深入解析Spark中的RPC

2017-09-01 Neo Spark技术日报

Spark是一个快速的、通用的分布式计算系统，而分布式的特性就意味着，必然存在节点间的通信。本文主要介绍不同的Spark组件之间是如何通过RPC（Remote Procedure Call) 进行点对点通信的，分为三个章节（前排提示：文中所有标蓝部分均可点击底部阅读原文获取详情）：

- Spark RPC的简单示例和实际应用；
- Spark RPC模块的设计原理；
- Spark RPC核心技术总结。

Spark RPC的简单示例和实际应用

Spark的RPC主要在两个模块中：

- 在Spark-core中，主要承载了更好的封装server和client的作用，以及和scala语言的融合，它依赖于模块org.apache.spark.spark-network-common；
- 在org.apache.spark.spark-network-common中，该模块是java语言编写的，最新版本是基于netty4开发的，提供全双工、多路复用I/O模型的Socket I/O能力，Spark的传输协议结构（wire protocol）也是自定义的。

为了更好的了解Spark RPC的内部实现细节，我基于Spark 2.1版本抽离了RPC通信的部分，单独启了一个项目，放到了github以及发布到Maven中央仓库做学习使用，提供了比较好的上手文档、参数设置和性能评估。下面就通过这个模块对Spark RPC先做一个感性的认识。

以下的代码均可以在kraps-rpc找到。

1.1 简单示例

假设我们要开发一个Hello服务，客户端可以传输string，服务端响应hi或者bye，并echo回去输入的string。

第一步，定义一个HelloEndpoint继承自RpcEndpoint表明可以并发的调用该服务，如果继承自ThreadSafeRpcEndpoint则表明该Endpoint不允许并发。

```
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)
```

和Java传统的RPC解决方案对比，可以看出这里不用定义接口或者方法标示（比如通常的id或者name），使用scala的模式匹配进行方法的路由。虽然点对点通信的契约交换受制于语言，这里就是SayHi和SayBye两个case class，但是Spark RPC定位于内部组件通信，所以无伤大雅。

第二步，把刚刚开发好的Endpoint交给Spark RPC管理其生命周期，用于响应外部请求。RpcEnvServerConfig可以定义一些参数、server名称（仅仅是一个标识）、bind地址和端口。通过NettyRpcEnvFactory这个工厂方法，生成RpcEnv，RpcEnv是整个Spark RPC的核心所在，后文会详细展开，通过setupEndpoint将”hello-service”这个名字和第一步定义的Endpoint绑定，后续client调用路由到这个Endpoint就需要”hello-service”这个名字。调用awaitTermination来阻塞服务端监听请求并且处理。

```
val config = RpcEnvServerConfig(new RpcConf(), "hello-server", "localhost", 52345)
val rpcEnv: RpcEnv = NettyRpcEnvFactory.create(config)
val helloEndpoint: RpcEndpoint = new HelloEndpoint(rpcEnv)
rpcEnv.setupEndpoint("hello-service", helloEndpoint)
rpcEnv.awaitTermination()
```

第三步，开发一个client调用刚刚启动的server，首先RpcEnvClientConfig和RpcEnv都是必须的，然后通过刚刚提到的”hello-service”名字新建一个远程Endpoint的引用（Ref），可以看做是stub，用于调用，这里首先展示通过异步的方式来做请求。

```
val rpcConf = new RpcConf()
val config = RpcEnvClientConfig(rpcConf, "hello-client")
val rpcEnv: RpcEnv = NettyRpcEnvFactory.create(config)
val endPointRef: RpcEndpointRef = rpcEnv.setupEndpointRef(RpcAddress("localhost", 52345), "hell-service")
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"))
```

也可以通过同步的方式，在最新的Spark中askWithRetry实际已更名为askSync。

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

这就是Spark RPC的通信过程，使用起来易用性可想而知，非常简单，RPC框架屏蔽了Socket I/O模型、线程模型、序列化/反序列化过程、使用netty做了包识别，长连接，网络重连重试等机制。

1.2 实际应用

在Spark内部，很多的Endpoint以及EndpointRef与之通信都是通过这种形式的，举例来说比如driver和executor之间的交互用到了心跳机制，使用HeartbeatReceiver来实现，这也是一个Endpoint，它的注册在SparkContext初始化的时候做的，代码如下：

```
_heartbeatReceiver = env.rpcEnv.setupEndpoint(HeartbeatReceiver.ENDPOINT_NAME, new HeartbeatReceiver(this))
```

而它的调用在Executor内的方式如下：

```
val message = Heartbeat(executorId, accumUpdates.toArray, env.blockManager.blockManagerId)
val response = heartbeatReceiverRef.askWithRetry[HeartbeatResponse](message, RpcTimeout(conf, "spark.executor.heartbeatInterval", "10s"))
```

二 ▸ Spark RPC模块的设计原理

首先说明下，自Spark 2.0后已经把Akka这个RPC框架剥离出去了（详细见SPARK-5293），原因很简单，因为很多用户会使用Akka做消息传递，那么就会和Spark内嵌的版本产生冲突，而Spark也仅仅用了Akka做RPC，所

以2.0之后，基于底层的org.apache.spark.spark-network-common模块实现了一个类似Akka Actor消息传递模式的scala模块，封装在了core里面，kraps-rpc也就是把这个部分从core里面剥离出来独立了一个项目。

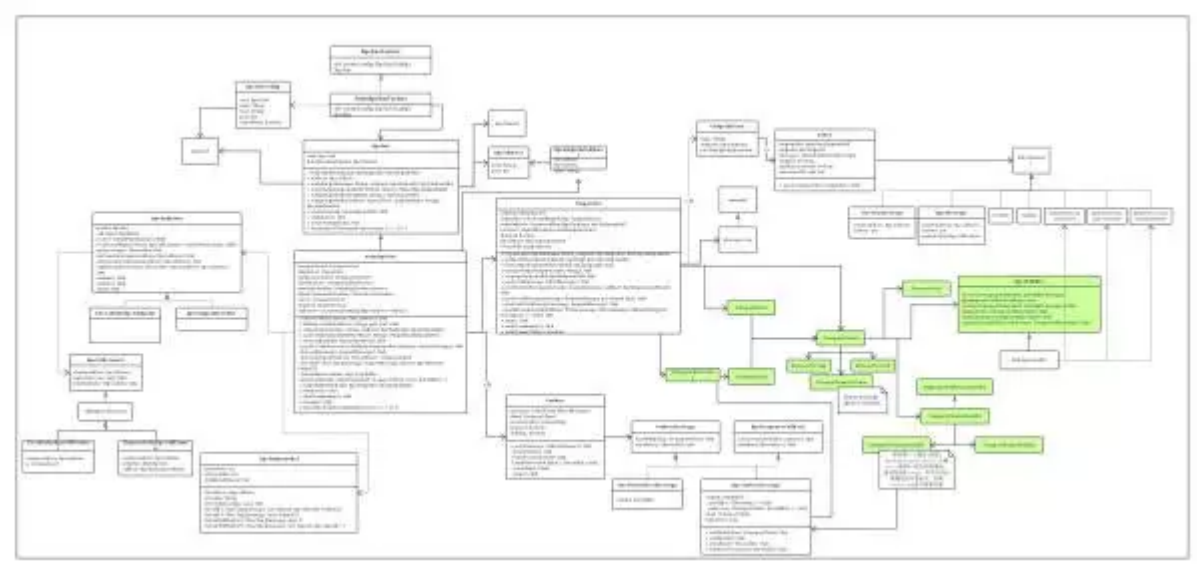
虽然剥离了Akka，但是还是沿袭了Actor模式中的一些概念，在现在的Spark RPC中有如下映射关系。

> RpcEndpoint => Actor
> RpcEndpointRef => ActorRef
> RpcEnv => ActorSystem

底层通信全部使用netty进行了替换，使用的是org.apache.spark.spark-network-common这个内部lib。

2.1 类图分析

这里先上一个UML图展示了Spark RPC模块内的类关系，白色的是Spark-core中的scala类，黄色的是org.apache.spark.spark-network-common中的java类。



不要被这张图所吓倒，经过下面的解释分析，相信读者可以领会其内涵，不用细究其设计的合理度，Spark是一个发展很快、不断演进的项目，代码不是一成不变的，持续变化是一定的。

RpcEndpoint和RpcCallContext

先看最左侧的RpcEndpoint，RpcEndpoint是一个可以响应请求的服务，和Akka中的Actor类似，从它的提供的方法签名（如下）可以看出，receive方法是单向方式的，可以比作UDP，而receiveAndReply是应答方式的，可以比作TCP。它的子类实现可以选择性的覆盖这两个函数，我们第一章实现的HelloEndpoint以及Spark中的HeartbeatReceiver都是它的子类。

```
def receive: PartialFunction[Any, Unit] = {
  case _ => throw new RpcException(self + " does not implement 'receive'")
```

```
}

def receiveAndReply(context: RpcCallContext): PartialFunction[Any, Unit] = {
    case _ => context.sendFailure(new RpcException(self + " won't reply anything"))
}
```

其中RpcCallContext是用于分离核心业务逻辑和底层传输的桥接方法，这也可以看出Spark RPC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模块内）->底层通信（spark-network-common）三者。RpcCallContext可以用于回复正常的响应以及错误异常，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的子类，由于Spark RPC默认采用Java序列化方式，所以异常可以完整的在客户端还原并且作为cause re-throw出去。
```

RpcCallContext也分为了两个子类，分别是LocalNettyRpcCallContext和RemoteNettyRpcCallContext，这个主要是框架内部使用，如果是本地就走LocalNettyRpcCallContext直接调用Endpoint即可，否则就走RemoteNetty-RpcCallContext需要通过RPC和远程交互，这点也体现了RPC的核心概念，就是如何执行另外一个地址空间上的函数、方法，就仿佛在本地调用一样。

另外，RpcEndpoint还提供了一系列回调函数覆盖。

- onError
- onConnected
- onDisconnected
- onNetworkError
- onStart
- onStop
- stop

另外需要注意下，它的一个子类是ThreadSafeRpcEndpoint，很多Spark中的Endpoint继承了这个类，Spark RPC框架对这种Endpoint不做并发处理，也就是同一时间只允许一个线程在做调用。

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。

RpcEndpointRef

RpcEndpointRef类似于Akka中ActorRef，顾名思义，它是RpcEndpoint的引用，提供的方法send等同于!, ask方法等同于?，send用于单向发送请求（RpcEndpoint中的receive响应它），提供fire-and-forget语义，而ask提供

请求响应的语义（RpcEndpoint中的receiveAndReply响应它），默认是需要返回response的，带有超时机制，可以同步阻塞等待，也可以返回一个Future句柄，不阻塞发起请求的工作线程。

RpcEndpointRef是客户端发起请求的入口，它可以从RpcEnv中获取，并且聪明的做本地调用或者RPC。

RpcEnv和NettyRpcEnv

类库中最核心的就是RpcEnv，刚刚提到了这就是ActorSystem，服务端和客户端都可以使用它来做通信。

对于server side来说，RpcEnv是RpcEndpoint的运行环境，负责RpcEndpoint的整个生命周期管理，它可以注册或者销毁Endpoint，解析TCP层的数据包并反序列化，封装成RpcMessage，并且路由请求到指定的Endpoint，调用业务逻辑代码，如果Endpoint需要响应，把返回的对象序列化后通过TCP层再传输到远程对端，如果Endpoint发生异常，那么调用RpcCallContext.sendFailure来把异常发送回去。

对client side来说，通过RpcEnv可以获取RpcEndpoint引用，也就是RpcEndpointRef的。

RpcEnv是和具体的底层通信模块交互的负责人，它的伴生对象包含创建RpcEnv的方法，签名如下：

```
def create(
    name: String,
    bindAddress: String,
    advertiseAddress: String,
    port: Int,
    conf: SparkConf,
    securityManager: SecurityManager,
    numUsableCores: Int,
    clientMode: Boolean): RpcEnv = {
  val config = RpcEnvConfig(conf, name, bindAddress, advertiseAddress, port, securityManager,
    numUsableCores, clientMode)
  new NettyRpcEnvFactory().create(config)
}
```

RpcEnv的创建由RpcEnvFactory负责，RpcEnvFactory目前只有一个子类是NettyRpcEnvFactory，原来还有AkkaRpcEnvFactory。NettyRpcEnvFactory.create方法一旦调用就会立即在bind的address和port上启动server。

它依赖的RpcEnvConfig就是一个包含了SparkConf以及一些参数（kraps-rpc中更名为RpcConf）。RpcEnv的参数都需要从RpcEnvConfig中拿，最基本的hostname和port，还有高级些的连接超时、重试次数、Reactor线程池大小等等。

下面看看RpcEnv最常用的两个方法：

```
// 注册endpoint，必须指定名称，客户端路由就靠这个名称来找endpoint
def setupEndpoint(name: String, endpoint: RpcEndpoint): RpcEndpointRef

// 拿到一个endpoint的引用
def setupEndpointRef(address: RpcAddress, endpointName: String): RpcEndpointRef
```

NettyRpcEnv由NettyRpcEnvFactory.create创建，这是整个Spark core和org.apache.spark.spark-network-common的桥梁，内部leverage底层提供的通信能力，同时包装了一个类Actor的语义。上面两个核心的方法，setupEndpoint会在Dispatcher中注册Endpoint，setupEndpointRef会先去调用RpcEndpointVerifier尝试验证本地或者远程是否存在某个endpoint，然后再创建RpcEndpointRef。更多关于服务端、客户端调用的细节将在时序图中阐述，这里不再展开。

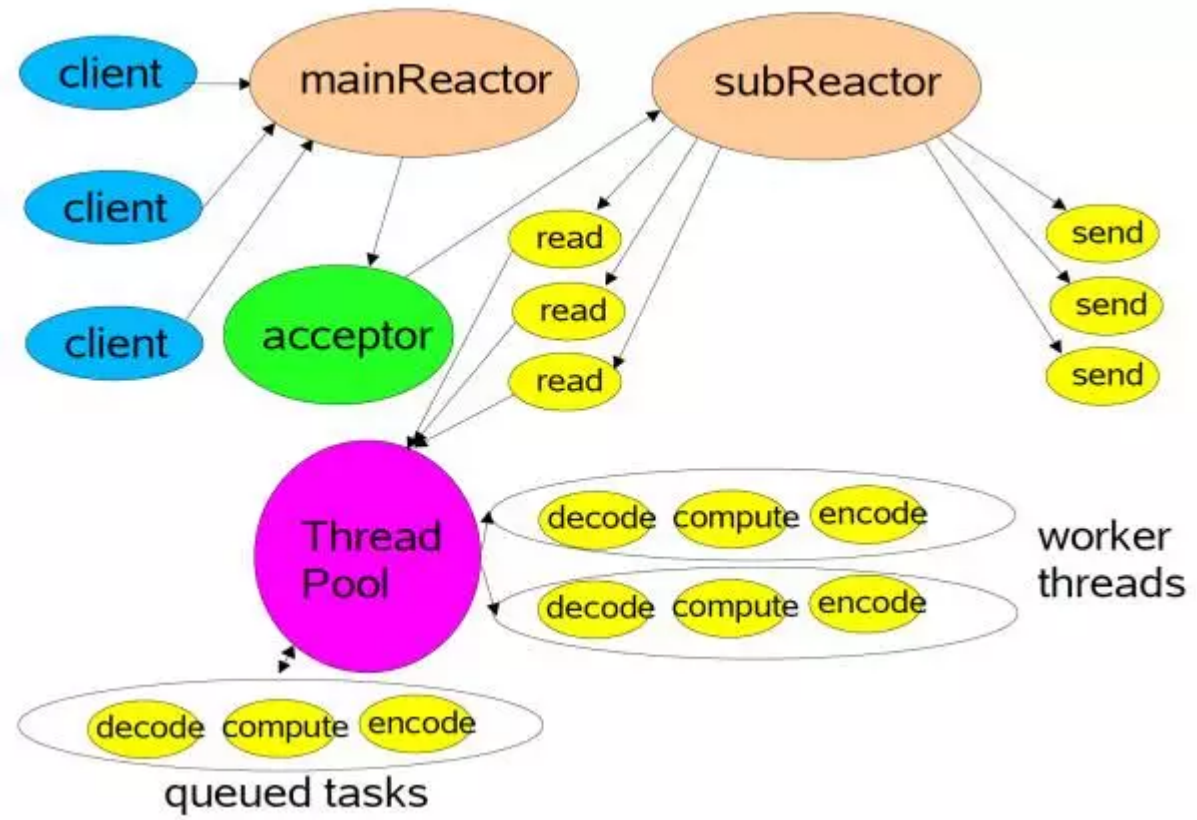
Dispatcher和Inbox

NettyRpcEnv中包含Dispatcher，主要针对服务端，帮助路由到正确的RpcEndpoint，并且调用其业务逻辑。

这里需要先阐述下Reactor模型，Spark RPC的Socket I/O一个典型的Reactor模型的，但是结合了Actor pattern中的mailbox，可谓是一种混合的实现方式。

使用Reactor模型，由底层netty创建的EventLoop做I/O多路复用，这里使用Multiple Reactors这种形式，如下图所示，从netty的角度而言，Main Reactor和Sub Reactor对应BossGroup和WorkerGroup的概念，前者负责监听TCP连接、建立和断开，后者负责真正的I/O读写，而图中的ThreadPool就是的Dispatcher中的线程池，它来解耦开来耗时的业务逻辑和I/O操作，这样就可以更scalabe，只需要少数的线程就可以处理成千上万的连接，这种思想是标准的分治策略，offload非I/O操作到另外的线程池。

真正处理RpcEndpoint的业务逻辑在ThreadPool里面，中间靠Reactor线程中的handler处理decode成RpcMessage，然后投递到Inbox中，所以compute的过程在另外的下面介绍的Dispatcher线程池里面做。



图片来源

刚刚还提到了Actor pattern中mailbox模式，Spark RPC最早起源于Akka，所以进化到现在，仍然了使用了这个模式。这里就介绍Inbox，每个Endpoint都有一个Inbox，Inbox里面有一个InboxMessage的链表，InboxMessage有很多子类，可以是远程调用过来的RpcMessage，可以是远程调用过来的fire-and-forget的单向消息OneWayMessage，还可以是各种服务启动，链路建立断开等Message，这些Message都会在Inbox内部的方法内做模式匹配，调用相应的RpcEndpoint的函数（都是一一对应的）。

Dispatcher中包含一个MessageLoop，它读取LinkedBlockingQueue中的投递RpcMessage，根据客户端指定的Endpoint标识，找到Endpoint的Inbox，然后投递进去，由于是阻塞队列，当没有消息的时候自然阻塞，一旦有消息，就开始工作。Dispatcher的ThreadPool负责消费这些Message。

Dispatcher的ThreadPool它使用参数spark.rpc.netty.dispatcher.numThreads来控制数量，如果kill -3 每个Spark driver或者executor进程，都会看到N个dispatcher线程：

> "dispatcher-event-loop-0" #26 daemon prio=5 os_prio=31 tid=0x00007f8877153800 nid=0x7103 waiting on condition [0x000000011f78b000]

那么另外的问题是谁会调用Dispatcher分发Message的方法呢？答案是RpcHandler的子类NettyRpcHandler，这就是Reactor中的线程做的事情。RpcHandler是底层org.apache.spark.spark-network-common提供的handler，当远程的数据包解析成功后，会调用这个handler做处理。

这样就完成了一个完全异步的流程，Network IO通信由底层负责，然后由Dispatcher分发，只要Dispatcher中的InboxMessage的链表足够大，那么就可以让Dispatcher中的ThreadPool慢慢消化消息，和底层的IO解耦开来，

完全在独立的线程中完成，一旦完成Endpoint内部业务逻辑，利用RpcCallContext回调来做消息的返回。

Outbox

NettyRpcEnv中包含一个ConcurrentHashMap[RpcAddress, Outbox]，每个远程Endpoint都对应一个Outbox，这和上面Inbox遥相呼应，是一个mailbox似的实现方式。

和Inbox类似，Outbox内部包含一个OutboxMessage的链表，OutboxMessage有两个子类，OneWayOutbox-Message和RpcOutboxMessage，分别对应调用RpcEndpoint的receive和receiveAndReply方法。

NettyRpcEnv中的send和ask方法会调用指定地址Outbox中的send方法，当远程连接未建立时，会先建立连接，然后去消化OutboxMessage。

同样，一个问题是Outbox中的send方法如何将消息通过Network IO发送出去，如果是ask方法又是如何读取远程响应的呢？答案是send方法通过org.apache.spark.spark-network-common创建的TransportClient发送出去消息，由Reactor线程负责序列化并且发送出去，每个Message都会返回一个UUID，由底层来维护一个发送出去消息与其Callback的HashMap，当Netty收到完整的远程RpcResponse时候，回调响应的Callback，做反序列化，进而回调Spark core中的业务逻辑，做Promise/Future的done，上层退出阻塞。

这也是一个异步的过程，发送消息到Outbox后，直接返回，Network IO通信由底层负责，一旦RPC调用成功或者失败，都会回调上层的函数，做相应的处理。

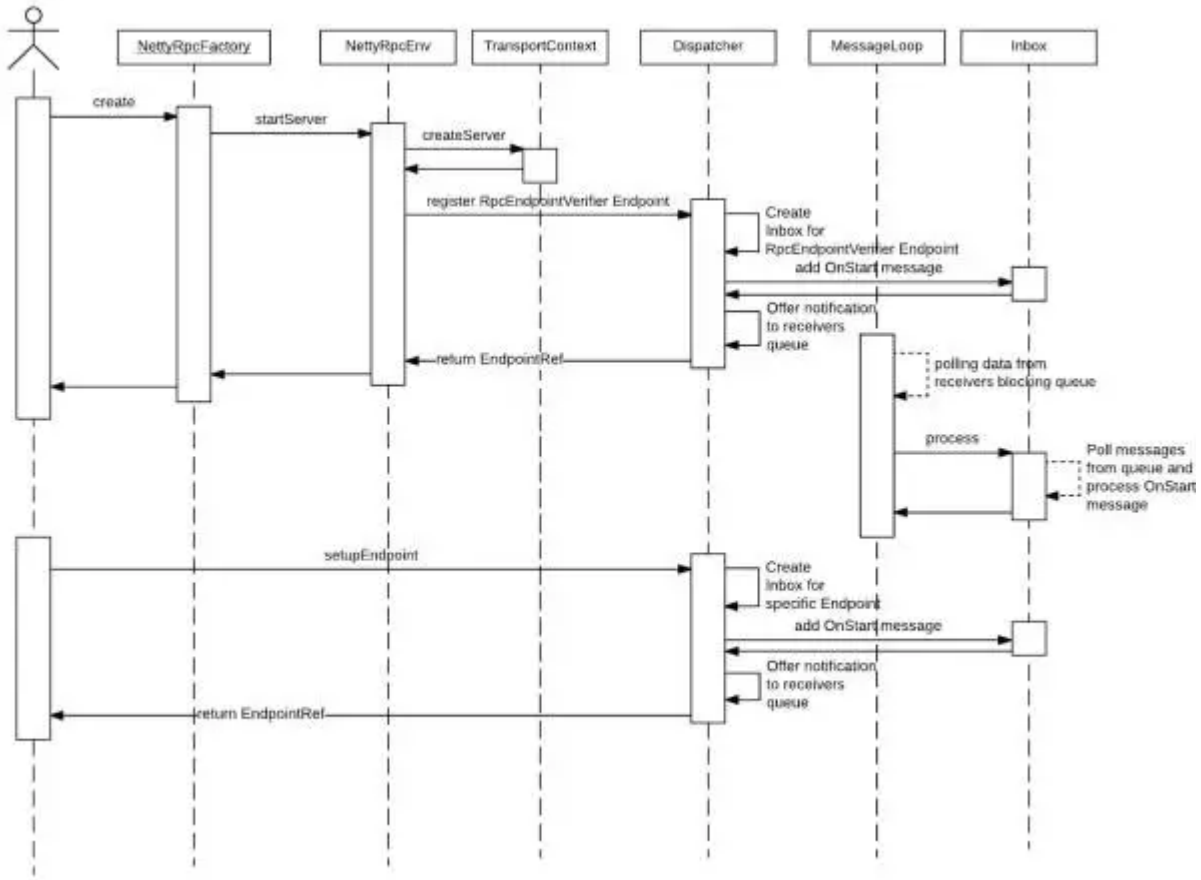
spark-network-common中的类

这里暂不做过多的展开，都是基于Netty的封装，有兴趣的读者可以自行阅读源码，当然还可以参考我之前开源的Navi-pbrpc框架的代码，其原理是基本相同的。

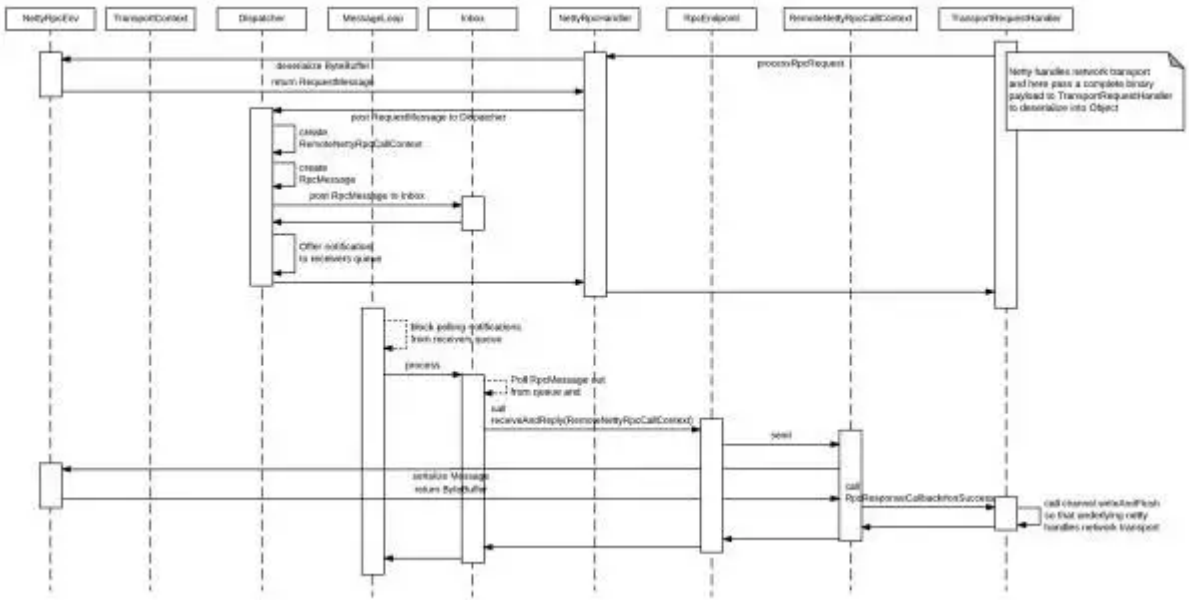
2.2 时序图分析

服务启动

话不多述，直接上图。



服务端响应

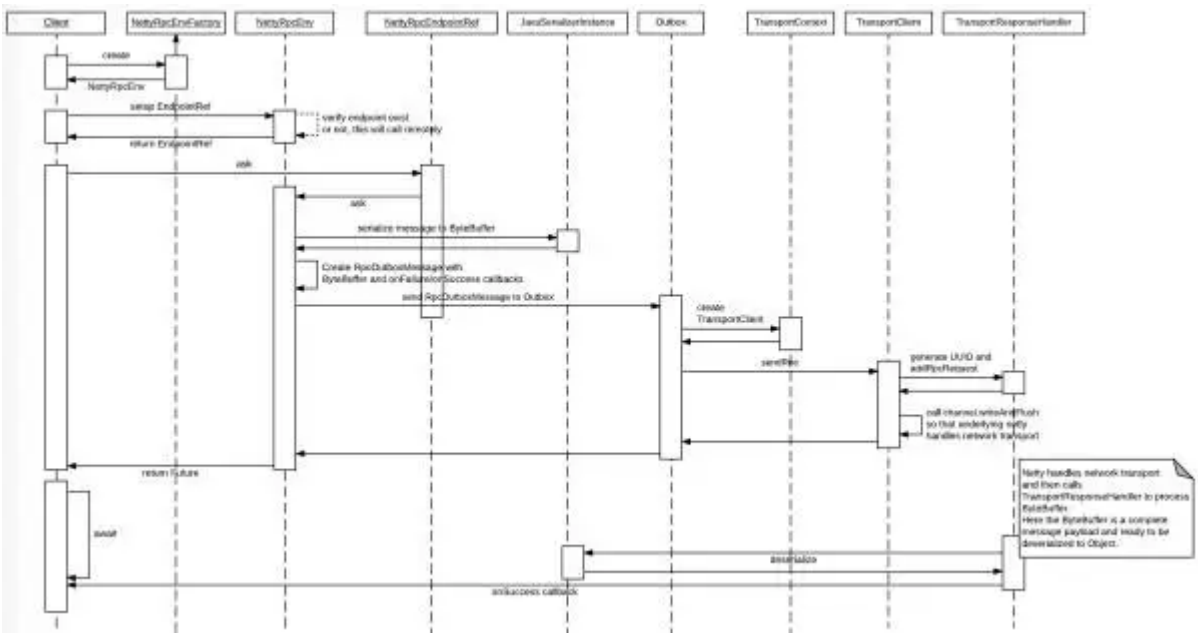


第一阶段，IO接收。TransportRequestHandler是netty的回调handler，它会根据wire format（下文会介绍）解析好一个完整的数据包，交给NettyRpcEnv做反序列化，如果是RPC调用会构造RpcMessage，然后回调RpcHandler的方法处理RpcMessage，内部会调用Dispatcher做RpcMessage的投递，放到Inbox中，到此结束。

第二阶段，IO响应。MessageLoop获取带处理的RpcMessage，交给Dispatcher中的ThreadPool做处理，实际就是调用RpcEndpoint的业务逻辑，通过RpcCallContext将消息序列化，通过回调函数，告诉TransportRequestHandler这有一个消息处理完毕，响应回去。

这里请重点体会异步处理带来的便利，使用Reactor和Actor mailbox的结合的模式，解耦了消息的获取以及处理逻辑。

客户端请求



客户端一般需要先建立RpcEnv，然后获取RpcEndpointRef。

第一阶段，IO发送。利用RpcEndpointRef做send或者ask动作，这里以send为例，send会先进行消息的序列化，然后投递到指定地址的Outbox中，Outbox如果发现连接未建立则先尝试建立连接，然后调用底层的TransportClient发送数据，直接通过该netty的API完成，完成后即可返回，这里返回了UUID作为消息的标识，用于下一个阶段的回调，使用的角度来说可以返回一个Future，客户端可以阻塞或者继续做其他操作。

第二，IO接收。TransportResponseHandler接收到远程的响应后，会先做反序列号，然后回调第一阶段的Future，完成调用，这个过程全部在Reactor线程中完成的，通过Future做线程间的通知。

三 Spark RPC核心技术总结

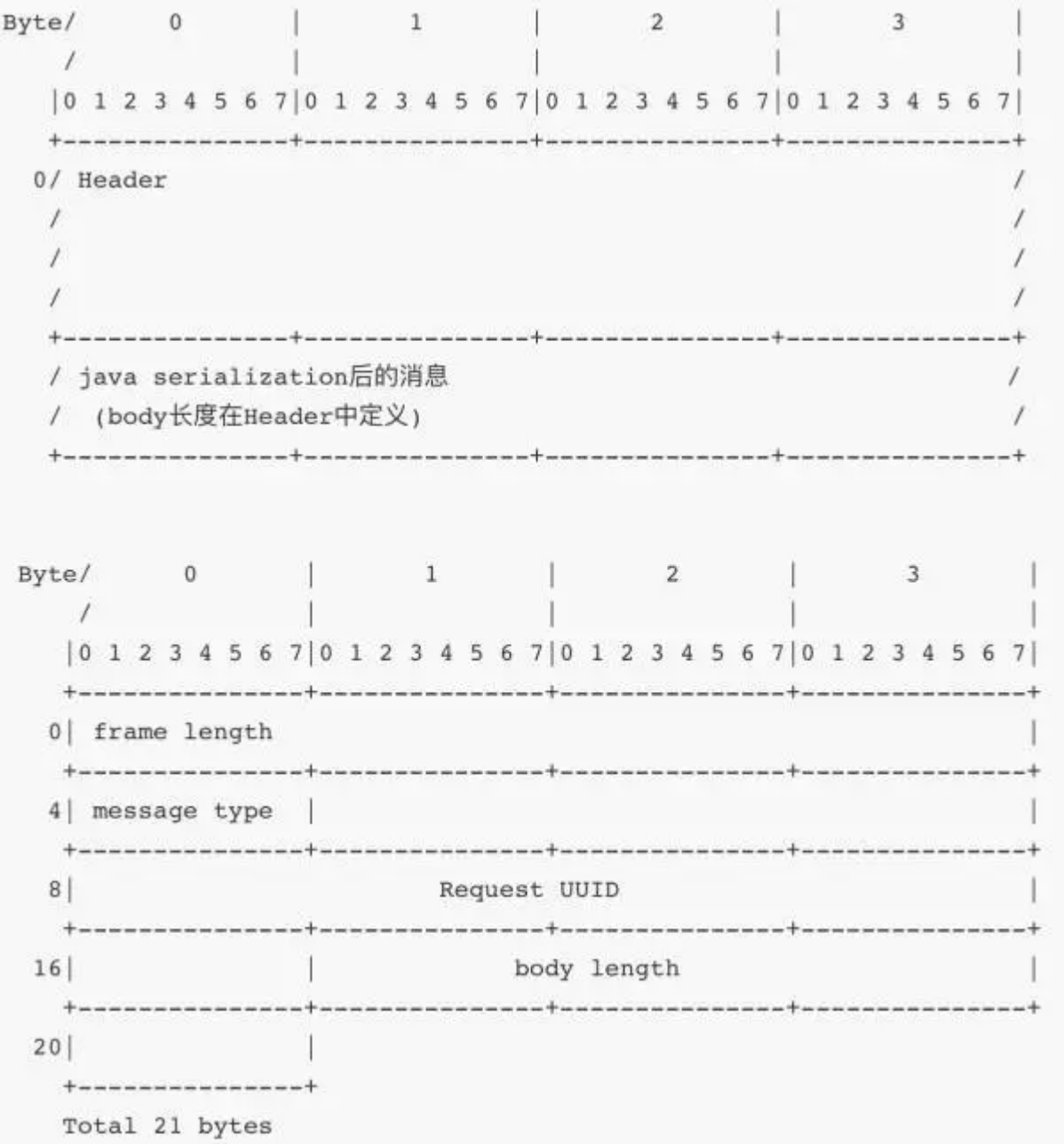
Spark RPC作为RPC传输层选择TCP协议，做可靠的、全双工的binary stream通道。

做一个高性能/scalable的RPC，需要能够满足第一，服务端尽可能多的处理并发请求，第二，同时尽可能短的处理完毕。CPU和I/O之前天然存在着差异，网络传输的延时不可控，CPU资源宝贵，系统进程/线程资源宝贵，为了尽可能避免Socket I/O阻塞服务端和客户端调用，有一些模式（pattern）是可以应用的。Spark RPC的I/O Model由于采用了Netty，因此使用的底层的I/O多路复用（I/O Multiplexing）机制，这里可以通过spark.rpc.io.mode参数设置，不同的平台使用的技术不同，例如linux使用epoll。

线程模型采用Multi-Reactors + mailbox的异步方式来处理，在上文中已经介绍过。

Schema Declaration和序列化方面，Spark RPC默认采用Java native serialization方案，主要从兼容性和JVM平台内部组件通信，以及scala语言的融合考虑，所以不具备跨语言通信的能力，性能上也不是追求极致，目前还没有使用Kyro等更好序列化性能和数据大小的方案。

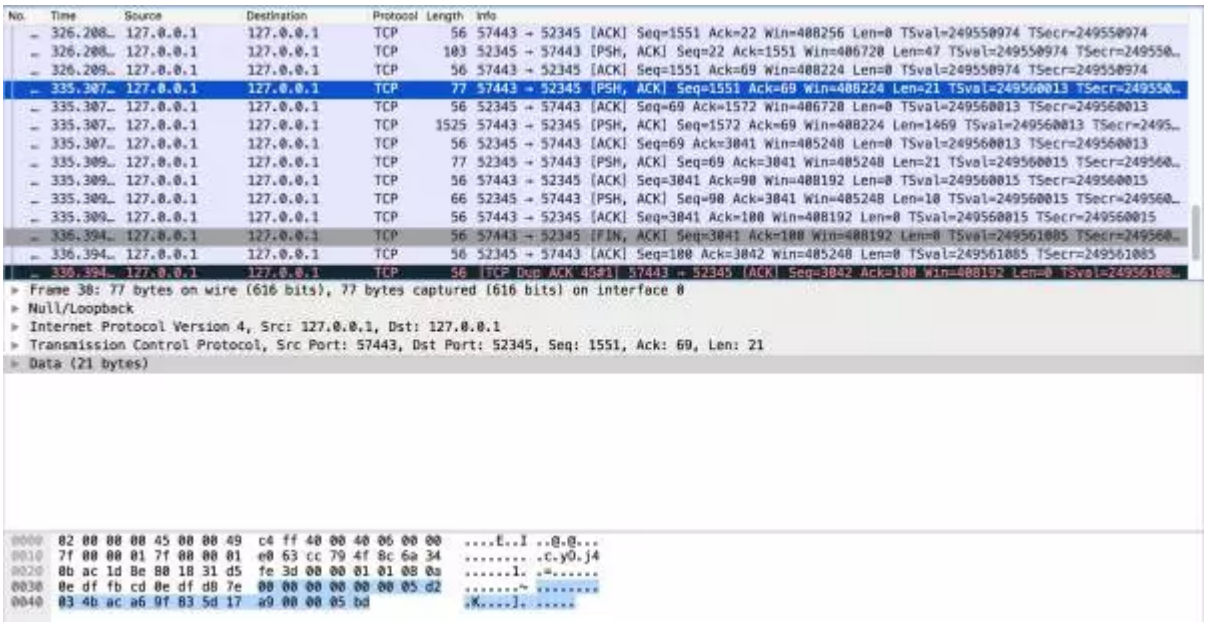
协议结构，Spark RPC采用私有的wire format如下，采用headr+payload的组织方式，header中包括整个frame的长度，message的类型，请求UUID。为解决TCP粘包和半包问题，以及组织成完整的Message的逻辑都在org.apache.spark.network.protocol.MessageEncoder中。



使用wireshake具体分析一下。

首先看一个RPC请求，就是调用第一章说的HelloEndpoint，客户端调用分两个TCP Segment传输，这是因为Spark使用netty的时候header和body分别writeAndFlush出去。

下图是第一个TCP segment：



例子中蓝色的部分是header，头中的字节解析如下：

> 00 00 00 00 00 00 05 d2 // 十进制1490，是整个frame的长度

03一个字节表示的是RpcRequest，枚举定义如下：

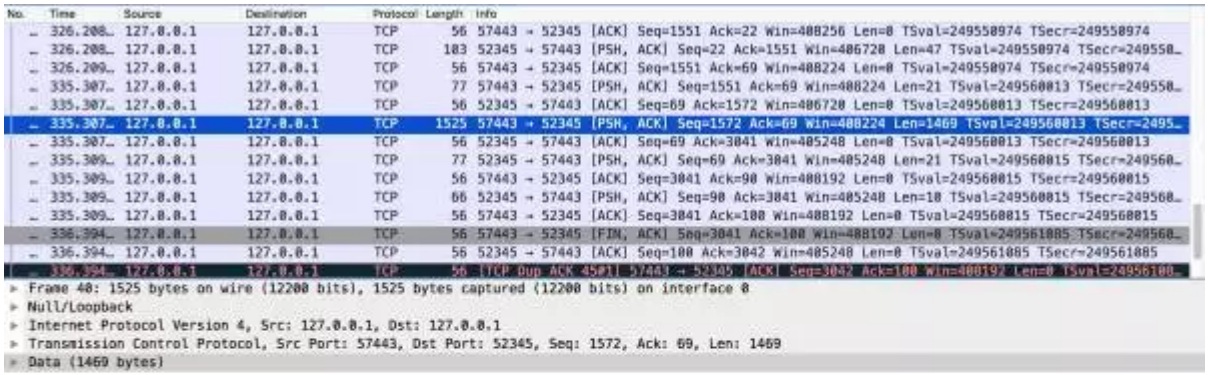
> RpcRequest(3)
> RpcResponse(4)
> RpcFailure(5)
> StreamRequest(6)
> StreamResponse(7)
> StreamFailure(8),
> OneWayMessage(9)
> User(-1)

每个字节的意义如下：

> 4b ac a6 9f 83 5d 17 a9 // 8个字节是UUID
> 05 bd // 十进制1469，payload长度

具体的Payload就长下面这个样子，可以看出使用Java native serialization，一个简单的Echo请求就有1469个字节，还是很大的，序列化的效率不高。但是Spark RPC定位内部通信，不是一个通用的RPC框架，并且使用的量

非常小，所以这点消耗也就可以忽略了，还有Spark Structured Streaming使用该序列化方式，其性能还是可以满足要求的。



另外，作者在kraps-rpc中还给Spark-rpc做了一次性能测试，具体可以参考github。

四 总结

作者从好奇的角度来深度挖掘了下Spark RPC的内幕，并且从2.1版本的Spark core中独立出了一个专门的项目Kraps-rpc，放到了github以及发布到Maven中央仓库做学习使用，提供了比较好的上手文档、参数设置和性能评估，在整合kraps-rpc还发现了一个小的改进点，给Spark提了一个PR——[SPARK-21701]，已经被merge到了主干，算是contribute社区了（10086个开心）。

接着深入剖析了Spark RPC模块内的类组织关系，使用UML类图和时序图帮助读者更好的理解一些核心的概念，包括RpcEnv，RpcEndpoint，RpcEndpointRef等，以及I/O的设计模式，包括I/O多路复用，Reactor和Actor mailbox等，这里还是重点提下Spark RPC的设计哲学，利用netty强大的Socket I/O能力，构建一个异步的通信框架。最后，从TCP层的segment二进制角度分析了wire protocol。

> 作者：Neo，研究生毕业于清华大学，本科毕业于北京邮电大学，目前工作在Hulu，从事Big data相关领域的研发工作，曾经在百度Ecom和程序化广告混迹6年，从事系统研发和架构工作，关注大数据、Web后端技术、广告系统技术以及致力于编写高质量的代码。欢迎访问作者的博客neoremind.com，欢迎技术交流。
> 原文载于知乎，感谢作者授权转载。