## 3.2 Spark内置RPC框架

在Spark中很多地方都涉及网络通信，比如Spark各个组件间的消息互通、用户文件与Jar包的上传、节点间的Shuffle过程、Block数据的复制与备份等。在Spark 0.x.x与Spark 1.x.x版本中，组件间的消息通信主要借助于Akka [[1]](#_1_15) ，使用Akka可以轻松地构建强有力的高并发与分布式应用。但是Akka在Spark 2.0.0版本中被移除了，Spark官网文档对此的描述为：“Akka的依赖被移除了，因此用户可以使用任何版本的Akka来编程了。”Spark团队的决策者或许认为对Akka具体版本的依赖，限制了用户对Akka不同版本的使用。尽管如此，笔者依然认为Akka是一款非常优秀的开源分布式系统，我参与的一些Java Application或者Java Web就利用Akka的丰富特性实现了分布式一致性、最终一致性及分布式事务等分布式环境面对的问题。在Spark 1.x.x版本中，用户文件与Jar包的上传采用了由Jetty [[2]](#_2_3) 实现的HttpFileServer，但在Spark 2.0.0版本中它也被废弃了，现在使用的是基于Spark内置RPC框架的NettyStreamManager。节点间的Shuffle过程和Block数据的复制与备份这两个部分在Spark 2.0.0版本中依然沿用了Netty [[3]](#_3_1) ，通过对接口和程序进行重新设计，将各个组件间的消息互通、用户文件与Jar包的上传等内容统一纳入Spark的RPC框架体系中。

我们先来看看RPC框架的基本架构，如图3-1所示。

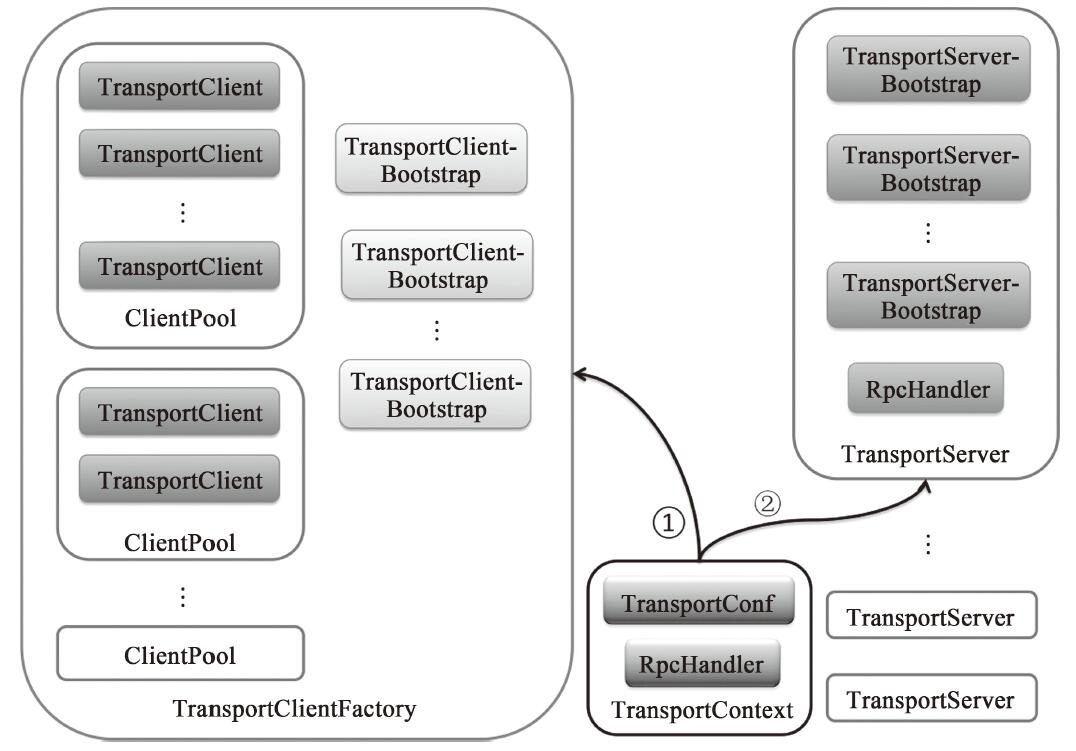


图3-1 Spark内置RPC框架的基本架构

TransportContext内部包含传输上下文的配置信息TransportConf和对客户端请求消息进行处理的RpcHandler。TransportConf在创建TransportClientFactory和TransportServer时都是必需的，而RpcHandler只用于创建TransportServer。TransportClientFactory是RPC客户端的工厂类。TransportServer是RPC服务端的实现。图中记号的含义如下。

·记号①表示通过调用TransportContext的createClientFactory方法创建传输客户端工厂TransportClientFactory的实例。在构造TransportClientFactory的实例时，还会传递客户端引导程序TransportClientBootstrap的列表。此外，TransportClientFactory内部还存在针对每个Socket地址的连接池ClientPool，这个连接池缓存的定义如下：

private final ConcurrentHashMap<SocketAddress, ClientPool> connectionPool;

ClientPool的类型定义如下：

private static class ClientPool {  
 TransportClient[] clients;  
 Object[] locks;  
  
 ClientPool(int size) {  
 clients = new TransportClient[size];  
 locks = new Object[size];  
 for (int i = 0; i < size; i++) {  
 locks[i] = new Object();  
 }  
 }  
}

由此可见，ClientPool实际是由TransportClient的数组构成的，而locks数组中的Object与clients数组中的TransportClient按照数组索引一一对应，通过对每个TransportClient分别采用不同的锁，降低并发情况下线程间对锁的争用，进而减少阻塞，提高并发度。

·记号②表示通过调用TransportContext的createServer方法创建传输服务端TransportServer的实例。在构造TransportServer的实例时，需要传递TransportContext、host、port、RpcHandler及服务端引导程序TransportServerBootstrap的列表。

有了对Spark内置RPC框架的基本架构的了解，现在正式介绍Spark的RPC框架所包含的各个组件。

·TransportContext：传输上下文，包含了用于创建传输服务端（TransportServer）和传输客户端工厂（TransportClientFactory）的上下文信息，并支持使用Transport-ChannelHandler设置Netty提供的SocketChannel的Pipeline的实现。

·TransportConf：传输上下文的配置信息。

·RpcHandler：对调用传输客户端（TransportClient）的sendRPC方法发送的消息进行处理的程序。

·MessageEncoder：在将消息放入管道前，先对消息内容进行编码，防止管道另一端读取时丢包和解析错误。

·MessageDecoder：对从管道中读取的ByteBuf进行解析，防止丢包和解析错误。

·TransportFrameDecoder：对从管道中读取的ByteBuf按照数据帧进行解析。

·RpcResponseCallback：RpcHandler对请求的消息处理完毕后进行回调的接口。

·TransportClientFactory：创建TransportClient的传输客户端工厂类。

·ClientPool：在两个对等节点间维护的关于TransportClient的池子。ClientPool是TransportClientFactory的内部组件。

·TransportClient：RPC框架的客户端，用于获取预先协商好的流中的连续块。TransportClient旨在允许有效传输大量数据，这些数据将被拆分成几百KB到几MB的块。当TransportClient处理从流中获取的块时，实际的设置是在传输层之外完成的。sendRPC方法能够在客户端和服务端的同一水平线的通信进行这些设置。

·TransportClientBootstrap：当服务端响应客户端连接时在客户端执行一次的引导程序。

·TransportRequestHandler：用于处理客户端的请求并在写完块数据后返回的处理程序。

·TransportResponseHandler：用于处理服务端的响应，并且对发出请求的客户端进行响应的处理程序。

·TransportChannelHandler：代理由TransportRequestHandler处理的请求和由Transport-ResponseHandler处理的响应，并加入传输层的处理。

·TransportServerBootstrap：当客户端连接到服务端时在服务端执行一次的引导程序。

·TransportServer：RPC框架的服务端，提供高效的、低级别的流服务。

拓展知识： 为什么需要MessageEncoder和MessageDecoder？因为在基于流的传输里（比如TCP/IP），接收到的数据首先会被存储到一个socket接收缓冲里。不幸的是，基于流的传输并不是一个数据包队列，而是一个字节队列。即使你发送了2个独立的数据包，操作系统也不会作为2个消息处理，而仅仅认为是一连串的字节，因此不能保证远程写入的数据会被准确地读取。举个例子，我们假设操作系统的TCP/TP协议栈已经接收了3个数据包：ABC、DEF、GHI。由于基于流传输的协议的这种统一的性质，你的应用程序在读取数据的时候有很大的可能性被分成下面的片段：AB、CDEFG、H、I。因此，接收方不管是客户端还是服务端，都应该把接收到的数据整理成一个或者多个更有意义并且让程序的逻辑更好理解的数据。

[[1]](#_1_14) Akka是基于Actor并发编程模型实现的并发的分布式框架。Akka是用Scala语言编写的，它提供了Java和Scala两种语言的API，减少了开发人员对并发的细节处理，并保证分布式调用的最终一致性。在附录B中有关于Akka的进一步介绍，感兴趣的读者不妨一读。

[[2]](#_2_2) Jetty是一个开源的Servlet容器，它为基于Java的Web容器，例如JSP和Servlet提供运行环境。Jetty是使用Java语言编写的，它的API以一组Jar包的形式发布。开发人员可以将Jetty容器实例化成一个对象，可以迅速为一些独立运行的Java应用提供网络和Web连接。在附录C中有对Jetty的简单介绍，感兴趣的读者可以选择阅读。

[[3]](#_3) Netty是由Jboss提供的一个基于NIO的客户、服务端编程框架，使用Netty可以确保你快速、简单地开发出一个网络应用，例如实现了某种协议的客户、服务端应用。附录G中有对Netty的简单介绍，感兴趣的读者可以一读。

### 3.2.1 RPC配置TransportConf

上文提到TransportContext中的TransportConf给Spark的RPC框架提供配置信息，它有两个成员属性——配置提供者conf和配置的模块名称module。这两个属性的定义如下：

private final ConfigProvider conf;  
private final String module;

其中conf是真正的配置提供者，其类型ConfigProvider是一个抽象类，如代码清单3-6所示。

代码清单3-6 ConfigProvider的实现

public abstract class ConfigProvider {  
public abstract String get(String name);  
  
public String get(String name, String defaultValue) {  
 try {  
 return get(name);  
 } catch (NoSuchElementException e) {  
 return defaultValue;  
 }  
}  
  
public int getInt(String name, int defaultValue) {  
 return Integer.parseInt(get(name, Integer.toString(defaultValue)));  
}  
  
public long getLong(String name, long defaultValue) {  
 return Long.parseLong(get(name, Long.toString(defaultValue)));  
}  
  
public double getDouble(String name, double defaultValue) {  
 return Double.parseDouble(get(name, Double.toString(defaultValue)));  
}  
  
public boolean getBoolean(String name, boolean defaultValue) {  
 return Boolean.parseBoolean(get(name, Boolean.toString(defaultValue)));  
 }  
}

从代码清单3-6可以看到，ConfigProvider中包括get、getInt、getLong、getDouble、getBoolean等方法，这些方法都是基于抽象方法get获取值，经过一次类型转换而实现。这个抽象的get方法将需要子类去实现。

Spark通常使用SparkTransportConf创建TransportConf，其实现如代码清单3-7所示。

代码清单3-7 SparkTransportConf的实现

object SparkTransportConf {  
 private val MAX\_DEFAULT\_NETTY\_THREADS = 8  
 def fromSparkConf(\_conf: SparkConf, module: String, numUsableCores: Int = 0): TransportConf = {  
 val conf = \_conf.clone  
 val numThreads = defaultNumThreads(numUsableCores)  
 conf.setIfMissing(s"spark.$module.io.serverThreads", numThreads.toString)  
 conf.setIfMissing(s"spark.$module.io.clientThreads", numThreads.toString)  
  
 new TransportConf(module, new ConfigProvider {  
 override def get(name: String): String = conf.get(name)  
 })  
 }  
 private def defaultNumThreads(numUsableCores: Int): Int = {  
 val availableCores =  
 if (numUsableCores > 0) numUsableCores else Runtime.getRuntime.availableProcessors()  
 math.min(availableCores, MAX\_DEFAULT\_NETTY\_THREADS)  
 }  
}

从代码清单3-7看到，可以使用SparkTransportConf的fromSparkConf方法来构造TransportConf。传递的三个参数分别为SparkConf、模块名module及可用的内核数num-UsableCores。如果numUsableCores小于等于0，那么线程数是系统可用处理器的数量，不过系统的内核数不可能全部用于网络传输，所以这里将分配给网络传输的内核数量最多限制在8个。最终确定的线程数将用于设置客户端传输线程数（spark.$module.io.clientThreads属性）和服务端传输线程数（spark.$module.io.serverThreads属性）。from-SparkConf最终构造TransportConf对象时传递的ConfigProvider为实现了get方法的匿名的内部类，get的实现实际是代理了SparkConf的get方法。

### 3.2.2 RPC客户端工厂TransportClientFactory

TransportClientFactory是创建TransportClient的工厂类。在说明图3-1中的记号①时提到过，TransportContext的createClientFactory方法可以创建TransportClientFactory的实例，其实现如代码清单3-8所示。

代码清单3-8 创建传输客户端工厂

public TransportClientFactory createClientFactory(List<TransportClientBootstrap> bootstraps) {  
 return new TransportClientFactory(this, bootstraps);  
}  
  
public TransportClientFactory createClientFactory() {  
 return createClientFactory(Lists.<TransportClientBootstrap>newArrayList());  
}

可以看到，TransportContext中有两个重载的createClientFactory方法，它们最终在构造TransportClientFactory时都会传递两个参数：TransportContext和TransportClientBootstrap列表。TransportClientFactory构造器的实现如代码清单3-9所示。

代码清单3-9 TransportClientFactory的构造器

public TransportClientFactory(  
 TransportContext context,  
 List<TransportClientBootstrap> clientBootstraps) {  
 this.context = Preconditions.checkNotNull(context);  
 this.conf = context.getConf();  
 this.clientBootstraps = Lists.newArrayList(Preconditions.checkNotNull(clientBootstraps));  
 this.connectionPool = new ConcurrentHashMap<>();  
 this.numConnectionsPerPeer = conf.numConnectionsPerPeer();  
 this.rand = new Random();  
  
 IOMode ioMode = IOMode.valueOf(conf.ioMode());  
 this.socketChannelClass = NettyUtils.getClientChannelClass(ioMode);  
 this.workerGroup = NettyUtils.createEventLoop(  
 ioMode,  
 conf.clientThreads(),  
 conf.getModuleName() + "-client");  
 this.pooledAllocator = NettyUtils.createPooledByteBufAllocator(  
 conf.preferDirectBufs(), false /\* allowCache \*/, conf.clientThreads());  
}

TransportClientFactory构造器中的各个变量如下。

·context：参数传递的TransportContext的引用。

·conf：指TransportConf，这里通过调用TransportContext的getConf获取。

·clientBootstraps：参数传递的TransportClientBootstrap列表。

·connectionPool：针对每个Socket地址的连接池ClientPool的缓存。connectionPool的数据结构较为复杂，为便于读者理解，这里以图3-2来表示connectionPool的数据结构。

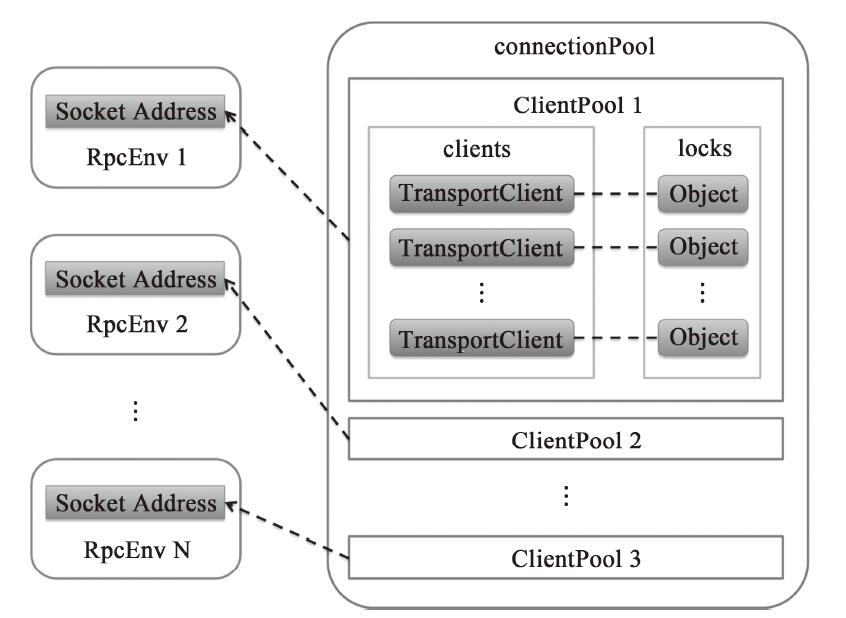


图3-2 TransportClientFactory的connectionPool

·numConnectionsPerPeer：从TransportConf获取的key为“spark.+模块名+.io.num-ConnectionsPerPeer”的属性值。此属性值用于指定对等节点间的连接数。这里的模块名实际为TransportConf的module字段。Spark的很多组件都利用RPC框架构建，它们之间按照模块名区分，例如，RPC模块的key为“spark.rpc.io.num ConnectionsPerPeer”。

·rand：对Socket地址对应的连接池ClientPool中缓存的TransportClient进行随机选择，对每个连接做负载均衡。

·ioMode：IO模式，即从TransportConf获取key为“spark.+模块名+.io.mode”的属性值。默认值为NIO，Spark还支持EPOLL。

·socketChannelClass：客户端Channel被创建时使用的类，通过ioMode来匹配，默认为NioSocketChannel，Spark还支持EpollEventLoopGroup。

·workerGroup：根据Netty的规范，客户端只有worker组，所以此处创建worker-Group。workerGroup的实际类型是NioEventLoopGroup。

·pooledAllocator：汇集ByteBuf但对本地线程缓存禁用的分配器。

·TransportClientFactory里大量使用了NettyUtils，关于NettyUtils的具体实现，请参看附录G [[1]](#_1_17) 。

Image00045.jpg 提示 NIO是Java中New IO的简称，其特点包括为所有的原始类型提供（Buffer）缓冲支持；字符集编码解码解决方案；提供一个新的原始I/O抽象Channel,支持锁和内存映射文件的文件访问接口；提供多路非阻塞式(non-blocking)的高伸缩性网络I/O。其具体使用属于Java语言的范畴，本文不过多介绍。

1.客户端引导程序TransportClientBootstrap

TransportClientFactory的clientBootstraps属性是TransportClientBootstrap的列表。Transport ClientBootstrap是在TransportClient上执行的客户端引导程序，主要对连接建立时进行一些初始化的准备（例如验证、加密）。TransportClientBootstrap所做的操作往往是昂贵的，好在建立的连接可以重用。TransportClientBootstrap的接口定义如代码清单3-10所示。

代码清单3-10 TransportClientBootstrap的定义

public interface TransportClientBootstrap {  
void doBootstrap(TransportClient client, Channel channel) throws RuntimeException;  
}

TransportClientBootstrap有两个实现类：EncryptionDisablerBootstrap和SaslClientBootstrap。为了对TransportClientBootstrap的作用能有更深的了解，这里以EncryptionDisablerBootstrap为例，EncryptionDisablerBootstrap的实现如代码清单3-11所示。

代码清单3-11 EncryptionDisablerBootstrap的实现

private static class EncryptionDisablerBootstrap implements Transport-ClientBootstrap {  
 @Override  
 public void doBootstrap(TransportClient client, Channel channel) {  
 channel.pipeline().remove(SaslEncryption.ENCRYPTION\_HANDLER\_NAME);  
 }  
}

根据代码清单3-11可以看到，EncryptionDisablerBootstrap的作用是移除客户端管道中的SASL加密。

2.创建RPC客户端TransportClient

有了TransportClientFactory，Spark的各个模块就可以使用它创建RPC客户端Transport-Client了。每个TransportClient实例只能和一个远端的RPC服务通信，所以Spark中的组件如果想要和多个RPC服务通信，就需要持有多个TransportClient实例。创建Transport-Client的方法如代码清单3-12所示（实际为从缓存中获取TransportClient）。

代码清单3-12 从缓存中获取TransportClient

public TransportClient createClient(String remoteHost, int remotePort)  
 throws IOException, InterruptedException {  
 // 创建InetSocketAddress  
 final InetSocketAddress unresolvedAddress =  
 InetSocketAddress.createUnresolved(remoteHost, remotePort);  
  
 ClientPool clientPool = connectionPool.get(unresolvedAddress);  
 if (clientPool == null) {  
 connectionPool.putIfAbsent(unresolvedAddress, new ClientPool(numConnectionsPerPeer));  
 clientPool = connectionPool.get(unresolvedAddress);  
 }  
   
 // 随机选择一个TransportClient  
 int clientIndex = rand.nextInt(numConnectionsPerPeer);   
 TransportClient cachedClient = clientPool.clients[clientIndex];  
  
 if (cachedClient != null && cachedClient.isActive()) {// 获取并返回激活的  
 //TransportClient  
 TransportChannelHandler handler = cachedClient.getChannel().pipeline()  
 .get(TransportChannelHandler.class);  
 synchronized (handler) {  
 handler.getResponseHandler().updateTimeOfLastRequest();  
 }  
  
 if (cachedClient.isActive()) {  
 logger.trace("Returning cached connection to {}: {}",  
 cachedClient.getSocketAddress(), cachedClient);  
 return cachedClient;  
 }  
 }  
  
 final long preResolveHost = System.nanoTime();  
 final InetSocketAddress resolvedAddress = new InetSocketAddress(remoteHost, remotePort);  
 final long hostResolveTimeMs = (System.nanoTime() - preResolveHost) / 1000000;  
 if (hostResolveTimeMs > 2000) {  
 logger.warn("DNS resolution for {} took {} ms", resolvedAddress, host ResolveTimeMs);  
 } else {  
 logger.trace("DNS resolution for {} took {} ms", resolvedAddress, host ResolveTimeMs);  
 }  
 // 创建并返回TransportClient对象  
 synchronized (clientPool.locks[clientIndex]) {  
 cachedClient = clientPool.clients[clientIndex];  
  
 if (cachedClient != null) {  
 if (cachedClient.isActive()) {  
 logger.trace("Returning cached connection to {}: {}", resolvedAddress, cachedClient);  
 return cachedClient;  
 } else {  
 logger.info("Found inactive connection to {}, creating a new one.", resolvedAddress);  
 }  
 }  
 clientPool.clients[clientIndex] = createClient(resolvedAddress);   
 return clientPool.clients[clientIndex];  
 }  
}

从代码清单3-12得知，创建TransportClient的步骤如下。

1）调用InetSocketAddress的静态方法createUnresolved构建InetSocketAddress（这种方式创建InetSocketAddress，可以在缓存中已经有TransportClient时避免不必要的域名解析），然后从connectionPool中获取与此地址对应的ClientPool，如果没有，则需要新建ClientPool，并放入缓存connectionPool中。

2）根据numConnectionsPerPeer的大小（使用“spark.+模块名+.io.numConnections-PerPeer”属性配置），从ClientPool中随机选择一个TransportClient。

3）如果ClientPool的clients数组中在随机产生的索引位置不存在TransportClient或者TransportClient没有激活，则进入第5步，否则对此TransportClient进行第4步的检查。

4）更新TransportClient的channel中配置的TransportChannelHandler的最后一次使用时间，确保channel没有超时，然后检查TransportClient是否是激活状态，最后返回此TransportClient给调用方。

5）由于缓存中没有TransportClient可用，于是调用InetSocketAddress的构造器创建InetSocketAddress对象（直接使用InetSocketAddress的构造器创建InetSocketAddress会进行域名解析），在这一步骤多个线程可能会产生竞态条件（由于没有同步处理，所以多个线程极有可能同时执行到此处，都发现缓存中没有TransportClient可用，于是都使用InetSocketAddress的构造器创建InetSocketAddress）。

6）第5步创建InetSocketAddress的过程中产生的竞态条件如果不妥善处理，会产生线程安全问题，所以到了ClientPool的locks数组发挥作用的时候了。按照随机产生的数组索引，locks数组中的锁对象可以对clients数组中的TransportClient一对一进行同步。即便之前产生了竞态条件，但是在这一步只能有一个线程进入临界区。在临界区内，先进入的线程调用重载的createClient方法创建TransportClient对象并放入ClientPool的clients数组中。当率先进入临界区的线程退出临界区后，其他线程才能进入，此时发现ClientPool的clients数组中已经存在了TransportClient对象，那么将不再创建TransportClient，而是直接使用它。

代码清单3-12的整个执行过程实际解决了TransportClient缓存的使用及createClient方法的线程安全问题，并没有涉及创建TransportClient的实现。TransportClient的创建过程在重载的createClient方法（见代码清单3-13）中实现。

代码清单3-13 创建TransportClient

private TransportClient createClient(InetSocketAddress address)  
 throws IOException, InterruptedException {  
 logger.debug("Creating new connection to {}", address);  
 // 构建根引导程序Bootstrap并对其进行配置  
 Bootstrap bootstrap = new Bootstrap();  
 bootstrap.group(workerGroup)  
 .channel(socketChannelClass)  
 .option(ChannelOption.TCP\_NODELAY, true)  
 .option(ChannelOption.SO\_KEEPALIVE, true)  
 .option(ChannelOption.CONNECT\_TIMEOUT\_MILLIS, conf.connectionTimeoutMs())  
 .option(ChannelOption.ALLOCATOR, pooledAllocator);  
  
 final AtomicReference<TransportClient> clientRef = new AtomicReference<>();  
 final AtomicReference<Channel> channelRef = new AtomicReference<>();  
 // 为根引导程序设置管道初始化回调函数  
 bootstrap.handler(new ChannelInitializer<SocketChannel>() {  
 @Override  
 public void initChannel(SocketChannel ch) {  
 TransportChannelHandler clientHandler = context.initializePipeline(ch);  
 clientRef.set(clientHandler.getClient());  
 channelRef.set(ch);  
 }  
 });  
  
 long preConnect = System.nanoTime();  
 ChannelFuture cf = bootstrap.connect(address);// 使用根引导程序连接远程服务器  
 if (!cf.await(conf.connectionTimeoutMs())) {  
 throw new IOException(  
 String.format("Connecting to %s timed out (%s ms)", address, conf.connectionTimeoutMs()));  
 } else if (cf.cause() != null) {  
 throw new IOException(String.format("Failed to connect to %s", address), cf.cause());  
 }  
  
 TransportClient client = clientRef.get();  
 Channel channel = channelRef.get();  
 assert client != null : "Channel future completed successfully with null client";  
  
 // Execute any client bootstraps synchronously before marking the Client as successful.  
 long preBootstrap = System.nanoTime();  
 logger.debug("Connection to {} successful, running bootstraps...", address);  
 try {  
 for (TransportClientBootstrap clientBootstrap : clientBootstraps) {  
 clientBootstrap.doBootstrap(client, channel);// 给TransportClient设置客户端引导程序  
 }  
 } catch (Exception e) { // catch non-RuntimeExceptions too as bootstrap may be written in Scala  
 long bootstrapTimeMs = (System.nanoTime() - preBootstrap) / 1000000;  
 logger.error("Exception while bootstrapping client after " + bootstrapTimeMs + " ms", e);  
 client.close();  
 throw Throwables.propagate(e);  
 }  
 long postBootstrap = System.nanoTime();  
  
 logger.info("Successfully created connection to {} after {} ms ({} ms spent in bootstraps)",  
 address, (postBootstrap - preConnect) / 1000000, (postBootstrap - preBootstrap) / 1000000);  
  
 return client;  
}

从代码清单3-13得知，真正创建TransportClient的步骤如下。

1）构建根引导程序Bootstrap并对其进行配置。

2）为根引导程序设置管道初始化回调函数，此回调函数将调用TransportContext的initializePipeline方法初始化Channel的pipeline。

3）使用根引导程序连接远程服务器，当连接成功对管道初始化时会回调初始化回调函数，将TransportClient和Channel对象分别设置到原子引用clientRef与channelRef中。

4）给TransportClient设置客户端引导程序，即设置TransportClientFactory中的Transport-ClientBootstrap列表。

5）返回此TransportClient对象。

[[1]](#_1_16) Spark将对Netty框架的使用细节都封装在NettyUtils工具类中，由于Netty的API使用不属于本书主要阐述的内容，故此放入附录G中，对Netty的使用感兴趣的读者可以选择阅读。

### 3.2.3 RPC服务端TransportServer

TransportServer是RPC框架的服务端，可提供高效、低级别的流服务。在说明图3-1中的记号②时提到过，TransportContext的createServer方法用于创建TransportServer，其实现如代码清单3-14所示。

代码清单3-14 创建RPC服务端

public TransportServer createServer(int port, List<TransportServerBootstrap> bootstraps) {  
 return new TransportServer(this, null, port, rpcHandler, bootstraps);  
}  
  
public TransportServer createServer(  
 String host, int port, List<TransportServerBootstrap> bootstraps) {  
 return new TransportServer(this, host, port, rpcHandler, bootstraps);  
}  
  
public TransportServer createServer(List<TransportServerBootstrap> bootstraps) {  
 return createServer(0, bootstraps);  
}  
  
public TransportServer createServer() {  
 return createServer(0, Lists.<TransportServerBootstrap>newArrayList());  
}

代码清单3-14中列出了4个名为createServer的重载方法，但是它们最终调用了TransportServer的构造器（见代码清单3-15）来创建TransportServer实例。

代码清单3-15 TransportServer的构造器

public TransportServer(  
 TransportContext context,  
 String hostToBind,  
 int portToBind,  
 RpcHandler appRpcHandler,  
 List<TransportServerBootstrap> bootstraps) {  
 this.context = context;  
 this.conf = context.getConf();  
 this.appRpcHandler = appRpcHandler;  
 this.bootstraps = Lists.newArrayList(Preconditions.checkNotNull(bootstraps));  
  
 try {  
 init(hostToBind, portToBind);  
 } catch (RuntimeException e) {  
 JavaUtils.closeQuietly(this);  
 throw e;  
 }  
}

TransportServer的构造器中的各个变量如下。

·context：参数传递的TransportContext的引用。

·conf：指TransportConf，这里通过调用TransportContext的getConf获取。

·appRpcHandler：RPC请求处理器RpcHandler。

·bootstraps：参数传递的TransportServerBootstrap列表。

TransportServer的构造器（见代码清单3-15）中调用了init方法，init方法用于对TransportServer进行初始化，如代码清单3-16所示。

代码清单3-16 TransportServer的初始化

private void init(String hostToBind, int portToBind) {  
 // 根据Netty的API文档，Netty服务端需同时创建bossGroup和workerGroup  
 IOMode ioMode = IOMode.valueOf(conf.ioMode());  
 EventLoopGroup bossGroup =  
 NettyUtils.createEventLoop(ioMode, conf.serverThreads(), conf.getModuleName() + "-server");  
 EventLoopGroup workerGroup = bossGroup;  
 // 创建一个汇集ByteBuf但对本地线程缓存禁用的分配器  
 PooledByteBufAllocator allocator = NettyUtils.createPooledByteBufAllocator(  
 conf.preferDirectBufs(), true /\* allowCache \*/, conf.serverThreads());  
 // 创建Netty的服务端根引导程序并对其进行配置  
 bootstrap = new ServerBootstrap()  
 .group(bossGroup, workerGroup)  
 .channel(NettyUtils.getServerChannelClass(ioMode))  
 .option(ChannelOption.ALLOCATOR, allocator)  
 .childOption(ChannelOption.ALLOCATOR, allocator);  
  
 if (conf.backLog() > 0) {  
 bootstrap.option(ChannelOption.SO\_BACKLOG, conf.backLog());  
 }  
 if (conf.receiveBuf() > 0) {  
 bootstrap.childOption(ChannelOption.SO\_RCVBUF, conf.receiveBuf());  
 }  
 if (conf.sendBuf() > 0) {  
 bootstrap.childOption(ChannelOption.SO\_SNDBUF, conf.sendBuf());  
 }  
 // 为根引导程序设置管道初始化回调函数  
 bootstrap.childHandler(new ChannelInitializer<SocketChannel>() {  
 @Override  
 protected void initChannel(SocketChannel ch) throws Exception {  
 RpcHandler rpcHandler = appRpcHandler;  
 for (TransportServerBootstrap bootstrap : bootstraps) {  
 rpcHandler = bootstrap.doBootstrap(ch, rpcHandler);  
 }  
 context.initializePipeline(ch, rpcHandler);  
 }  
 });  
 // 给根引导程序绑定Socket的监听端口  
 InetSocketAddress address = hostToBind == null ?  
 new InetSocketAddress(portToBind): new InetSocketAddress(hostToBind, portToBind);  
 channelFuture = bootstrap.bind(address);  
 channelFuture.syncUninterruptibly();  
  
 port = ((InetSocketAddress) channelFuture.channel().localAddress()).getPort();  
 logger.debug("Shuffle server started on port: {}", port);  
}

代码清单3-16中TransportServer初始化的步骤如下。

1）创建bossGroup和workerGroup。

小贴士： 根据Netty的API文档，Netty服务端需同时创建bossGroup和workerGroup。

2）创建一个汇集ByteBuf但对本地线程缓存禁用的分配器。

3）调用Netty的API创建Netty的服务端根引导程序并对其进行配置。

4）为根引导程序设置管道初始化回调函数，此回调函数首先设置TransportServer-Bootstrap到根引导程序中，然后调用TransportContext的initializePipeline方法初始化Channel的pipeline。

5）给根引导程序绑定Socket的监听端口，最后返回监听的端口。

Image00046.jpg 提示 代码清单3-16中使用了NettyUtils工具类的很多方法，在附录G中有对它们的详细介绍。EventLoopGroup、PooledByteBufAllocator、ServerBootstrap都是Netty提供的API，对于它们的更多介绍，请访问<http://netty.io/> 。

### 3.2.4 管道初始化

在代码清单3-13创建TransportClient和代码清单3-16对TransportServer初始化的实现中，都在管道初始化回调函数中调用了TransportContext的initializePipeline方法，initializePipeline方法（见代码清单3-17）将调用Netty的API对管道初始化。

代码清单3-17 管道初始化

public TransportChannelHandler initializePipeline(  
 SocketChannel channel,  
 RpcHandler channelRpcHandler) {  
 try {  
 TransportChannelHandler channelHandler = createChannelHandler(channel, channelRpcHandler);  
 channel.pipeline()  
 .addLast("encoder", ENCODER)  
 .addLast(TransportFrameDecoder.HANDLER\_NAME, NettyUtils.createFrame Decoder())  
 .addLast("decoder", DECODER)  
 .addLast("idleStateHandler", new IdleStateHandler(0, 0, conf.connection TimeoutMs() / 1000))  
 .addLast("handler", channelHandler);  
 return channelHandler;  
 } catch (RuntimeException e) {  
 logger.error("Error while initializing Netty pipeline", e);  
 throw e;  
 }  
}

根据代码清单3-17，initializePipeline方法的执行步骤如下。

1）调用createChannelHandler方法创建TransportChannelHandler，从createChannel-Handler的实现（见代码清单3-18）中可以看到，真正创建TransportClient是在这里发生的。通过TransportClient的构造过程看到RpcHandler与TransportClient毫无关系，Transport-Client只使用了TransportResponseHandler。TransportChannelHandler在服务端将代理Transport-RequestHandler对请求消息进行处理，并在客户端代理TransportResponseHandler对响应消息进行处理。

代码清单3-18 创建TransportChannelHandler

private TransportChannelHandler createChannelHandler(Channel channel, RpcHandler rpcHandler) {  
 TransportResponseHandler responseHandler = new TransportResponseHandler(channel);  
 TransportClient client = new TransportClient(channel, responseHandler);  
 TransportRequestHandler requestHandler = new TransportRequestHandler(channel, client,  
 rpcHandler);  
 return new TransportChannelHandler(client, responseHandler, requestHandler,  
 conf.connectionTimeoutMs(), closeIdleConnections);  
}

2）对管道进行设置，这里的ENCODER（即MessageEncoder）派生自Netty的ChannelOutboundHandler接口；DECODER（即MessageDecoder）、TransportChannelHandler及TransportFrameDecoder（由工具类NettyUtils的静态方法createFrameDecoder创建）派生自Netty的ChannelInboundHandler接口；IdleState Handler同时实现了ChannelOutbound-Handler和ChannelInboundHandler接口。根据Netty的API行为，通过addLast方法注册多个Handler时，ChannelInboundHandler按照注册的先后顺序执行，ChannelOutboundHandler按照注册的先后顺序逆序执行，因此在管道两端（无论是服务端还是客户端）处理请求和响应的流程如图3-3所示。

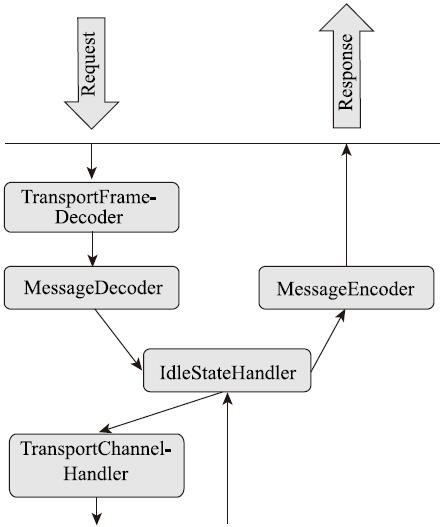


图3-3 管道处理请求和响应的流程图

### 3.2.5 TransportChannelHandler详解

TransportChannelHandler实现了Netty的ChannelInboundHandler [[1]](#_1_19) ，以便对Netty管道中的消息进行处理。图3-3中的这些Handler（除了MessageEncoder）由于都实现了ChannelInbound-Handler接口，作为自定义的ChannelInbound-Handler，所以都要重写channelRead方法。Netty框架使用工作链模式来对每个ChannelInboundHandler的实现类的channelRead方法进行链式调用。TransportChannelHandler实现的channelRead方法如代码清单3-19所示。

代码清单3-19 TransportChannelHandler的channelRead实现

@Override  
public void channelRead(ChannelHandlerContext ctx, Object request) throws Exception {  
 if (request instanceof RequestMessage) {  
 requestHandler.handle((RequestMessage) request);  
 } else if (request instanceof ResponseMessage) {  
 responseHandler.handle((ResponseMessage) request);  
 } else {  
 ctx.fireChannelRead(request);  
 }  
}

从代码清单3-19看到，当TransportChannelHandler读取的request是RequestMessage时，则将此消息的处理进一步交给TransportRequestHandler，当读取的request是Response-Message时，则将此消息的处理进一步交给TransportResponseHandler。

1.MessageHandler的继承体系

TransportRequestHandler与TransportResponseHandler都继承自抽象类MessageHandler，MessageHandler定义了子类的规范，详细定义如代码清单3-20所示。

代码清单3-20 MessageHandler规范

public abstract class MessageHandler<T extends Message> {  
 public abstract void handle(T message) throws Exception;  
 public abstract void channelActive();  
 public abstract void exceptionCaught(Throwable cause);  
 public abstract void channelInactive();  
}

MessageHandler中定义的各个方法的作用分别如下。

·handle：用于对接收到的单个消息进行处理。

·channelActive：当channel激活时调用。

·exceptionCaught：当捕获到channel发生异常时调用。

·channelInactive：当channel非激活时调用。

Spark中MessageHandler类的继承体系如图3-4所示。

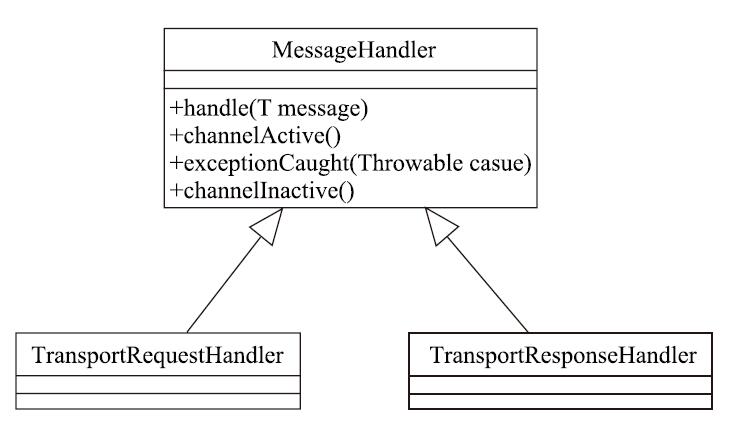


图3-4 MessageHandler类的继承体系

2.Message的继承体系

根据代码清单3-20我们知道，MessageHandler同时也是一个Java泛型类，其子类能处理的消息都派生自接口Message。Message的定义如代码清单3-21所示。

代码清单3-21 Message的定义

public interface Message extends Encodable {  
 Type type();  
 ManagedBuffer body();  
 boolean isBodyInFrame();

Message中定义的三个接口方法的作用分别如下。

·type：返回消息的类型。

·body：返回消息中可选的内容体。

·isBodyInFrame：用于判断消息的主体是否包含在消息的同一帧中。

Message接口继承了Encodable接口，Encodable的定义如代码清单3-22所示。

代码清单3-22 Encodable的定义

public interface Encodable {  
 int encodedLength();  
 void encode(ByteBuf buf);  
}

实现Encodable接口的类将可以转换到一个ByteBuf中，多个对象将被存储到预先分配的单个ByteBuf，所以这里的encodedLength用于返回转换的对象数量。下面一起来看看Message的类继承体系，如图3-5所示。

从图3-5看到，最终的消息实现类都直接或间接地实现了RequestMessage或Response-Message接口，其中RequestMessage的具体实现有4种，分别如下。

·ChunkFetchRequest：请求获取流的单个块的序列。

·RpcRequest：此消息类型由远程的RPC服务端进行处理，是一种需要服务端向客户端回复的RPC请求信息类型。

·OneWayMessage：此消息也需要由远程的RPC服务端进行处理，与RpcRequest不同的是，不需要服务端向客户端回复。

·StreamRequest：此消息表示向远程的服务发起请求，以获取流式数据。

由于OneWayMessage不需要响应，所以ResponseMessage对于成功或失败状态的实现各有3种，分别如下。

·ChunkFetchSuccess：处理ChunkFetchRequest成功后返回的消息。

·ChunkFetchFailure：处理ChunkFetchRequest失败后返回的消息。

·RpcResponse：处理RpcRequest成功后返回的消息。

·RpcFailure：处理RpcRequest失败后返回的消息。

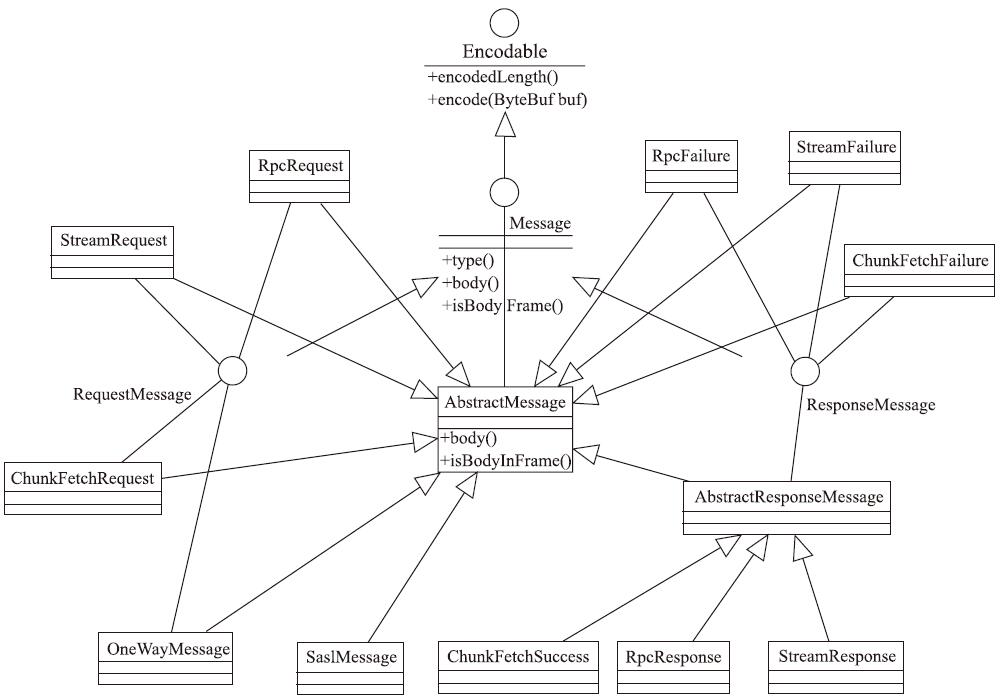


图3-5 Message的类继承体系

·StreamResponse：处理StreamRequest成功后返回的消息。

·StreamFailure：处理StreamRequest失败后返回的消息。

3.ManagedBuffer的继承体系

回头再看看代码清单3-21中对body接口的定义，可以看到，其返回内容体的类型为ManagedBuffer。ManagedBuffer提供了由字节构成数据的不可变视图（也就是说，ManagedBuffer并不存储数据，也不是数据的实际来源，这同关系型数据库的视图类似）。我们先来看看抽象类ManagedBuffer中对行为的定义，如代码清单3-23所示。

代码清单3-23 ManagedBuffer的定义

public abstract class ManagedBuffer {  
 public abstract long size();  
 public abstract ByteBuffer nioByteBuffer() throws IOException;  
 public abstract InputStream createInputStream() throws IOException;  
 public abstract ManagedBuffer retain();  
 public abstract ManagedBuffer release();  
 public abstract Object convertToNetty() throws IOException;  
}

ManagedBuffer中定义了6个方法，分别如下。

·size：返回数据的字节数。

·nioByteBuffer：将数据按照NIO的ByteBuffer类型返回。

·createInputStream：将数据按照InputStream返回。

·retain：当有新的使用者使用此视图时，增加引用此视图的引用数。

·release：当有使用者不再使用此视图时，减少引用此视图的引用数。当引用数为0时释放缓冲区。

·convertToNetty：将缓冲区的数据转换为Netty的对象，用来将数据写到外部。此方法返回的数据类型要么是io.netty.buffer.ByteBuf，要么是io.netty.channel.File-Region。

ManagedBuffer的具体实现有很多，我们可以通过图3-6来了解。

图3-6中列出了ManagedBuffer的5个实现类，其中TestManagedBuffer和Recording-ManagedBuffer用于测试。NettyManagedBuffer中的缓冲为io.netty.buffer.ByteBuf，NioManaged-Buffer中的缓冲为java.nio.ByteBuffer。NettyManagedBuffer和NioManagedBuffer的实现都非常简单，留给读者自行阅读。本节挑选FileSegmentManagedBuffer作为ManagedBuffer具体实现的例子进行介绍。

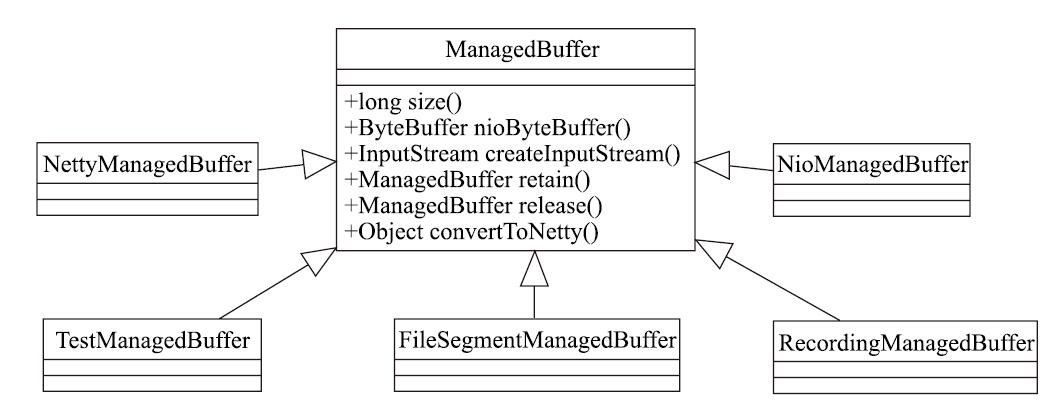


图3-6 ManagedBuffer的继承体系

FileSegmentManagedBuffer的作用为获取一个文件中的一段，它一共有4个由final修饰的属性，全部都通过FileSegmentManagedBuffer的构造器传入属性值，这4个属性分别如下。

·conf：也就是TransportConf。

·file：所要读取的文件。

·offset：所要读取文件的偏移量。

·length：所要读取的长度。

下面将逐个介绍FileSegmentManagedBuffer对于ManagedBuffer的实现。

·NIO方式读取文件。FileSegmentManagedBuffer实现的nioByteBuffer方法如代码清单3-24所示。

代码清单3-24 nioByteBuffer方法的实现

@Override  
public ByteBuffer nioByteBuffer() throws IOException {  
 FileChannel channel = null;  
 try {  
 channel = new RandomAccessFile(file, "r").getChannel();  
 if (length < conf.memoryMapBytes()) {  
 ByteBuffer buf = ByteBuffer.allocate((int) length);  
 channel.position(offset);  
 while (buf.remaining() != 0) {  
 if (channel.read(buf) == -1) {  
 throw new IOException(String.format("Reached EOF before filling buffer\n" +  
 "offset=%s\nfile=%s\nbuf.remaining=%s",  
 offset, file.getAbsoluteFile(), buf.remaining()));  
 }  
 }  
 buf.flip();  
 return buf;  
 } else {  
 return channel.map(FileChannel.MapMode.READ\_ONLY, offset, length);  
 }  
 } catch (IOException e) {  
 try {  
 if (channel != null) {  
 long size = channel.size();  
 throw new IOException("Error in reading " + this + " (actual file length " + size + ")",  
 e);  
 }  
 } catch (IOException ignored) {  
 // ignore  
 }  
 throw new IOException("Error in opening " + this, e);  
 } finally {  
 JavaUtils.closeQuietly(channel);  
 }  
}

nioByteBuffer的实现还是很简单的，主要利用RandomAccessFile获取FileChannel，然后使用java.nio.ByteBuffer和FileChannel的API将数据写入缓冲区java.nio.ByteBuffer中。

·文件流方式读取文件。FileSegmentManagedBuffer实现的createInputStream方法如代码清单3-25所示。

代码清单3-25 createInputStream的实现

@Override  
public InputStream createInputStream() throws IOException {  
 FileInputStream is = null;  
 try {  
 is = new FileInputStream(file);  
 ByteStreams.skipFully(is, offset);  
 return new LimitedInputStream(is, length);  
 } catch (IOException e) {  
 try {  
 if (is != null) {  
 long size = file.length();  
 throw new IOException("Error in reading " + this + " (actual file length " + size + ")",  
 e);  
 }  
 } catch (IOException ignored) {  
 // ignore  
 } finally {  
 JavaUtils.closeQuietly(is);  
 }  
 throw new IOException("Error in opening " + this, e);  
 } catch (RuntimeException e) {  
 JavaUtils.closeQuietly(is);  
 throw e;  
 }  
}

createInputStream的实现还是很简单的，这里不过多介绍。

·将数据转换为Netty对象。FileSegmentManagedBuffer实现的convertToNetty方法如代码清单3-26所示。

代码清单3-26 convertToNetty的实现

@Override  
public Object convertToNetty() throws IOException {  
 if (conf.lazyFileDescriptor()) {  
 return new DefaultFileRegion(file, offset, length);  
 } else {  
 FileChannel fileChannel = new FileInputStream(file).getChannel();  
 return new DefaultFileRegion(fileChannel, offset, length);  
 }  
}

·其他方法的实现。其他方法由于实现非常简单，所以这里就不一一列出了，感兴趣的读者可以自行查阅。

[[1]](#_1_18) ChannelInboundHandler接口的实现及原理不属于本书要分析的内容，感兴趣的同学可以阅读Netty的官方文档或者研究Netty 的源码。

### 3.2.6 服务端RpcHandler详解

由于TransportRequestHandler实际是把请求消息交给RpcHandler进行进一步处理，所以这里对RpcHandler首先做个介绍。RpcHandler是一个抽象类，定义了一些RPC处理器的规范，其主要实现如代码清单3-27所示。

代码清单3-27 RpcHandler的实现

public abstract class RpcHandler {  
private static final RpcResponseCallback ONE\_WAY\_CALLBACK = new OneWayRpc-Callback();  
public abstract void receive(  
 TransportClient client,  
 ByteBuffer message,  
 RpcResponseCallback callback);  
public abstract StreamManager getStreamManager();  
public void receive(TransportClient client, ByteBuffer message) {  
 receive(client, message, ONE\_WAY\_CALLBACK);  
}  
public void channelActive(TransportClient client) { }  
public void channelInactive(TransportClient client) { }  
public void exceptionCaught(Throwable cause, TransportClient client) { }  
private static class OneWayRpcCallback implements RpcResponseCallback {  
 private static final Logger logger = LoggerFactory.getLogger(OneWayRpcCallback.class);  
 @Override  
 public void onSuccess(ByteBuffer response) {  
 logger.warn("Response provided for one-way RPC.");  
 }  
 @Override  
 public void onFailure(Throwable e) {  
 logger.error("Error response provided for one-way RPC.", e);  
 }  
 }  
}

代码清单3-27中RpcHandler的各个方法的作用如下。

·receive：这是一个抽象方法，用来接收单一的RPC消息，具体处理逻辑需要子类去实现。receive接收3个参数，分别是TransportClient、ByteBuffer和RpcResponse-Callback。RpcResponseCallback用于对请求处理结束后进行回调，无论处理结果是成功还是失败，RpcResponseCallback都会被调用一次。RpcResponseCallback的接口定义如下。

public interface RpcResponseCallback {  
 void onSuccess(ByteBuffer response);  
 void onFailure(Throwable e);  
}

·重载的receive：只接收TransportClient和ByteBuffer两个参数，RpcResponse-Callback为默认的ONE\_WAY\_CALLBACK，其类型为OneWayRpcCallback，从代码清单3-27中OneWayRpcCallback的实现可以看出，其onSuccess和onFailure只是打印日志，并没有针对客户端做回复处理。

·channelActive：当与给定客户端相关联的channel处于活动状态时调用。

·channelInactive：当与给定客户端相关联的channel处于非活动状态时调用。

·exceptionCaught：当channel产生异常时调用。

·getStreamManager：获取StreamManager，StreamManager可以从流中获取单个的块，因此它也包含着当前正在被TransportClient获取的流的状态。

介绍完RpcHandler，现在回到TransportRequestHandler的处理过程。Transport-RequestHandler处理以上4种RequestMessage的实现如代码清单3-28所示。

代码清单3-28 TransportRequestHandler的handle方法

@Override  
public void handle(RequestMessage request) {  
 if (request instanceof ChunkFetchRequest) {  
 processFetchRequest((ChunkFetchRequest) request);  
 } else if (request instanceof RpcRequest) {  
 processRpcRequest((RpcRequest) request);  
 } else if (request instanceof OneWayMessage) {  
 processOneWayMessage((OneWayMessage) request);  
 } else if (request instanceof StreamRequest) {  
 processStreamRequest((StreamRequest) request);  
 } else {  
 throw new IllegalArgumentException("Unknown request type: " + request);  
 }  
}

结合代码清单3-28，下面详细分析这4种类型请求的处理过程。

1.处理块获取请求

processFetchRequest方法用于处理ChunkFetchRequest类型的消息，其实现如代码清单3-29所示。

代码清单3-29 processFetchRequest的实现

private void processFetchRequest(final ChunkFetchRequest req) {  
 if (logger.isTraceEnabled()) {  
 logger.trace("Received req from {} to fetch block {}", getRemoteAddress (channel),  
 req.streamChunkId);  
 }  
 ManagedBuffer buf;  
 try {  
 streamManager.checkAuthorization(reverseClient, req.streamChunkId.streamId);  
 streamManager.registerChannel(channel, req.streamChunkId.streamId);  
 buf = streamManager.getChunk(req.streamChunkId.streamId, req.streamChunkId.chunkIndex);  
 } catch (Exception e) {  
 logger.error(String.format("Error opening block %s for request from %s",  
 req.streamChunkId, getRemoteAddress(channel)), e);  
 respond(new ChunkFetchFailure(req.streamChunkId, Throwables.getStackTraceAsString(e)));  
 return;  
 }  
 respond(new ChunkFetchSuccess(req.streamChunkId, buf));  
}

代码清单3-29中的streamManager是通过调用RpcHandler的getStreamManager方法获取的StreamManager。processFetchRequest的处理都依托于RpcHandler的Stream-Manager，其处理步骤如下。

1）调用StreamManager的checkAuthorization方法，校验客户端是否有权限从给定的流中读取。

2）调用StreamManager的registerChannel方法，将一个流和一条（只能是一条）客户端的TCP连接关联起来，这可以保证对于单个的流只会有一个客户端读取。流关闭之后就永远不能够重用了。

3）调用StreamManager的getChunk方法，获取单个的块（块被封装为Managed-Buffer）。由于单个的流只能与单个的TCP连接相关联，因此getChunk方法不能为了某个特殊的流而并行调用。

4）将ManagedBuffer和流的块Id封装为ChunkFetchSuccess后，调用respond方法返回给客户端。

有关StreamManager的具体实现，读者可以参考5.3.5节介绍的NettyStreamManager和6.9.2节介绍的NettyBlockRpcServer中的OneForOneStreamManager。

2.处理RPC请求

processRpcRequest方法用于处理RpcRequest类型的消息，其实现如代码清单3-30所示。

代码清单3-30 processRpcRequest的实现

private void processRpcRequest(final RpcRequest req) {  
 try {  
 rpcHandler.receive(reverseClient, req.body().nioByteBuffer(), new RpcResponse Callback() {  
 @Override  
 public void onSuccess(ByteBuffer response) {  
 respond(new RpcResponse(req.requestId, new NioManagedBuffer(response)));  
 }  
  
 @Override  
 public void onFailure(Throwable e) {  
 respond(new RpcFailure(req.requestId, Throwables.getStackTraceAsString(e)));  
 }  
 });  
 } catch (Exception e) {  
 logger.error("Error while invoking RpcHandler#receive() on RPC id " + req.requestId, e);  
 respond(new RpcFailure(req.requestId, Throwables.getStackTraceAsString(e)));  
 } finally {  
 req.body().release();  
 }  
}

代码清单3-30中将RpcRequest消息的内容体、发送消息的客户端及一个RpcResponse-Callback类型的匿名内部类作为参数传递给了RpcHandler的receive方法。这就是说真正用于处理RpcRequest消息的是RpcHandler，而非TransportRequestHandler。由于RpcHandler是抽象类（见代码清单3-27），其receive方法也是抽象方法，所以具体的操作将由RpcHandler的实现了receive方法的子类来完成。所有继承RpcHandler的子类都需要在其receive方法的具体实现中回调RpcResponseCallback的onSuccess（处理成功时）或者onFailure（处理失败时）方法。从RpcResponseCallback的实现来看，无论处理结果成功还是失败，都将调用respond方法对客户端进行响应。

3.处理流请求

processStreamRequest方法用于处理StreamRequest类型的消息，其实现如代码清单3-31所示。

代码清单3-31 processStreamRequest的实现

private void processStreamRequest(final StreamRequest req) {  
 ManagedBuffer buf;  
 try {  
 buf = streamManager.openStream(req.streamId);// 将获取到的流数据封装为ManagedBuffer  
 } catch (Exception e) {  
 logger.error(String.format(  
 "Error opening stream %s for request from %s", req.streamId, getRemoteAddress(channel)), e);  
 respond(new StreamFailure(req.streamId, Throwables.getStackTrace-AsString(e)));  
 return;  
 }  
 if (buf != null) {  
 respond(new StreamResponse(req.streamId, buf.size(), buf));  
 } else {  
 respond(new StreamFailure(req.streamId, String.format(  
 "Stream '%s' was not found.", req.streamId)));  
 }  
}

代码清单3-31中也使用了RpcHandler的StreamManager，其处理步骤如下。

1）调用StreamManager的openStream方法将获取到的流数据封装为ManagedBuffer。

2）当成功或失败时调用respond方法向客户端响应。

4.处理无需回复的RPC请求

processOneWayMessage方法用于处理StreamRequest类型的消息，其实现如代码清单3-32所示。

代码清单3-32 processOneWayMessage的实现

private void processOneWayMessage(OneWayMessage req) {  
 try {  
 rpcHandler.receive(reverseClient, req.body().nioByteBuffer());  
 } catch (Exception e) {  
 logger.error("Error while invoking RpcHandler#receive() for one-way message.", e);  
 } finally {  
 req.body().release();  
 }  
}

processOneWayMessage方法的实现与processRpcRequest非常相似，区别在于processOneWayMessage调用了代码清单3-27中ONE\_WAY\_CALLBACK的receive方法，因而processOneWayMessage在处理完RPC请求后不会对客户端作出响应。

从以上4种处理的分析可以看出，最终的处理都由RpcHandler及其内部组件完成。除了OneWayMessage的消息外，其余三种消息都是最终调用respond方法响应客户端，其实现如代码清单3-33所示。

代码清单3-33 respond的实现

private void respond(final Encodable result) {  
 final SocketAddress remoteAddress = channel.remoteAddress();  
 channel.writeAndFlush(result).addListener(  
 new ChannelFutureListener() {  
 @Override  
 public void operationComplete(ChannelFuture future) throws Exception {  
 if (future.isSuccess()) {  
 logger.trace("Sent result {} to client {}", result, remoteAddress);  
 } else {  
 logger.error(String.format("Error sending result %s to %s; closing connection",  
 result, remoteAddress), future.cause());  
 channel.close();  
 }  
 }  
 }  
 );  
}

可以看到respond方法中实际调用了Channel的writeAndFlush方法 [[1]](#_1_21) 来响应客户端。

[[1]](#_1_20) Channel的writeAndFlush方法涉及Netty的实现细节及原理，这并不是本书所要阐述的内容，有兴趣的读者可以访问Netty 官网：http://netty.io 获取更多信息。

### 3.2.7 服务端引导程序TransportServerBootstrap

TransportServer的构造器（见代码清单3-15）中的bootstraps是TransportServerBootstrap的列表。接口TransportServerBootstrap定义了服务端引导程序的规范，服务端引导程序旨在当客户端与服务端建立连接之后，在服务端持有的客户端管道上执行的引导程序。TransportServerBootstrap的定义如代码清单3-34所示。

代码清单3-34 TransportServerBootstrap的定义

public interface TransportServerBootstrap {  
 RpcHandler doBootstrap(Channel channel, RpcHandler rpcHandler);  
}

TransportServerBootstrap的doBootstrap方法将对服务端的RpcHandler进行代理，接收客户端的请求。TransportServerBootstrap有SaslServerBootstrap和EncryptionChecker-Bootstrap两个实现类。为了更清楚地说明TransportServerBootstrap的意义，我们以SaslServer-Bootstrap为例，来讲解其实现（见代码清单3-35）。

代码清单3-35 SaslServerBootstrap的doBootstrap实现

public RpcHandler doBootstrap(Channel channel, RpcHandler rpcHandler) {  
 return new SaslRpcHandler(conf, channel, rpcHandler, secretKeyHolder);  
}

根据代码清单3-35我们知道，SaslServerBootstrap的doBootstrap方法实际创建了SaslRpcHandler，SaslRpcHandler负责对管道进行SASL（Simple Authentication and Security Layer）加密。SaslRpcHandler本身也继承了RpcHandler，所以我们重点来看其receive方法的实现，如代码清单3-36所示。

代码清单3-36 SaslRpcHandler的receive方法

@Override  
public void receive(TransportClient client, ByteBuffer message, RpcResponseCallback callback) {  
 if (isComplete) {  
 // 将消息传递给SaslRpcHandler所代理的下游RpcHandler并返回  
 delegate.receive(client, message, callback);  
 return;  
 }  
  
 ByteBuf nettyBuf = Unpooled.wrappedBuffer(message);  
 SaslMessage saslMessage;  
 try {  
 saslMessage = SaslMessage.decode(nettyBuf);// 对客户端发送的消息进行SASL解密  
 } finally {  
 nettyBuf.release();  
 }  
  
 if (saslServer == null) {  
 // 如果saslServer还未创建，则需要创建SparkSaslServer  
 client.setClientId(saslMessage.appId);  
 saslServer = new SparkSaslServer(saslMessage.appId, secretKeyHolder,  
 conf.saslServerAlwaysEncrypt());  
 }  
 byte[] response;  
 try {  
 response = saslServer.response(JavaUtils.bufferToArray(// 使用saslServer处理已解密的消息  
 saslMessage.body().nioByteBuffer()));  
 } catch (IOException ioe) {  
 throw new RuntimeException(ioe);  
 }  
 callback.onSuccess(ByteBuffer.wrap(response));  
 if (saslServer.isComplete()) {  
 logger.debug("SASL authentication successful for channel {}", client);  
 isComplete = true;// SASL认证交换已经完成  
 if (SparkSaslServer.QOP\_AUTH\_CONF.equals(saslServer.getNegotiatedProperty(Sasl.QOP))) {  
 logger.debug("Enabling encryption for channel {}", client);  
 // 对管道进行SASL加密  
 SaslEncryption.addToChannel(channel, saslServer, conf.maxSaslEncrypted-BlockSize());  
 saslServer = null;  
 } else {  
 saslServer.dispose();  
 saslServer = null;  
 }  
 }  
}

根据代码清单3-36，SaslRpcHandler处理客户端消息的步骤如下。

1）如果SASL认证交换已经完成（isComplete等于true），则将消息传递给SaslRpc Handler所代理的下游RpcHandler并返回。

2）如果SASL认证交换未完成（isComplete等于false），则对客户端发送的消息进行SASL解密。

3）如果saslServer还未创建，则需要创建SparkSaslServer。当SaslRpcHandler接收到客户端的第一条消息时会做此操作。

4）使用saslServer处理已解密的消息，并将处理结果通过RpcResponseCallback的回调方法返回给客户端。

5）如果SASL认证交换已经完成，则将isComplete置为true。

6）对管道进行SASL加密。

SaslServerBootstrap是通过SaslRpcHandler对下游RpcHandler进行代理的一种Tran sportServerBootstrap。EncryptionCheckerBootstrap是另一种TransportServerBootstrap的实现，它通过将自身加入Netty的管道中实现引导，EncryptionCheckerBootstrap的doBootstrap方法的实现如代码清单3-37所示。

代码清单3-37 EncryptionCheckerBootstrap的doBootstrap实现

@Override  
public RpcHandler doBootstrap(Channel channel, RpcHandler rpcHandler) {  
 channel.pipeline().addFirst("encryptionChecker", this);  
 return rpcHandler;  
}

在详细介绍了TransportChannelHandler之后，我们就可以对图3-3进行扩展，把Transport RequestHandler、TransportServerBootstrap及RpcHandler的处理流程增加进来，如图3-7所示。

有读者可能会问，图3-7中并未见TransportServerBootstrap的身影。根据对Transport-ServerBootstrap的两种实现的举例，我们知道TransportServerBootstrap将可能存在于图中任何两个组件的箭头连线中间，起到引导、包装、代理的作用。

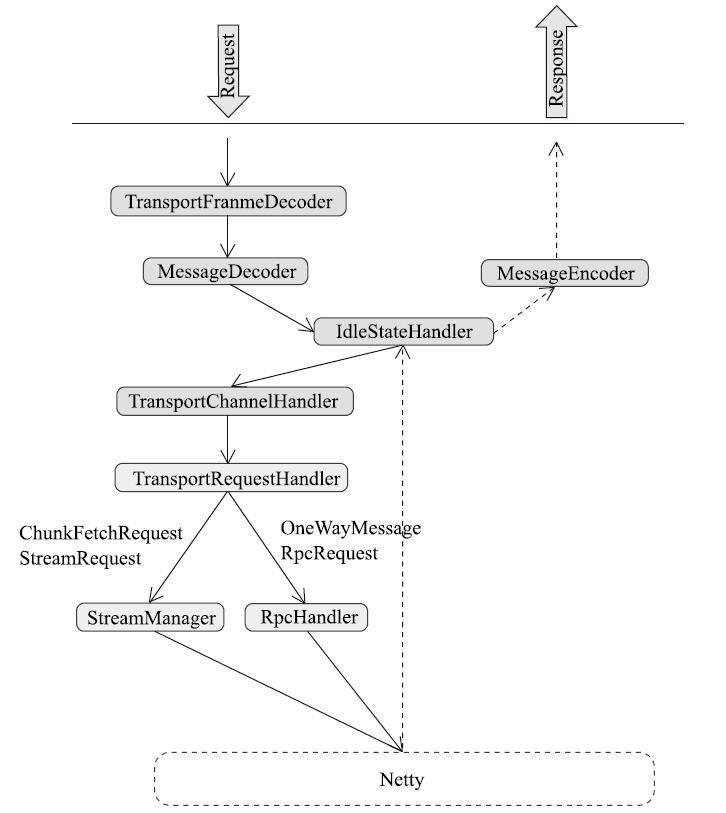


图3-7 RPC框架服务端处理请求、响应流程图

### 3.2.8 客户端TransportClient详解

在介绍完服务端RpcHandler对请求消息的处理之后，现在来看看客户端发送RPC请求的原理。我们在分析代码清单3-18中的createChannelHandler方法时，看到调用了TransportClient的构造器（见代码清单3-38），其中TransportResponseHandler的引用将赋给handler属性。

代码清单3-38 TransportClient的构造器

public TransportClient(Channel channel, TransportResponseHandler handler) {  
 this.channel = Preconditions.checkNotNull(channel);  
 this.handler = Preconditions.checkNotNull(handler);  
 this.timedOut = false;  
}

TransportClient一共有5个方法用于发送请求，分别如下。

1）fetchChunk：从远端协商好的流中请求单个块。

2）stream：使用流的ID，从远端获取流数据。

3）sendRpc：向服务端发送RPC的请求，通过At least Once Delivery原则保证请求不会丢失。

4）sendRpcSync：向服务端发送异步的RPC的请求，并根据指定的超时时间等待响应。

5）send：向服务端发送RPC的请求，但是并不期望能获取响应，因而不能保证投递的可靠性。

本节只选择最常用的sendRpc和fetchChunk进行分析，其余实现都可以触类旁通。

1.发送RPC请求

sendRpc方法的实现如代码清单3-39所示。

代码清单3-39 sendRpc的实现

public long sendRpc(ByteBuffer message, final RpcResponseCallback callback) {  
 final long startTime = System.currentTimeMillis();  
 if (logger.isTraceEnabled()) {  
 logger.trace("Sending RPC to {}", getRemoteAddress(channel));  
 }  
 // 使用UUID生成请求主键requestId  
 final long requestId = Math.abs(UUID.randomUUID().getLeastSignificantBits());  
 handler.addRpcRequest(requestId, callback);// 添加requestId与RpcResponseCallback的引用之间的关系  
 // 发送RPC请求  
 channel.writeAndFlush(new RpcRequest(requestId, new NioManagedBuffer　(message))).addListener(  
 new ChannelFutureListener() {  
 @Override  
 public void operationComplete(ChannelFuture future) throws Exception {  
 if (future.isSuccess()) {  
 long timeTaken = System.currentTimeMillis() - startTime;  
 if (logger.isTraceEnabled()) {  
 logger.trace("Sending request {} to {} took {} ms", requestId,  
 getRemoteAddress(channel), timeTaken);  
 }  
 } else {  
 String errorMsg = String.format("Failed to send RPC %s to %s: %s", requestId,  
 getRemoteAddress(channel), future.cause());  
 logger.error(errorMsg, future.cause());  
 handler.removeRpcRequest(requestId);  
 channel.close();  
 try {  
 callback.onFailure(new IOException(errorMsg, future.cause()));  
 } catch (Exception e) {  
 logger.error("Uncaught exception in RPC response callback handler!", e);  
 }  
 }  
 }  
 });  
 return requestId;  
}

结合代码清单3-39，我们知道sendRpc方法的实现步骤如下。

1）使用UUID生成请求主键requestId。

2）调用addRpcRequest向handler（特别提醒下读者，这里的handler不是RpcHandler，而是通过TransportClient构造器传入的TransportResponseHandler）添加requestId与回调类RpcResponseCallback的引用之间的关系。TransportResponseHandler的addRpcRequest方法（见代码清单3-40）将更新最后一次请求的时间为当前系统时间，然后将requestId与RpcResponseCallback之间的映射加入到outstandingRpcs缓存中。outstandingRpcs专门用于缓存发出的RPC请求信息。

代码清单3-40 添加RPC请求到缓存

public void addRpcRequest(long requestId, RpcResponseCallback callback) {  
 updateTimeOfLastRequest();  
 outstandingRpcs.put(requestId, callback);  
}

3）调用Channel的writeAndFlush方法将RPC请求发送出去，这和在代码清单3-33中服务端调用的respond方法响应客户端的一样，都是使用Channel的writeAndFlush方法。当发送成功或者失败时会回调ChannelFutureListener的operationComplete方法。如果发送成功，那么只会打印requestId、远端地址及花费时间的日志，如果发送失败，除了打印错误日志外，还要调用TransportResponseHandler的removeRpcRequest方法（见代码清单3-41），将此次请求从outstandingRpcs缓存中移除。

代码清单3-41 从缓存中删除RPC请求

public void removeRpcRequest(long requestId) {  
 outstandingRpcs.remove(requestId);  
}

请求发送成功后，客户端将等待接收服务端的响应。根据图3-3，返回的消息也会传递给TransportChannelHandler的channelRead方法（见代码清单3-19），根据之前的分析，消息的分析将最后交给TransportResponseHandler的handle方法来处理。TransportResponseHandler的handle方法分别对图3-5中的6种ResponseMessage进行处理，由于服务端使用processRpcRequest方法（见代码清单3-30）处理RpcRequest类型的消息后，返回给客户端的消息为RpcResponse或RpcFailure，所以我们来看看客户端的TransportResponseHandler的handle方法如何处理RpcResponse和RpcFailure，如代码清单3-42所示。

代码清单3-42 RpcResponse和RpcFailure消息的处理

} else if (message instanceof RpcResponse) {  
 RpcResponse resp = (RpcResponse) message;  
 RpcResponseCallback listener = outstandingRpcs.get(resp.requestId);// 获取RpcResponseCallback  
 if (listener == null) {  
 logger.warn("Ignoring response for RPC {} from {} ({} bytes) since it is not outstanding",  
 resp.requestId, getRemoteAddress(channel), resp.body().size());  
 } else {  
 outstandingRpcs.remove(resp.requestId);  
 try {  
 listener.onSuccess(resp.body().nioByteBuffer());  
 } finally {  
 resp.body().release();  
 }  
 }  
} else if (message instanceof RpcFailure) {  
 RpcFailure resp = (RpcFailure) message;  
 RpcResponseCallback listener = outstandingRpcs.get(resp.requestId); // 获取RpcResponseCallback  
 if (listener == null) {  
 logger.warn("Ignoring response for RPC {} from {} ({}) since it is not outstanding",  
 resp.requestId, getRemoteAddress(channel), resp.errorString);  
 } else {  
 outstandingRpcs.remove(resp.requestId);  
 listener.onFailure(new RuntimeException(resp.errorString));  
 }

从代码清单3-42看到，处理RpcResponse的逻辑如下。

1）使用RpcResponse对应的RpcRequest的主键requestId，从outstandingRpcs缓存中获取注册的RpcResponseCallback，此处的RpcResponseCallback即为代码清单3-39中传递给sendRpc方法的RpcResponseCallback。

2）移除outstandingRpcs缓存中requestId和RpcResponseCallback的注册信息。

3）调用RpcResponseCallback的onSuccess方法，处理成功响应后的具体逻辑。这里的RpcResponseCallback需要在各个使用TransportClient的sendRpc方法的场景中分别实现。

4）释放RpcResponse的body，回收资源。

处理RpcFailure的逻辑如下。

1）使用RpcFailure对应的RpcRequest的主键requestId，从outstandingRpcs缓存中获取注册的RpcResponseCallback，此处的RpcResponseCallback即为代码清单3-39中传递给sendRpc方法的RpcResponseCallback。

2）移除outstandingRpcs缓存中requestId和RpcResponseCallback的注册信息。

3）调用RpcResponseCallback的onFailure方法，处理失败响应后的具体逻辑。这里的RpcResponseCallback需要在使用TransportClient的sendRpc方法时指定或实现。

2.发送获取块请求

fetchChunk的实现如代码清单3-43所示。

代码清单3-43 fetchChunk的实现

public void fetchChunk(  
 long streamId,  
 final int chunkIndex,  
 final ChunkReceivedCallback callback) {  
 final long startTime = System.currentTimeMillis();  
 if (logger.isDebugEnabled()) {  
 logger.debug("Sending fetch chunk request {} to {}", chunkIndex, getRemoteAddress (channel));  
 }  
 final StreamChunkId streamChunkId = new StreamChunkId(streamId, chunkIndex);// 创建StreamChunkId  
 // 添加StreamChunkId与ChunkReceivedCallback之间的对应关系  
 handler.addFetchRequest(streamChunkId, callback);  
 // 发送块请求  
 channel.writeAndFlush(new ChunkFetchRequest(streamChunkId)).addListener(  
 new ChannelFutureListener() {  
 @Override  
 public void operationComplete(ChannelFuture future) throws Exception {  
 if (future.isSuccess()) {  
 long timeTaken = System.currentTimeMillis() - startTime;  
 if (logger.isTraceEnabled()) {  
 logger.trace("Sending request {} to {} took {} ms", streamChunkId,  
 getRemoteAddress(channel), timeTaken);  
 }  
 } else {  
 String errorMsg = String.format("Failed to send request %s to %s: %s", streamChunkId,  
 getRemoteAddress(channel), future.cause());  
 logger.error(errorMsg, future.cause());  
 handler.removeFetchRequest(streamChunkId);  
 channel.close();  
 try {  
 callback.onFailure(chunkIndex, new IOException(errorMsg, future.cause()));  
 } catch (Exception e) {  
 logger.error("Uncaught exception in RPC response callback handler!", e);  
 }  
 }  
 }  
 });  
}

结合代码清单3-43，我们知道fetchChunk方法的实现步骤如下。

1）使用流的标记streamId和块的索引chunkIndex创建StreamChunkId。

2）调用addFetchRequest向handler（特别提醒下读者，这里的handler不是RpcHandler，而是通过TransportClient构造器传入的TransportResponseHandler）添加StreamChunkId与回调类ChunkReceivedCallback的引用之间的关系。TransportResponseHandler的addFetchRequest方法（见代码清单3-44）将更新最后一次请求的时间为当前系统时间，然后将StreamChunkId与ChunkReceivedCallback之间的映射加入到outstandingFetches缓存中。outstandingFetches专门用于缓存发出的块请求信息。

代码清单3-44 添加块请求到缓存

public void addFetchRequest(StreamChunkId streamChunkId, ChunkReceivedCallback callback) {  
 updateTimeOfLastRequest();  
 outstandingFetches.put(streamChunkId, callback);  
}

3）调用Channel的writeAndFlush方法将块请求发送出去，这和在代码清单3-33中服务端调用的respond方法响应客户端的一样，都是使用Channel的writeAndFlush方法。当发送成功或者失败时会回调ChannelFutureListener的operationComplete方法。如果发送成功，那么只会打印StreamChunkId、远端地址及花费时间的日志，如果发送失败，除了打印错误日志外，还要调用TransportResponseHandler的removeFetchRequest方法（见代码清单3-45），将此次请求从outstandingFetches缓存中移除。

代码清单3-45 从缓存中删除RPC请求

public void removeRpcRequest(long requestId) {  
 outstandingRpcs.remove(requestId);  
}

请求发送成功后，客户端将等待接收服务端的响应。根据图3-3，返回的消息也会传递给TransportChannelHandler的channelRead方法（见代码清单3-19），根据之前的分析，消息的分析将最后交给TransportResponseHandler的handle方法来处理。TransportResponseHandler的handle方法分别对图3-5中的6种处理结果进行处理，由于服务端使用processFetchRequest方法（见代码清单3-29）处理ChunkFetchRequest类型的消息后返回给客户端的消息为ChunkFetchSuccess或ChunkFetchFailure，所以我们来看看客户端的TransportResponseHandler的handle方法如何处理ChunkFetchSuccess和ChunkFetchFailure，如代码清单3-46所示。

代码清单3-46 ChunkFetchSuccess和ChunkFetchFailure消息的处理

if (message instanceof ChunkFetchSuccess) {  
 ChunkFetchSuccess resp = (ChunkFetchSuccess) message;  
 ChunkReceivedCallback listener = outstandingFetches.get(resp.streamChunkId);  
 if (listener == null) {  
 logger.warn("Ignoring response for block {} from {} since it is not outstanding",  
 resp.streamChunkId, getRemoteAddress(channel));  
 resp.body().release();  
 } else {  
 outstandingFetches.remove(resp.streamChunkId);  
 listener.onSuccess(resp.streamChunkId.chunkIndex, resp.body());  
 resp.body().release();  
 }  
} else if (message instanceof ChunkFetchFailure) {  
 ChunkFetchFailure resp = (ChunkFetchFailure) message;  
 ChunkReceivedCallback listener = outstandingFetches.get(resp.streamChunkId);  
 if (listener == null) {  
 logger.warn("Ignoring response for block {} from {} ({}) since it is not outstanding",  
 resp.streamChunkId, getRemoteAddress(channel), resp.errorString);  
 } else {  
 outstandingFetches.remove(resp.streamChunkId);  
 listener.onFailure(resp.streamChunkId.chunkIndex, new ChunkFetchFailure-Exception(  
 "Failure while fetching " + resp.streamChunkId + ": " + resp.errorString));  
 }  
}

从代码清单3-46看到，处理ChunkFetchSuccess的逻辑如下。

1）使用ChunkFetchSuccess对应的StreamChunkId，从outstandingFetches缓存中获取注册的ChunkReceivedCallback，此处的ChunkReceivedCallback即为代码清单3-43中传递给fetchChunk方法的ChunkReceivedCallback。

2）移除outstandingFetches缓存中StreamChunkId和ChunkReceivedCallback的注册信息。

3）调用ChunkReceivedCallback的onSuccess方法，处理成功响应后的具体逻辑。这里的ChunkReceivedCallback需要在各个使用TransportClient的fetchChunk方法的场景中分别实现。

4）释放ChunkFetchSuccess的body，回收资源。

处理ChunkFetchFailure的逻辑如下。

1）使用ChunkFetchFailure对应的StreamChunkId，从outstandingFetches缓存中获取注册的ChunkReceivedCallback，此处的ChunkReceivedCallback即为代码清单3-43中传递给fetchChunk方法的ChunkReceivedCallback。

2）移除outstandingFetches缓存中StreamChunkId和ChunkReceivedCallback的注册信息。

3）调用ChunkReceivedCallback的onFailure方法，处理失败响应后的具体逻辑。这里的ChunkReceivedCallback需要在各个使用TransportClient的fetchChunk方法的场景中分别实现。

在详细介绍了TransportClient和TransportResponseHandler之后，对于客户端我们就可以扩展图3-3，把TransportResponseHandler及TransportClient的处理流程增加进来，如图3-8所示。

图3-8中的序号①表示调用TransportResponseHandler的addRpcRequest方法（或addFetchRequest方法），将更新最后一次请求的时间为当前系统时间，然后将requestId与RpcResponseCallback之间的映射加入到outstandingRpcs缓存中（或将StreamChunkId与ChunkReceivedCallback之间的映射加入到outstandingFetches缓存中）。序号②表示调用Channel的writeAndFlush方法将RPC请求发送出去。图中的虚线表示当TransportResponseHandler处理RpcResponse和RpcFailure时，将从outstandingRpcs缓存中获取此请求对应的RpcResponseCallback（或处理ChunkFetchSuccess和ChunkFetchFailure时，将从outstandingFetches缓存中获取StreamChunkId对应的ChunkReceivedCallback），并执行回调。此外，TransportClientBootstrap将可能存在于图3-8中任何两个组件的箭头连线中间。

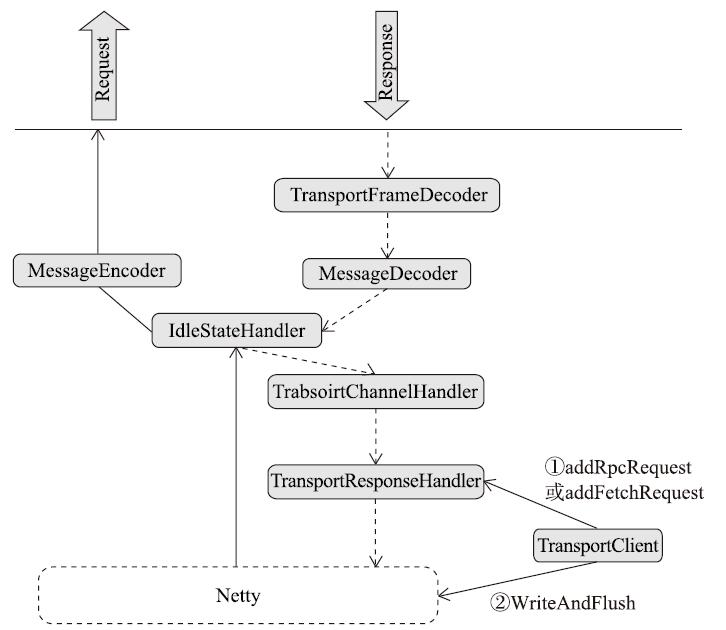


图3-8 客户端请求、响应流程图