Endpoint发送消息。由于是在同一节点，所以直接调用Dispatcher的postLocalMessage或postOneWayMessage方法将消息放入EndpointData内部Inbox的messages列表中。Message-Loop线程最后处理消息，并将消息发给对应的RpcEndpoint处理。

序号②： 表示通过调用NettyRpcEndpointRef的send和ask方法向远端节点的Rpc-Endpoint发送消息。这种情况下，消息将首先被封装为OutboxMessage，然后放入到远端RpcEndpoint的地址所对应的Outbox的messages列表中。

序号③： 表示每个Outbox的drainOutbox方法通过循环，不断从messages列表中取得OutboxMessage。

序号④： 表示每个Outbox的drainOutbox方法使用Outbox内部的TransportClient向远端的NettyRpcEnv发送序号③中取得的OutboxMessage。

序号⑤： 表示序号④发出的请求在与远端NettyRpcEnv的TransportServer建立了连接后，请求消息首先经过Netty管道的处理，然后经由NettyRpcHandler处理，最后NettyRpcHandler的receive方法会调用Dispatcher的postRemoteMessage或postOneWay-Message方法，将消息放入EndpointData内部Inbox的messages列表中。MessageLoop线程最后处理消息，并将消息发给对应的RpcEndpoint处理。

[[1]](#_1_38) 说明处理请求的RpcEndpoint位于本地的NettyRpcEnv 中。

[[2]](#_2_6) 说明处理请求的RpcEndpoint位于其他节点的NettyRpcEnv中。

### 5.3.9 NettyRpcEnv中的常用方法

在NettyRpcEnv中有一些方法经常要用到，所以对它们做一些简单介绍。

1）获取RpcEndpoint的引用对象RpcEndpointRef。NettyRpcEnv重写了抽象类RpcEnv的endpointRef方法，其实现如代码清单5-40所示。

代码清单5-40 获取RpcEndpoint的引用

override def endpointRef(endpoint: RpcEndpoint): RpcEndpointRef = {  
 dispatcher.getRpcEndpointRef(endpoint)  
}

可以看到，endpointRef方法的实质是调用了Dispatcher的getRpcEndpointRef方法。getRpcEndpointRef实际是从endpointRefs缓存中获取的RpcEndpoint的引用对象RpcEndpoint-Ref，如代码清单5-41所示。

代码清单5-41 Dispatcher的getRpcEndpointRef方法

def getRpcEndpointRef(endpoint: RpcEndpoint): RpcEndpointRef = endpointRefs.get(endpoint)

2）使用RpcAddress和RpcEndpoint的名称，得到对应的RpcEndpointRef。setupEndpoint-Ref方法实际是NettyRpcEnv继承自父类RpcEnv的方法，其实现如代码清单5-42所示。

代码清单5-42 setupEndpointRef的实现

def setupEndpointRef(address: RpcAddress, endpointName: String): RpcEndpointRef = {  
 setupEndpointRefByURI(RpcEndpointAddress(address, endpointName).toString)  
}

可以看到setupEndpointRef方法内部调用了setupEndpointRefByURI，其实现如代码清单5-43所示。

代码清单5-43 setupEndpointRefByURI的实现

def setupEndpointRefByURI(uri: String): RpcEndpointRef = {  
 defaultLookupTimeout.awaitResult(asyncSetupEndpointRefByURI(uri))  
}

这里的defaultLookupTimeout是通过调用RpcUtils的lookupRpcTimeout方法（详见附录H）得到的查找超时时长。asyncSetupEndpointRefByURI的功能是向远端NettyRpcEnv询问指定名称的RpcEndpoint的NettyRpcEndpointRef。可以看到这里调用NettyRpcEndpointRef的ask方法（5.3.8节对此有详细介绍）向远端NettyRpcEnv中的RpcEndpointVerifier发送RpcEndpointVerifier.CheckExistence类型的消息。RpcEndpointVerifier的实现已在代码清单5-27中详细介绍过。asyncSetupEndpointRefByURI的实现如代码清单5-44所示。

代码清单5-44 异步获取远端NettyRpcEndpointRef

def asyncSetupEndpointRefByURI(uri: String): Future[RpcEndpointRef] = {  
 val addr = RpcEndpointAddress(uri)  
 val endpointRef = new NettyRpcEndpointRef(conf, addr, this)  
 val verifier = new NettyRpcEndpointRef(  
 conf, RpcEndpointAddress(addr.rpcAddress, RpcEndpointVerifier.NAME), this)  
 verifier.ask[Boolean](RpcEndpointVerifier. CheckExistence (endpointRef.name)).flatMap { find =>  
 if (find) {  
 Future.successful(endpointRef)  
 } else {  
 Future.failed(new RpcEndpointNotFoundException(uri))  
 }  
 }(ThreadUtils.sameThread)  
}

·停止RpcEndpointRef所引用的RpcEndpoint。NettyRpcEnv重写了抽象类RpcEnv的stop方法，其实现如代码清单5-45所示。

代码清单5-45 停止RpcEndpoint

override def stop(endpointRef: RpcEndpointRef): Unit = {  
 require(endpointRef.isInstanceOf[NettyRpcEndpointRef])  
 dispatcher.stop(endpointRef)  
}

可以看到，stop方法的实质是调用了Dispatcher的stop方法（见代码清单5-10）。Dispatcher的stop方法对RpcEndpoint去注册，此RpcEndpoint相关联的数据（即Endpoint-Data关联的RpcEndpoint、RpcEndpointRef和Inbox）都将被移除，RpcEndpoint也会停止。

4）设置Endpoint。其实现如代码清单5-46所示。

代码清单5-46 设置Endpoint

override def setupEndpoint(name: String, endpoint: RpcEndpoint): RpcEndpointRef = {  
 dispatcher.registerRpcEndpoint(name, endpoint)  
}

从代码清单5-46可以看到，实际是调用了代码清单5-9中Dispatcher的registerRpc-Endpoint方法注册Endpoint。