# hadoop-rpc源码分析

## 一 hadoop-rpc模块

### 1.1 概述

|  |
| --- |
| Hadoop RPC 框架中的序列化机制实现有两种：   1. Avro Writable 接口实现，简单易懂 2. Google Protobuf 跨语言实现，跨语言，高扩展，高效率   Hadoop-1.x 版本使用默认实现的 Writable 协议作为 RPC 协议，而在 Hadoop-2.x 版本，重写了 RPC 框架，改成默认使用 Protobuf 协议作为 Hadoop 的默认 RPC通信协议。 在 YARN 中， 任何两个需相互通信的组件之间仅有一个 RPC 协议，而对于任何一个 RPC 协议，通信双方有一端是 Client，另一端为 Server，且Client总是主动连接 Server 的，因此，YARN 实际上采用的是拉式（pull-based）通信模型 |

### 1.2 案例

#### 1.2.1 Writable 协议

Pom依赖

|  |
| --- |
| <dependency>  <groupId>org.apache.hadoop</groupId>  <artifactId>hadoop-common</artifactId>  <version>3.1.3</version>  </dependency> |

协议接口

|  |
| --- |
| /\*\*  \* 业务协议接口  \*/  public interface BusinessProtocol {  /\*\*  \* 版本 ID  \*/  long versionID = 123456L;  void mkdirs(String path);  String getName(String name);  } |

协议实现类

|  |
| --- |
| public class BusinessProtocolImpl implements BusinessProtocol {  @Override  public void mkdirs(String path) {  System.out.println("server revived mkdirs request with path " + path);  }  @Override  public String getName(String name) {  System.out.println("server revived getName request");  return "server name";  }  } |

Server 端

|  |
| --- |
| public class BusinessServer {  public static void main(String[] args) {  try {  /\*\*  \* 构建 rpc server  \*/  RPC.Server server = new RPC.Builder(new Configuration())  .setProtocol(BusinessProtocol.class)  .setInstance(new BusinessProtocolImpl())  .setBindAddress("127.0.0.1")  .setPort(10001)  .build();  /\*\*  \* 启动 rpc server  \*/  server.start();  } catch (Exception e) {  e.printStackTrace();  }  }  } |

Client 端

|  |
| --- |
| public class BusinessClient {  public static void main(String[] args) {  try {  /\*\*  \* 获取 rpc client proxy  \*/  BusinessProtocol proxy = RPC.getProxy(  BusinessProtocol.class,  BusinessProtocol.versionID,  new InetSocketAddress("127.0.0.1", 10001),  new Configuration()  );  /\*\*  \* 执行业务处理  \*/  proxy.mkdirs("/usr/root");  String response = proxy.getName("hello-server");  System.out.println("client revived request response with " + response);  } catch (Exception e) {  e.printStackTrace();  }  }  } |

#### 1.2.2 Protobuf 协议

备注：本地笔记本安装 protobuf

Protobuf-3.19 下载链接：

<https://github.com/protocolbuffers/protobuf/releases/tag/v3.19.5>

相关帖子：<https://blog.csdn.net/liu644911337/article/details/128176129>

Pom 依赖

|  |
| --- |
| <dependency>  <groupId>org.apache.hadoop</groupId>  <artifactId>hadoop-common</artifactId>  <version>3.1.3</version>  </dependency>  <dependency>  <groupId>com.google.protobuf</groupId>  <artifactId>protobuf-java</artifactId>  <version>3.19.5</version>  </dependency> |

ResourceTrackerMessage.proto

|  |
| --- |
| option java\_package = "com.tan.hadoop.rpc.proto";  option java\_outer\_classname = "ResourceTrackerMessage";  option java\_generic\_services = true;  option java\_generate\_equals\_and\_hash = true;  message RequestProto{  required string hostname = 1;  required int32 cpu = 2;  required int32 memory = 3;  }  message ResponseProto{  required string flag = 1;  }  // protoc.exe --java\_out=./ ResourceTrackerMessage.proto |

ResourceTracker.proto

|  |
| --- |
| option java\_package = "com.tan.hadoop.rpc.proto";  option java\_outer\_classname = "ResourceTracker";  option java\_generic\_services = true;  option java\_generate\_equals\_and\_hash = true;  import "ResourceTrackerMessage.proto";  service ResourceTrackerService {  rpc registerNodeManager(RequestProto) returns (ResponseProto);  }  // protoc.exe --java\_out=./ ResourceTracker.proto |

协议接口

|  |
| --- |
| public interface ResourceTrackerProtocol {  ResourceTrackerMessage.ResponseProto registerNodeManager(ResourceTrackerMessage.RequestProto request) throws Exception;  } |

协议实现类

|  |
| --- |
| public class ResourceTrackerProtocolImpl implements ResourceTrackerProtocol {  @Override  public ResourceTrackerMessage.ResponseProto registerNodeManager(ResourceTrackerMessage.RequestProto request) throws Exception {  // 构建一个响应对象，用于返回  ResourceTrackerMessage.ResponseProto.Builder builder =  ResourceTrackerMessage.ResponseProto  .newBuilder();  // 输出注册的消息  String hostname = request.getHostname();  int cpu = request.getCpu();  int memory = request.getMemory();  System.out.println("注册消息： hostname = " + hostname + ", cpu = " + cpu + ", memory = " + memory);  // 直接暴力返回 True  builder.setFlag("true");  return builder.build();  }  } |

Proto 协议接口

|  |
| --- |
| @ProtocolInfo(protocolName = "com.tan.hadoop.rpc.proto.ResourceTrackerPB", protocolVersion = 1)  public interface ResourceTrackerPB extends ResourceTracker.ResourceTrackerService.BlockingInterface {  } |

Proto 协议实现类

|  |
| --- |
| public class ResourceTrackerServerSidePB implements ResourceTrackerPB {  final private ResourceTrackerProtocol server;  public ResourceTrackerServerSidePB(ResourceTrackerProtocol server) {  this.server = server;  }  @Override  public ResourceTrackerMessage.ResponseProto registerNodeManager(RpcController controller, ResourceTrackerMessage.RequestProto request) throws ServiceException {  try {  return server.registerNodeManager(request);  } catch (Exception e) {  e.printStackTrace();  }  return null;  }  } |

Server 端

|  |
| --- |
| public class ProtobufRpcServer {  public static void main(String[] args) throws IOException {  Configuration conf = new Configuration();  String hostname = "localhost";  int port = 9998;  RPC.setProtocolEngine(conf, ResourceTrackerPB.class, ProtobufRpcEngine.class);  // 构建 Rpc Server  RPC.Server server = new RPC.Builder(conf)  .setProtocol(ResourceTrackerPB.class)  .setInstance((BlockingService) ResourceTracker.ResourceTrackerService  .newReflectiveBlockingService(new ResourceTrackerServerSidePB(new ResourceTrackerProtocolImpl())))  .setBindAddress(hostname)  .setPort(port)  .setNumHandlers(1)  .setVerbose(true)  .build();  // Rpc Server 启动  server.start();  }  } |

Client 端

|  |
| --- |
| public class ProtobufRpcClient {  public static void main(String[] args) throws IOException {  // 设置 RPC 引擎为 ProtobufRpcEngine  Configuration conf = new Configuration();  String hostname = "localhost";  int port = 9998;  RPC.setProtocolEngine(conf, ResourceTrackerPB.class, ProtobufRpcEngine.class);  // 获取代理  ResourceTrackerPB protocolProxy = RPC  .getProxy(ResourceTrackerPB.class, 1, new InetSocketAddress(hostname, port), conf);  // 构建请求对象  ResourceTrackerMessage.RequestProto.Builder builder =  ResourceTrackerMessage.RequestProto  .newBuilder();  ResourceTrackerMessage.RequestProto requestProto = builder  .setHostname("hadoop")  .setCpu(64)  .setMemory(128)  .build();  // 发送 RPC 请求，获取响应  ResourceTrackerMessage.ResponseProto response = null;  try {  response = protocolProxy  .registerNodeManager(null, requestProto);  // 处理响应  String flag = response.getFlag();  System.out.println("最终注册结果： flag = " + flag);  } catch (ServiceException e) {  e.printStackTrace();  }  }  } |

### 1.3 rpc-server 源码剖析

备注：基于 Protobuf 协议探索 rpc 源码

架构图如下

|  |
| --- |
| Hadoop-RPC架构 |

#### 1.3.0 设置协议引擎

类：org.apache.hadoop.ipc.RPC

|  |
| --- |
| RPC.setProtocolEngine(conf, ResourceTrackerPB.class, ProtobufRpcEngine.class);  ---------------------------------------------------------------------------------------------------------------------------------  public static void setProtocolEngine(Configuration conf,  Class<?> protocol, Class<?> engine) {  conf.setClass(ENGINE\_PROP + "." + protocol.getName(), engine, RpcEngine.class);  } |

#### 1.3.1 初始化 server 配置

类：org.apache.hadoop.ipc.RPC.Builder

##### 1.3.1.1 设置协议接口

|  |
| --- |
| /\*\* Mandatory field \*/  public Builder setProtocol(Class<?> protocol) {  this.protocol = protocol;  return this;  } |

##### 1.3.1.2 设置协议实现类

|  |
| --- |
| /\*\* Mandatory field \*/  public Builder setInstance(Object instance) {  this.instance = instance;  return this;  } |

##### 1.3.1.3 绑定IP和PORT

|  |
| --- |
| /\*\* Default: 0.0.0.0 \*/  public Builder setBindAddress(String bindAddress) {  this.bindAddress = bindAddress;  return this;  }  /\*\* Default: 0 \*/  public Builder setPort(int port) {  this.port = port;  return this;  } |

##### 1.3.1.4 其他核心参数设置

|  |
| --- |
| /\*\* Default: -1 \*/  public Builder setnumReaders(int numReaders) {  this.numReaders = numReaders;  return this;  }  /\*\* Default: 1 \*/  public Builder setNumHandlers(int numHandlers) {  this.numHandlers = numHandlers;  return this;  }  /\*\* Default: -1 \*/  public Builder setQueueSizePerHandler(int queueSizePerHandler) {  this.queueSizePerHandler = queueSizePerHandler;  return this;  } |

#### 1.3.2 构建 RPC.Server 对象

|  |
| --- |
| public Server build() throws IOException, HadoopIllegalArgumentException {  if (this.conf == null) {  throw new HadoopIllegalArgumentException("conf is not set");  }  if (this.protocol == null) {  throw new HadoopIllegalArgumentException("protocol is not set");  }  if (this.instance == null) {  throw new HadoopIllegalArgumentException("instance is not set");  }  return // 获取协议引擎 ProtobufRpcEngine  getProtocolEngine(this.protocol, this.conf)  // 获取 RPC.Server  .getServer(  this.protocol,  this.instance,  this.bindAddress,  this.port,  this.numHandlers,  this.numReaders,  this.queueSizePerHandler,  this.verbose,  this.conf,  this.secretManager,  this.portRangeConfig,  this.alignmentContext);  }  } |

##### 1.3.2.1 获取协议引擎 ProtobufRpcEngine

|  |
| --- |
| // return the RpcEngine configured to handle a protocol  static synchronized RpcEngine getProtocolEngine(Class<?> protocol,  Configuration conf) {  RpcEngine engine = PROTOCOL\_ENGINES.get(protocol);  if (engine == null) {  Class<?> impl = conf.getClass(ENGINE\_PROP + "." + protocol.getName(),  WritableRpcEngine.class);  // 通过反射创建 ProtobufRpcEngine 对象  engine = (RpcEngine) ReflectionUtils.newInstance(impl, conf);  PROTOCOL\_ENGINES.put(protocol, engine);  }  return engine;  } |

##### 1.3.2.2 获取 RPC.Server

|  |
| --- |
| public Server(Class<?> protocolClass, Object protocolImpl,  Configuration conf, String bindAddress, int port, int numHandlers,  int numReaders, int queueSizePerHandler, boolean verbose,  SecretManager<? extends TokenIdentifier> secretManager,  String portRangeConfig, AlignmentContext alignmentContext)  throws IOException {  // 调用父类  super(bindAddress, port, null, numHandlers,  numReaders, queueSizePerHandler, conf, classNameBase(protocolImpl  .getClass().getName()), secretManager, portRangeConfig);  setAlignmentContext(alignmentContext);  this.verbose = verbose;  // 注册协议及其实现类  registerProtocolAndImpl(RPC.RpcKind.RPC\_PROTOCOL\_BUFFER, protocolClass,  protocolImpl);  } |

###### 1.3.2.2.1 RPC.Server 构造方法

|  |
| --- |
| protected Server(String bindAddress, int port,  Class<? extends Writable> rpcRequestClass, int handlerCount,  int numReaders, int queueSizePerHandler, Configuration conf,  String serverName, SecretManager<? extends TokenIdentifier> secretManager,  String portRangeConfig)  throws IOException {  this.bindAddress = bindAddress;  this.conf = conf;  this.portRangeConfig = portRangeConfig;  this.port = port;  this.rpcRequestClass = rpcRequestClass;  this.handlerCount = handlerCount;  this.socketSendBufferSize = 0;  this.serverName = serverName;  this.auxiliaryListenerMap = null;  // Server 端接收最大数据长度 默认值 64MB (key = ipc.maximum.data.length)  this.maxDataLength = conf.getInt(CommonConfigurationKeys.IPC\_MAXIMUM\_DATA\_LENGTH,  CommonConfigurationKeys.IPC\_MAXIMUM\_DATA\_LENGTH\_DEFAULT);  if (queueSizePerHandler != -1) {  this.maxQueueSize = handlerCount \* queueSizePerHandler;  } else {  // 每个 handler 线程队列最大接收多少个 call 默认值 100 (key = ipc.server.handler.queue.size)  this.maxQueueSize = handlerCount \* conf.getInt(  CommonConfigurationKeys.IPC\_SERVER\_HANDLER\_QUEUE\_SIZE\_KEY,  CommonConfigurationKeys.IPC\_SERVER\_HANDLER\_QUEUE\_SIZE\_DEFAULT);  }  // Server 端返回客户端请求最大数 默认值 1024\*1024 (key = ipc.server.max.response.size)  this.maxRespSize = conf.getInt(  CommonConfigurationKeys.IPC\_SERVER\_RPC\_MAX\_RESPONSE\_SIZE\_KEY,  CommonConfigurationKeys.IPC\_SERVER\_RPC\_MAX\_RESPONSE\_SIZE\_DEFAULT);  if (numReaders != -1) {  this.readThreads = numReaders;  } else {  // Server 端读取 socket 线程数 默认值 1 (key = ipc.server.read.threadpool.size)  this.readThreads = conf.getInt(  CommonConfigurationKeys.IPC\_SERVER\_RPC\_READ\_THREADS\_KEY,  CommonConfigurationKeys.IPC\_SERVER\_RPC\_READ\_THREADS\_DEFAULT);  }  // Server 端连接 socket 最大阻塞数 默认值 100 (key = ipc.server.read.connection-queue.size)  this.readerPendingConnectionQueue = conf.getInt(  CommonConfigurationKeys.IPC\_SERVER\_RPC\_READ\_CONNECTION\_QUEUE\_SIZE\_KEY,  CommonConfigurationKeys.IPC\_SERVER\_RPC\_READ\_CONNECTION\_QUEUE\_SIZE\_DEFAULT);  // Setup appropriate callqueue  final String prefix = getQueueClassPrefix();  // 创建 CallQueueManager 对象  this.callQueue = new CallQueueManager<Call>(  // 获取阻塞队列类型 默认值 LinkedBlockingQueue.class  getQueueClass(prefix, conf),  // 获取调度器 默认值 DefaultRpcScheduler.class  getSchedulerClass(prefix, conf),  // 默认值 false  getClientBackoffEnable(prefix, conf),  maxQueueSize,  prefix,  conf);  this.secretManager = (SecretManager<TokenIdentifier>) secretManager;  this.authorize =  conf.getBoolean(CommonConfigurationKeys.HADOOP\_SECURITY\_AUTHORIZATION,  false);  // configure supported authentications  this.enabledAuthMethods = getAuthMethods(secretManager, conf);  this.negotiateResponse = buildNegotiateResponse(enabledAuthMethods);  // Start the listener here and let it bind to the port  // 创建 Listener 线程并启动 ServerSocketChannel 监听 OP\_ACCEPT 事件  listener = new Listener(port);  // set the server port to the default listener port.  this.port = listener.getAddress().getPort();  // 创建客户端连接管理器 ConnectionManager  connectionManager = new ConnectionManager();  this.rpcMetrics = RpcMetrics.create(this, conf);  this.rpcDetailedMetrics = RpcDetailedMetrics.create(this.port);  this.tcpNoDelay = conf.getBoolean(  CommonConfigurationKeysPublic.IPC\_SERVER\_TCPNODELAY\_KEY,  CommonConfigurationKeysPublic.IPC\_SERVER\_TCPNODELAY\_DEFAULT);  this.setLogSlowRPC(conf.getBoolean(  CommonConfigurationKeysPublic.IPC\_SERVER\_LOG\_SLOW\_RPC,  CommonConfigurationKeysPublic.IPC\_SERVER\_LOG\_SLOW\_RPC\_DEFAULT));  // Create the responder here  // 创建 Responder 线程  responder = new Responder();  if (secretManager != null || UserGroupInformation.isSecurityEnabled()) {  SaslRpcServer.init(conf);  saslPropsResolver = SaslPropertiesResolver.getInstance(conf);  }  this.exceptionsHandler.addTerseLoggingExceptions(StandbyException.class);  } |

1. 创建 CallQueueManager

|  |
| --- |
| public CallQueueManager(Class<? extends BlockingQueue<E>> backingClass,  Class<? extends RpcScheduler> schedulerClass,  boolean clientBackOffEnabled, int maxQueueSize, String namespace,  Configuration conf) {  int priorityLevels = parseNumLevels(namespace, conf);  // 创建 DefaultRpcScheduler 对象  this.scheduler = createScheduler(schedulerClass, priorityLevels,  namespace, conf);  // 创建 LinkedBlockingQueue 对象  BlockingQueue<E> bq = createCallQueueInstance(backingClass,  priorityLevels, maxQueueSize, namespace, conf);  this.clientBackOffEnabled = clientBackOffEnabled;  // 创建两个原子引用 LinkedBlockingQueue  this.putRef = new AtomicReference<BlockingQueue<E>>(bq);  this.takeRef = new AtomicReference<BlockingQueue<E>>(bq);  LOG.info("Using callQueue: " + backingClass + " queueCapacity: " +  maxQueueSize + " scheduler: " + schedulerClass);  } |

1. 创建 Listener 线程

|  |
| --- |
| Listener(int port) throws IOException {  address = new InetSocketAddress(bindAddress, port);  // Create a new server socket and set to non-blocking mode  acceptChannel = ServerSocketChannel.open();  acceptChannel.configureBlocking(false);  // Bind the server socket to the local host and port  bind(acceptChannel.socket(), address, backlogLength, conf, portRangeConfig);  // Could be an ephemeral port  this.listenPort = acceptChannel.socket().getLocalPort();  Thread.currentThread().setName("Listener at " +  bindAddress + "/" + this.listenPort);  // create a selector;  selector = Selector.open();  // 默认 Reader 数组大小 1  readers = new Reader[readThreads];  for (int i = 0; i < readThreads; i++) {  // 创建 Reader 线程并启动  Reader reader = new Reader(  "Socket Reader #" + (i + 1) + " for port " + port);  readers[i] = reader;  reader.start();  }  // Register accepts on the server socket with the selector  acceptChannel.register(selector, SelectionKey.OP\_ACCEPT);  this.setName("IPC Server listener on " + port);  this.setDaemon(true);  } |

* 1. 创建 Reader 线程

|  |
| --- |
| Reader(String name) throws IOException {  super(name);  // 创建 LinkedBlockingQueue 默认值 100  this.pendingConnections =  new LinkedBlockingQueue<Connection>(readerPendingConnectionQueue);  // open one selector to loop socket connection  this.readSelector = Selector.open();  } |

1. 创建 ConnectionManager

|  |
| --- |
| ConnectionManager() {  this.idleScanTimer = new Timer(  "IPC Server idle connection scanner for port " + getPort(), true);  this.idleScanThreshold = conf.getInt(  CommonConfigurationKeysPublic.IPC\_CLIENT\_IDLETHRESHOLD\_KEY,  CommonConfigurationKeysPublic.IPC\_CLIENT\_IDLETHRESHOLD\_DEFAULT);  this.idleScanInterval = conf.getInt(  CommonConfigurationKeys.IPC\_CLIENT\_CONNECTION\_IDLESCANINTERVAL\_KEY,  CommonConfigurationKeys.IPC\_CLIENT\_CONNECTION\_IDLESCANINTERVAL\_DEFAULT);  this.maxIdleTime = 2 \* conf.getInt(  CommonConfigurationKeysPublic.IPC\_CLIENT\_CONNECTION\_MAXIDLETIME\_KEY,  CommonConfigurationKeysPublic.IPC\_CLIENT\_CONNECTION\_MAXIDLETIME\_DEFAULT);  this.maxIdleToClose = conf.getInt(  CommonConfigurationKeysPublic.IPC\_CLIENT\_KILL\_MAX\_KEY,  CommonConfigurationKeysPublic.IPC\_CLIENT\_KILL\_MAX\_DEFAULT);  this.maxConnections = conf.getInt(  CommonConfigurationKeysPublic.IPC\_SERVER\_MAX\_CONNECTIONS\_KEY,  CommonConfigurationKeysPublic.IPC\_SERVER\_MAX\_CONNECTIONS\_DEFAULT);  // create a set with concurrency -and- a thread-safe iterator, add 2  // for listener and idle closer threads  this.connections = Collections.newSetFromMap(  new ConcurrentHashMap<Connection, Boolean>(  maxQueueSize, 0.75f, readThreads + 2));  this.userToConnectionsMap = new ConcurrentHashMap<>();  } |

1. 创建 Responder 线程

|  |
| --- |
| Responder() throws IOException {  this.setName("IPC Server Responder");  this.setDaemon(true);  writeSelector = Selector.open(); // create a selector  pending = 0;  } |

###### 1.3.2.2.2 注册协议及其实现类

|  |
| --- |
| // Register protocol and its impl for rpc calls  void registerProtocolAndImpl(RpcKind rpcKind, Class<?> protocolClass,  Object protocolImpl) {  // 获取协议接口 @ProtocolInfo 注解的 protocolName 属性对应的值  String protocolName = RPC.getProtocolName(protocolClass);  long version;  try {  // 获取协议接口 @ProtocolInfo 注解的 protocolVersion 属性对应的值  version = RPC.getProtocolVersion(protocolClass);  } catch (Exception ex) {  LOG.warn("Protocol " + protocolClass +  " NOT registered as cannot get protocol version ");  return;  }  // 获取协议容器 Map  getProtocolImplMap(rpcKind)  // 协议及其协议实现类添加到容器 Map  .put(  new ProtoNameVer(protocolName, version),  new ProtoClassProtoImpl(protocolClass, protocolImpl));  if (LOG.isDebugEnabled()) {  LOG.debug("RpcKind = " + rpcKind + " Protocol Name = " + protocolName +  " version=" + version +  " ProtocolImpl=" + protocolImpl.getClass().getName() +  " protocolClass=" + protocolClass.getName());  }  } |

#### 1.3.3 启动 RPC.Server

|  |
| --- |
| /\*\*  \* Starts the service. Must be called before any calls will be handled.  \*/  public synchronized void start() {  // 启动 Responder 线程 (本质调用 Responder run())  responder.start();  // 启动 Listener 线程 (本质调用 Listener run())  listener.start();  if (auxiliaryListenerMap != null && auxiliaryListenerMap.size() > 0) {  for (Listener newListener : auxiliaryListenerMap.values()) {  newListener.start();  }  }  // 创建 Handler 线程数组 默认大小 1  handlers = new Handler[handlerCount];  // 初始化 Handler 线程并启动  for (int i = 0; i < handlerCount; i++) {  handlers[i] = new Handler(i);  handlers[i].start();  }  } |

#### 1.3.4 RPC.Server 接收 Client 请求

##### 1.3.4.1 Listener 线程监听 Client连接请求

备注：调用 Listener 线程的 run()

|  |
| --- |
| @Override  public void run() {  LOG.info(Thread.currentThread().getName() + ": starting");  SERVER.set(Server.this);  // 定时扫描非激活状态的 Connection 并清除 避免 OOM  connectionManager.startIdleScan();  while (running) {  SelectionKey key = null;  try {  // 阻塞等待 Client 端 OP\_ACCEPT 事件  getSelector().select();  Iterator<SelectionKey> iter = getSelector().selectedKeys().iterator();  while (iter.hasNext()) {  key = iter.next();  iter.remove();  try {  if (key.isValid()) {  if (key.isAcceptable())  // 执行接收 Client 端 OP\_ACCEPT 事件  doAccept(key);  }  } catch (IOException e) {  }  key = null;  }  } catch (OutOfMemoryError e) {  ......  } catch (Exception e) {  ......  }  }  ......  } |

###### 1.3.4.1.1 doAccept() 核心业务逻辑

|  |
| --- |
| void doAccept(SelectionKey key) throws InterruptedException, IOException, OutOfMemoryError {  ServerSocketChannel server = (ServerSocketChannel) key.channel();  SocketChannel channel;  // 接收 Client 端的 SocketChannel 通道  while ((channel = server.accept()) != null) {  // 设置 SocketChannel 属性  channel.configureBlocking(false);  channel.socket().setTcpNoDelay(tcpNoDelay);  channel.socket().setKeepAlive(true);  // 轮询获取 Reader 线程  Reader reader = getReader();  // 封装 SocketChannel 为 Connection 对象并添加到 ConnectionManager 的 Set 集合  Connection c = connectionManager.register(channel, this.listenPort);  // If the connectionManager can't take it, close the connection.  if (c == null) {  if (channel.isOpen()) {  IOUtils.cleanup(null, channel);  }  connectionManager.droppedConnections.getAndIncrement();  continue;  }  // 绑定 Connection 到 SelectionKey 以便后续通过 SelectionKey.attachment() 获取对应的 Channel  key.attach(c); // so closeCurrentConnection can get the object  // 将 Connection 添加到 Reader 线程的 LinkedBlockingQueue 阻塞队列并唤醒 Selector  reader.addConnection(c);  }  } |

##### 1.3.4.2 Reader 线程监听阻塞队列是否有 Connection

备注：调用 Reader 线程的 run() 核心代码如下

|  |
| --- |
| private synchronized void doRunLoop() {  while (running) {  SelectionKey key = null;  try {  int size = pendingConnections.size();  for (int i = size; i > 0; i--) {  // 弹出 Connection  Connection conn = pendingConnections.take();  // 往 Reader 线程的 NIO Selector 注册 OP\_READ 事件并绑定 Connection  conn.channel.register(readSelector, SelectionKey.OP\_READ, conn);  }  // 被 Listener 线程唤醒  readSelector.select();  // 第一次被 Listener 线程唤醒 iter 为空  Iterator<SelectionKey> iter = readSelector.selectedKeys().iterator();  while (iter.hasNext()) {  key = iter.next();  iter.remove();  try {  if (key.isReadable()) {  // 监听到 OP\_READ 事件 执行读取 SocketChannel 数据  doRead(key);  }  } catch (CancelledKeyException cke) {  ......  }  key = null;  }  } catch (InterruptedException e) {  ......  }  } |

###### 1.3.4.2.1 doRead() 核心业务逻辑

|  |
| --- |
| void doRead(SelectionKey key) throws InterruptedException {  int count;  // 获取 SelectionKey 绑定对象 也即 Connection  Connection c = (Connection) key.attachment();  if (c == null) {  return;  }  // 更新 Connection 上次连接时间戳  c.setLastContact(Time.now());  try {  // 读取 Connection 的 SocketChannel 数据区并处理  count = c.readAndProcess();  } catch (InterruptedException ieo) {  LOG.info(Thread.currentThread().getName() + ": readAndProcess caught InterruptedException", ieo);  throw ieo;  } catch (Exception e) {  // Any exceptions that reach here are fatal unexpected internal errors  // that could not be sent to the client.  LOG.info(Thread.currentThread().getName() +  ": readAndProcess from client " + c +  " threw exception [" + e + "]", e);  count = -1; //so that the (count < 0) block is executed  }  // setupResponse will signal the connection should be closed when a  // fatal response is sent.  if (count < 0 || c.shouldClose()) {  closeConnection(c);  c = null;  } else {  c.setLastContact(Time.now());  }  } |

1 读取 RPC Request 数据区

|  |
| --- |
| public int readAndProcess() throws IOException, InterruptedException {  while (!shouldClose()) { // stop if a fatal response has been sent.  // dataLengthBuffer is used to read "hrpc" or the rpc-packet length  int count = -1;  // 判断数据长度 buffer 是否可以写 一般情况下都是数据长度为 4  if (dataLengthBuffer.remaining() > 0) {  // 读取数据长度到 dataLengthBuffer  count = channelRead(channel, dataLengthBuffer);  if (count < 0 || dataLengthBuffer.remaining() > 0)  return count;  }  if (!connectionHeaderRead) {  // Every connection is expected to send the header;  // so far we read "hrpc" of the connection header.  if (connectionHeaderBuf == null) {  // for the bytes that follow "hrpc", in the connection header  // 申请 3 字节缓存数据头部 header 信息  connectionHeaderBuf = ByteBuffer.allocate(HEADER\_LEN\_AFTER\_HRPC\_PART);  }  // 读取头部 header 信息 (3 个字节)  count = channelRead(channel, connectionHeaderBuf);  if (count < 0 || connectionHeaderBuf.remaining() > 0) {  return count;  }  // 从头部 header 获取数据版本号  int version = connectionHeaderBuf.get(0);  // TODO we should add handler for service class later  this.setServiceClass(connectionHeaderBuf.get(1));  // 数据长度 buffer 变为可读  dataLengthBuffer.flip();  // Check if it looks like the user is hitting an IPC port  // with an HTTP GET - this is a common error, so we can  // send back a simple string indicating as much.  if (HTTP\_GET\_BYTES.equals(dataLengthBuffer)) {  setupHttpRequestOnIpcPortResponse();  return -1;  }  if (!RpcConstants.HEADER.equals(dataLengthBuffer)) {  LOG.warn("Incorrect RPC Header length from {}:{} "  + "expected length: {} got length: {}",  hostAddress, remotePort, RpcConstants.HEADER, dataLengthBuffer);  setupBadVersionResponse(version);  return -1;  }  if (version != CURRENT\_VERSION) {  //Warning is ok since this is not supposed to happen.  LOG.warn("Version mismatch from " +  hostAddress + ":" + remotePort +  " got version " + version +  " expected version " + CURRENT\_VERSION);  setupBadVersionResponse(version);  return -1;  }  // this may switch us into SIMPLE  authProtocol = initializeAuthContext(connectionHeaderBuf.get(2));  // 清除 buffer (不是请求 buffer 数据而是更新 buffer 的标识符)  dataLengthBuffer.clear(); // clear to next read rpc packet len  connectionHeaderBuf = null;  connectionHeaderRead = true;  continue; // connection header read, now read 4 bytes rpc packet len  }  if (data == null) { // just read 4 bytes - length of RPC packet  dataLengthBuffer.flip();  // 读取数据长度 (4 个字节表示 RPC 包的长度)  dataLength = dataLengthBuffer.getInt();  checkDataLength(dataLength);  // Set buffer for reading EXACTLY the RPC-packet length and no more.  // 申请指定大小的 ByteBuffer  data = ByteBuffer.allocate(dataLength);  }  // Now read the RPC packet  // 读取数据到 ByteBuffer  count = channelRead(channel, data);  // 读取指定数据长度完成 也即读取完成一个 RPC package  if (data.remaining() == 0) {  dataLengthBuffer.clear(); // to read length of future rpc packets  data.flip();  ByteBuffer requestData = data;  data = null; // null out in case processOneRpc throws.  boolean isHeaderRead = connectionContextRead;  // 处理一个 RPC 请求  processOneRpc(requestData);  // the last rpc-request we processed could have simply been the  // connectionContext; if so continue to read the first RPC.  if (!isHeaderRead) {  continue;  }  }  return count;  }  return -1;  } |

2 处理 RPC Request 请求

|  |
| --- |
| private void processOneRpc(ByteBuffer bb)  throws IOException, InterruptedException {  // exceptions that escape this method are fatal to the connection.  // setupResponse will use the rpc status to determine if the connection  // should be closed.  int callId = -1;  int retry = RpcConstants.INVALID\_RETRY\_COUNT;  try {  // 封装数据 ByteBuffer 为 Buffer 对象 (Buffer 底层还是维护一个 ByteBuffer)  final RpcWritable.Buffer buffer = RpcWritable.Buffer.wrap(bb);  // 读取数据区的 RpcRequestHeaderProto (通过依次读取字节 因为 RpcRequestHeaderProto 对象的字节固定)  final RpcRequestHeaderProto header =  getMessage(RpcRequestHeaderProto.getDefaultInstance(), buffer);  callId = header.getCallId();  retry = header.getRetryCount();  if (LOG.isDebugEnabled()) {  LOG.debug(" got #" + callId);  }  checkRpcHeaders(header);  if (callId < 0) { // callIds typically used during connection setup  processRpcOutOfBandRequest(header, buffer);  } else if (!connectionContextRead) {  throw new FatalRpcServerException(  RpcErrorCodeProto.FATAL\_INVALID\_RPC\_HEADER,  "Connection context not established");  } else {  // 处理 RPC 请求  processRpcRequest(header, buffer);  }  } catch (RpcServerException rse) {  ......  }  } |

|  |
| --- |
| private void processRpcRequest(RpcRequestHeaderProto header,  RpcWritable.Buffer buffer) throws RpcServerException,  InterruptedException {  // 基于 rpcKind 获取 RpcProtobufRequest  Class<? extends Writable> rpcRequestClass =  getRpcRequestWrapper(header.getRpcKind());  if (rpcRequestClass == null) {  LOG.warn("Unknown rpc kind " + header.getRpcKind() +  " from client " + getHostAddress());  final String err = "Unknown rpc kind in rpc header" +  header.getRpcKind();  throw new FatalRpcServerException(  RpcErrorCodeProto.FATAL\_INVALID\_RPC\_HEADER, err);  }  Writable rpcRequest;  try {  // Read the rpc request  // 读取 rpc request 字节反射创建 RpcProtobufRequest 对象  rpcRequest = buffer.newInstance(rpcRequestClass, conf);  } catch (RpcServerException rse) { // lets tests inject failures.  throw rse;  } catch (Throwable t) { // includes runtime exception from newInstance  LOG.warn("Unable to read call parameters for client " +  getHostAddress() + "on connection protocol " +  this.protocolName + " for rpcKind " + header.getRpcKind(), t);  String err = "IPC server unable to read call parameters: " + t.getMessage();  throw new FatalRpcServerException(  RpcErrorCodeProto.FATAL\_DESERIALIZING\_REQUEST, err);  }  TraceScope traceScope = null;  if (header.hasTraceInfo()) {  if (tracer != null) {  // If the incoming RPC included tracing info, always continue the  // trace  SpanId parentSpanId = new SpanId(  header.getTraceInfo().getTraceId(),  header.getTraceInfo().getParentId());  traceScope = tracer.newScope(  RpcClientUtil.toTraceName(rpcRequest.toString()),  parentSpanId);  traceScope.detach();  }  }  CallerContext callerContext = null;  if (header.hasCallerContext()) {  callerContext =  new CallerContext.Builder(header.getCallerContext().getContext())  .setSignature(header.getCallerContext().getSignature()  .toByteArray())  .build();  }  // 创建 RpcCall 对象 封装一个 rpc request  RpcCall call = new RpcCall(  // this -> 当前 Connection  this,  header.getCallId(),  header.getRetryCount(),  rpcRequest,  ProtoUtil.convert(header.getRpcKind()),  header.getClientId().toByteArray(),  traceScope,  callerContext);  // Save the priority level assignment by the scheduler  call.setPriorityLevel(callQueue.getPriorityLevel(call));  call.markCallCoordinated(false);  if (alignmentContext != null && call.rpcRequest != null &&  (call.rpcRequest instanceof ProtobufRpcEngine.RpcProtobufRequest)) {  // if call.rpcRequest is not RpcProtobufRequest, will skip the following  // step and treat the call as uncoordinated. As currently only certain  // ClientProtocol methods request made through RPC protobuf needs to be  // coordinated.  String methodName;  String protoName;  ProtobufRpcEngine.RpcProtobufRequest req =  (ProtobufRpcEngine.RpcProtobufRequest) call.rpcRequest;  try {  methodName = req.getRequestHeader().getMethodName();  protoName = req.getRequestHeader().getDeclaringClassProtocolName();  if (alignmentContext.isCoordinatedCall(protoName, methodName)) {  call.markCallCoordinated(true);  long stateId;  stateId = alignmentContext.receiveRequestState(  header, getMaxIdleTime());  call.setClientStateId(stateId);  }  } catch (IOException ioe) {  throw new RpcServerException("Processing RPC request caught ", ioe);  }  }  try {  // 将 RpcCall 对象添加到 CallQueueManager 的 putRef 属性原子引用的阻塞队列中 等待 Handler 线程处理  internalQueueCall(call);  } catch (RpcServerException rse) {  throw rse;  } catch (IOException ioe) {  throw new FatalRpcServerException(  RpcErrorCodeProto.ERROR\_RPC\_SERVER, ioe);  }  incRpcCount(); // Increment the rpc count  } |

##### 1.3.4.3 Handler 线程处理 Rpc Request

备注：调用 Handler 线程的 run()

|  |
| --- |
| @Override  public void run() {  LOG.debug(Thread.currentThread().getName() + ": starting");  SERVER.set(Server.this);  while (running) {  TraceScope traceScope = null;  Call call = null;  long startTimeNanos = 0;  // True iff the connection for this call has been dropped.  // Set to true by default and update to false later if the connection  // can be succesfully read.  boolean connDropped = true;  try {  // 从 CallQueueManager 阻塞队列弹出一个 RpcCall  call = callQueue.take(); // pop the queue; maybe blocked here  startTimeNanos = Time.monotonicNowNanos();  if (alignmentContext != null && call.isCallCoordinated() &&  call.getClientStateId() > alignmentContext.getLastSeenStateId()) {  // Re-queue the call and continue  requeueCall(call);  continue;  }  if (LOG.isDebugEnabled()) {  LOG.debug(Thread.currentThread().getName() + ": " + call + " for RpcKind " + call.rpcKind);  }  // 当前线程缓存 RpcCall  CurCall.set(call);  if (call.traceScope != null) {  call.traceScope.reattach();  traceScope = call.traceScope;  traceScope.getSpan().addTimelineAnnotation("called");  }  // always update the current call context  CallerContext.setCurrent(call.callerContext);  UserGroupInformation remoteUser = call.getRemoteUser();  connDropped = !call.isOpen();  if (remoteUser != null) {  remoteUser.doAs(call);  } else {  // 执行 RpcCall run()  call.run();  }  } catch (InterruptedException e) {  if (running) { // unexpected -- log it  LOG.info(Thread.currentThread().getName() + " unexpectedly interrupted", e);  if (traceScope != null) {  traceScope.getSpan().addTimelineAnnotation("unexpectedly interrupted: " +  StringUtils.stringifyException(e));  }  }  } catch (Exception e) {  LOG.info(Thread.currentThread().getName() + " caught an exception", e);  if (traceScope != null) {  traceScope.getSpan().addTimelineAnnotation("Exception: " +  StringUtils.stringifyException(e));  }  } finally {  // 清除当前线程缓存 RpcCall  CurCall.set(null);  IOUtils.cleanupWithLogger(LOG, traceScope);  if (call != null) {  updateMetrics(call, startTimeNanos, connDropped);  ProcessingDetails.LOG.debug(  "Served: [{}]{} name={} user={} details={}",  call, (call.isResponseDeferred() ? ", deferred" : ""),  call.getDetailedMetricsName(), call.getRemoteUser(),  call.getProcessingDetails());  }  }  }  LOG.debug(Thread.currentThread().getName() + ": exiting");  } |

###### 1.3.4.3.1 执行 RPC Request 的核心业务逻辑

备注：RpcCall.run()

|  |
| --- |
| @Override  public Void run() throws Exception {  if (!connection.channel.isOpen()) {  Server.LOG.info(Thread.currentThread().getName() + ": skipped " + this);  return null;  }  long startNanos = Time.monotonicNowNanos();  Writable value = null;  // 创建 ResponseParams 对象 以便返回 Rpc 请求  ResponseParams responseParams = new ResponseParams();  try {  // 执行 Rpc 请求  value = call(  rpcKind,  connection.protocolName,  rpcRequest,  timestampNanos);  } catch (Throwable e) {  populateResponseParamsOnError(e, responseParams);  }  if (!isResponseDeferred()) {  long deltaNanos = Time.monotonicNowNanos() - startNanos;  ProcessingDetails details = getProcessingDetails();  details.set(Timing.PROCESSING, deltaNanos, TimeUnit.NANOSECONDS);  deltaNanos -= details.get(Timing.LOCKWAIT, TimeUnit.NANOSECONDS);  deltaNanos -= details.get(Timing.LOCKSHARED, TimeUnit.NANOSECONDS);  deltaNanos -= details.get(Timing.LOCKEXCLUSIVE, TimeUnit.NANOSECONDS);  details.set(Timing.LOCKFREE, deltaNanos, TimeUnit.NANOSECONDS);  startNanos = Time.monotonicNowNanos();  // 设置当前 RpcCall 的返回值参数  setResponseFields(value, responseParams);  // 发送 Rpc Request 请求结果  sendResponse();  deltaNanos = Time.monotonicNowNanos() - startNanos;  details.set(Timing.RESPONSE, deltaNanos, TimeUnit.NANOSECONDS);  } else {  if (LOG.isDebugEnabled()) {  LOG.debug("Deferring response for callId: " + this.callId);  }  }  return null;  } |

1 通过读取 RPC Request 的字节数据反序列化得到具体的执行类以及方法并返回结果

备注：RPC.call()

|  |
| --- |
| @Override  public Writable call(RPC.RpcKind rpcKind, String protocol,  Writable rpcRequest, long receiveTime) throws Exception {  // 获取 Rpc 执行类 也即 ProtoBufRpcInvoker  return getRpcInvoker(rpcKind)  .call(this, protocol, rpcRequest, receiveTime);  } |

备注：ProtoBufInvoker.call()

|  |
| --- |
| public Writable call(RPC.Server server, String connectionProtocolName,  Writable writableRequest, long receiveTime) throws Exception {  // 强转为 RpcProtobufRequest  RpcProtobufRequest request = (RpcProtobufRequest) writableRequest;  // 获取 Rpc Request 头部 header  RequestHeaderProto rpcRequest = request.getRequestHeader();  // 获取 Rpc Request 执行方法名称  String methodName = rpcRequest.getMethodName();  // 获取 Rpc Request 执行方法全类名  String declaringClassProtoName =  rpcRequest.getDeclaringClassProtocolName();  long clientVersion = rpcRequest.getClientProtocolVersion();  if (server.verbose)  LOG.info("Call: connectionProtocolName=" + connectionProtocolName +  ", method=" + methodName);  // 获取 Rpc Request 请求处理类  ProtoClassProtoImpl protocolImpl = getProtocolImpl(server,  declaringClassProtoName, clientVersion);  BlockingService service = (BlockingService) protocolImpl.protocolImpl;  // 通过 Rpc Request 处理类以及执行方法名称得到具体的执行方法描述器  MethodDescriptor methodDescriptor = service.getDescriptorForType()  .findMethodByName(methodName);  if (methodDescriptor == null) {  String msg = "Unknown method " + methodName + " called on "  + connectionProtocolName + " protocol.";  LOG.warn(msg);  throw new RpcNoSuchMethodException(msg);  }  Message prototype = service.getRequestPrototype(methodDescriptor);  // Rpc Request 请求参数  Message param = request.getValue(prototype);  Message result;  // 从当前线程获取当前 RpcCall 对象  Call currentCall = Server.getCurCall().get();  try {  server.rpcDetailedMetrics.init(protocolImpl.protocolClass);  currentCallInfo.set(new CallInfo(server, methodName));  currentCall.setDetailedMetricsName(methodName);  // 执行具体的 Rpc Request 处理方法并返回值  result = service.callBlockingMethod(methodDescriptor, null, param);  // Check if this needs to be a deferred response,  // by checking the ThreadLocal callback being set  if (currentCallback.get() != null) {  currentCall.deferResponse();  currentCallback.set(null);  return null;  }  } catch (ServiceException e) {  Exception exception = (Exception) e.getCause();  currentCall.setDetailedMetricsName(  exception.getClass().getSimpleName());  throw (Exception) e.getCause();  } catch (Exception e) {  currentCall.setDetailedMetricsName(e.getClass().getSimpleName());  throw e;  } finally {  currentCallInfo.set(null);  }  // 最终执行完成具体的 Rpc Request 请求的具体方法返回结果  return RpcWritable.wrap(result);  }  }  } |

2 处理 RPC Request 请求结果

备注：RpcCall.doResponse()

|  |
| --- |
| @Override  void doResponse(Throwable t, RpcStatusProto status) throws IOException {  RpcCall call = this;  if (t != null) {  if (status == null) {  status = RpcStatusProto.FATAL;  }  // clone the call to prevent a race with another thread stomping  // on the response while being sent. the original call is  // effectively discarded since the wait count won't hit zero  call = new RpcCall(this);  setupResponse(call, status, RpcErrorCodeProto.ERROR\_RPC\_SERVER,  null, t.getClass().getName(), StringUtils.stringifyException(t));  } else {  // 设置 Rpc Request 返回值头部信息以及返回字节数据  setupResponse(call, call.responseParams.returnStatus,  call.responseParams.detailedErr, call.rv,  call.responseParams.errorClass,  call.responseParams.error);  }  // 将 Rpc Request 处理结果发送给 Client 端 (先 put 到当前 Connection 的 response 阻塞队列 等待 Responder 线程执行)  connection.sendResponse(call);  } |

|  |
| --- |
| // ipc reader threads should invoke this directly, whereas handlers  // must invoke call.sendResponse to allow lifecycle management of  // external, postponed, deferred calls, etc.  private void sendResponse(RpcCall call) throws IOException {  // 将 Rpc Request 请求结果添加到当前 Connection 的 response 柱塞队列 等待 Responder 线程处理返回给 Client 端  responder.doRespond(call);  } |

备注：Responder.doRespond()

|  |
| --- |
| void doRespond(RpcCall call) throws IOException {  synchronized (call.connection.responseQueue) {  // must only wrap before adding to the responseQueue to prevent  // postponed responses from being encrypted and sent out of order.  if (call.connection.useWrap) {  wrapWithSasl(call);  }  // 将 Rpc Request 请求结果添加到当前 Connection 对象的 response 阻塞队列尾部  call.connection.responseQueue.addLast(call);  // 如果当前 Connection 对象的 response 阻塞队列大小为 1 直接由 Handler 线程返回 否则等待 Responder 线程处理  if (call.connection.responseQueue.size() == 1) {  processResponse(call.connection.responseQueue, true);  }  }  } |

|  |
| --- |
| // Processes one response. Returns true if there are no more pending  // data for this channel.  //  private boolean processResponse(LinkedList<RpcCall> responseQueue,  boolean inHandler) throws IOException {  boolean error = true;  boolean done = false; // there is more data for this channel.  int numElements = 0;  RpcCall call = null;  try {  synchronized (responseQueue) {  //  // If there are no items for this channel, then we are done  //  numElements = responseQueue.size();  if (numElements == 0) {  error = false;  return true; // no more data for this channel.  }  //  // Extract the first call  // 弹出 RpcCall 对象 以便后续返回 Rpc 结果给 Client 端  call = responseQueue.removeFirst();  // 获取当前 Rpc Request SocketChannel 通道  SocketChannel channel = call.connection.channel;  if (LOG.isDebugEnabled()) {  LOG.debug(Thread.currentThread().getName() + ": responding to " + call);  }  //  // Send as much data as we can in the non-blocking fashion  // 执行将 Rpc Request 的结果返回给 Client 端  int numBytes = channelWrite(channel, call.rpcResponse);  if (numBytes < 0) {  return true;  }  if (!call.rpcResponse.hasRemaining()) {  //Clear out the response buffer so it can be collected  call.rpcResponse = null;  call.connection.decRpcCount();  if (numElements == 1) { // last call fully processes.  done = true; // no more data for this channel.  } else {  done = false; // more calls pending to be sent.  }  if (LOG.isDebugEnabled()) {  LOG.debug(Thread.currentThread().getName() + ": responding to " + call  + " Wrote " + numBytes + " bytes.");  }  } else {  //  // If we were unable to write the entire response out, then  // insert in Selector queue.  // 如果一次性返回不了全部数据给 Client 端 则将当前 RpcCall 重新塞入 response 阻塞队列 后续数据由 Responder 线程处理  call.connection.responseQueue.addFirst(call);  if (inHandler) {  // set the serve time when the response has to be sent later  call.timestampNanos = Time.monotonicNowNanos();  // 累计 pending 加 1 后续 Responder 线程进行处理  incPending();  try {  // Wakeup the thread blocked on select, only then can the call  // to channel.register() complete.  // 唤醒 Responder 线程的 NIO Selector  writeSelector.wakeup();  // 将当前 Rpc Request 的 SocketChannel 注册 OP\_WRITE 事件到 Responder 线程的 NIO Selector  channel.register(writeSelector, SelectionKey.OP\_WRITE, call);  } catch (ClosedChannelException e) {  //Its ok. channel might be closed else where.  done = true;  } finally {  // pending 减 1 并唤醒 Responder 线程  decPending();  }  }  if (LOG.isDebugEnabled()) {  LOG.debug(Thread.currentThread().getName() + ": responding to " + call  + " Wrote partial " + numBytes + " bytes.");  }  }  error = false; // everything went off well  }  } finally {  if (error && call != null) {  LOG.warn(Thread.currentThread().getName() + ", call " + call + ": output error");  done = true; // error. no more data for this channel.  closeConnection(call.connection);  }  }  return done;  } |

##### 1.3.4.4 Responder 线程处理 Rpc Request 请求结果给 Client 端

备注：Responder.run()

|  |
| --- |
| private void doRunLoop() {  long lastPurgeTimeNanos = 0; // last check for old calls.  while (running) {  try {  // 阻塞等待 pending > 0  waitPending(); // If a channel is being registered, wait.  // 监听 OP\_WRITE 事件  writeSelector.select(  TimeUnit.NANOSECONDS.toMillis(PURGE\_INTERVAL\_NANOS));  Iterator<SelectionKey> iter = writeSelector.selectedKeys().iterator();  while (iter.hasNext()) {  SelectionKey key = iter.next();  iter.remove();  try {  if (key.isWritable()) {  // 将 Rpc Request 请求结果返回给 Client 端  doAsyncWrite(key);  }  } catch (CancelledKeyException cke) {  // something else closed the connection, ex. reader or the  // listener doing an idle scan. ignore it and let them clean  // up  RpcCall call = (RpcCall) key.attachment();  if (call != null) {  LOG.info(Thread.currentThread().getName() +  ": connection aborted from " + call.connection);  }  } catch (IOException e) {  LOG.info(Thread.currentThread().getName() + ": doAsyncWrite threw exception " + e);  }  }  long nowNanos = Time.monotonicNowNanos();  if (nowNanos < lastPurgeTimeNanos + PURGE\_INTERVAL\_NANOS) {  continue;  }  lastPurgeTimeNanos = nowNanos;  //  // If there were some calls that have not been sent out for a  // long time, discard them.  //  if (LOG.isDebugEnabled()) {  LOG.debug("Checking for old call responses.");  }  ArrayList<RpcCall> calls;  // get the list of channels from list of keys.  synchronized (writeSelector.keys()) {  calls = new ArrayList<RpcCall>(writeSelector.keys().size());  iter = writeSelector.keys().iterator();  while (iter.hasNext()) {  SelectionKey key = iter.next();  RpcCall call = (RpcCall) key.attachment();  if (call != null && key.channel() == call.connection.channel) {  calls.add(call);  }  }  }  for (RpcCall call : calls) {  doPurge(call, nowNanos);  }  } catch (OutOfMemoryError e) {  //  // we can run out of memory if we have too many threads  // log the event and sleep for a minute and give  // some thread(s) a chance to finish  //  LOG.warn("Out of Memory in server select", e);  try {  Thread.sleep(60000);  } catch (Exception ie) {  }  } catch (Exception e) {  LOG.warn("Exception in Responder", e);  }  }  } |

###### 1.3.4.4.1 执行 Rpc Request 请求返回结果给 Client 端核心业务逻辑

|  |
| --- |
| private void doAsyncWrite(SelectionKey key) throws IOException {  // 获取 SelectionKey 绑定的 RpcCall 对象  RpcCall call = (RpcCall) key.attachment();  if (call == null) {  return;  }  if (key.channel() != call.connection.channel) {  throw new IOException("doAsyncWrite: bad channel");  }  synchronized (call.connection.responseQueue) {  // 返回数据给 Client 端  if (processResponse(call.connection.responseQueue, false)) {  try {  // 成功写完数据给 Client 端 恢复 interestOps 为 0  key.interestOps(0);  } catch (CancelledKeyException e) {  /\* The Listener/reader might have closed the socket.  \* We don't explicitly cancel the key, so not sure if this will  \* ever fire.  \* This warning could be removed.  \*/  LOG.warn("Exception while changing ops : " + e);  }  }  }  } |

如果还是没有完全返回数据给 Client 端 则循环 (但是不在注册 OP\_WRITE 事件)

|  |
| --- |
| // Processes one response. Returns true if there are no more pending  // data for this channel.  //  private boolean processResponse(LinkedList<RpcCall> responseQueue,  boolean inHandler) throws IOException {  boolean error = true;  boolean done = false; // there is more data for this channel.  int numElements = 0;  RpcCall call = null;  try {  synchronized (responseQueue) {  //  // If there are no items for this channel, then we are done  //  numElements = responseQueue.size();  if (numElements == 0) {  error = false;  return true; // no more data for this channel.  }  //  // Extract the first call  // 弹出 RpcCall 对象 以便后续返回 Rpc 结果给 Client 端  call = responseQueue.removeFirst();  // 获取当前 Rpc Request SocketChannel 通道  SocketChannel channel = call.connection.channel;  if (LOG.isDebugEnabled()) {  LOG.debug(Thread.currentThread().getName() + ": responding to " + call);  }  //  // Send as much data as we can in the non-blocking fashion  // 执行将 Rpc Request 的结果返回给 Client 端  int numBytes = channelWrite(channel, call.rpcResponse);  if (numBytes < 0) {  return true;  }  if (!call.rpcResponse.hasRemaining()) {  //Clear out the response buffer so it can be collected  call.rpcResponse = null;  call.connection.decRpcCount();  if (numElements == 1) { // last call fully processes.  done = true; // no more data for this channel.  } else {  done = false; // more calls pending to be sent.  }  if (LOG.isDebugEnabled()) {  LOG.debug(Thread.currentThread().getName() + ": responding to " + call  + " Wrote " + numBytes + " bytes.");  }  } else {  //  // If we were unable to write the entire response out, then  // insert in Selector queue.  // 如果一次性返回不了全部数据给 Client 端 则将当前 RpcCall 重新塞入 response 阻塞队列 后续数据由 Responder 线程处理  call.connection.responseQueue.addFirst(call);  if (inHandler) {  // set the serve time when the response has to be sent later  call.timestampNanos = Time.monotonicNowNanos();  // 累计 pending 加 1 后续 Responder 线程进行处理  incPending();  try {  // Wakeup the thread blocked on select, only then can the call  // to channel.register() complete.  // 唤醒 Responder 线程的 NIO Selector  writeSelector.wakeup();  // 将当前 Rpc Request 的 SocketChannel 注册 OP\_WRITE 事件到 Responder 线程的 NIO Selector  channel.register(writeSelector, SelectionKey.OP\_WRITE, call);  } catch (ClosedChannelException e) {  //Its ok. channel might be closed else where.  done = true;  } finally {  // pending 减 1  decPending();  }  }  if (LOG.isDebugEnabled()) {  LOG.debug(Thread.currentThread().getName() + ": responding to " + call  + " Wrote partial " + numBytes + " bytes.");  }  }  error = false; // everything went off well  }  } finally {  if (error && call != null) {  LOG.warn(Thread.currentThread().getName() + ", call " + call + ": output error");  done = true; // error. no more data for this channel.  closeConnection(call.connection);  }  }  return done;  } |

### 1.4 rpc-client 源码剖析

备注：RPC.getProxy()

|  |
| --- |
| public static <T> T getProxy(Class<T> protocol,  long clientVersion,  InetSocketAddress addr,  Configuration conf)  throws IOException {  // 获取协议代理 ProtocolProxy  return getProtocolProxy(protocol, clientVersion, addr, conf)  .getProxy();  } |

#### 1.4.1 获取协议代理 ProtocolProxy 对象

|  |
| --- |
| public static <T> ProtocolProxy<T> getProtocolProxy(Class<T> protocol,  long clientVersion,  InetSocketAddress addr, Configuration conf)  throws IOException {  // Client 端获取协议代理  return getProtocolProxy(protocol, clientVersion, addr, conf,  // 创建 StandardSocketFactory 对象  NetUtils.getDefaultSocketFactory(conf));  } |

|  |
| --- |
| public static <T> ProtocolProxy<T> getProtocolProxy(Class<T> protocol,  long clientVersion,  InetSocketAddress addr,  UserGroupInformation ticket,  Configuration conf,  SocketFactory factory,  int rpcTimeout,  RetryPolicy connectionRetryPolicy,  AtomicBoolean fallbackToSimpleAuth)  throws IOException {  if (UserGroupInformation.isSecurityEnabled()) {  SaslRpcServer.init(conf);  }  // 通过反射的方式创建 ProtobufRpcEngine 对象  return getProtocolEngine(protocol, conf)  // 获取 ProtocolProxy 对象  .getProxy(protocol, clientVersion,  addr, ticket, conf, factory, rpcTimeout, connectionRetryPolicy,  fallbackToSimpleAuth, null);  } |

备注：ProtobufRpcEngine.getProxy()

|  |
| --- |
| public <T> ProtocolProxy<T> getProxy(Class<T> protocol, long clientVersion,  InetSocketAddress addr, UserGroupInformation ticket, Configuration conf,  SocketFactory factory, int rpcTimeout, RetryPolicy connectionRetryPolicy,  AtomicBoolean fallbackToSimpleAuth, AlignmentContext alignmentContext)  throws IOException {  // 创建 Invoker 对象  final Invoker invoker = new Invoker(protocol, addr, ticket, conf, factory,  rpcTimeout, connectionRetryPolicy, fallbackToSimpleAuth,  alignmentContext);  // 创建 ProtocolProxy 对象  return new ProtocolProxy<T>(protocol,  // JDK 动态代理标配操作  (T) Proxy.newProxyInstance(  protocol.getClassLoader(), new Class[]{protocol}, invoker),  false);  } |

|  |
| --- |
| public ProtocolProxy(Class<T> protocol, T proxy,  boolean supportServerMethodCheck) {  // 协议接口类  this.protocol = protocol;  // 协议接口代理对象  this.proxy = proxy;  // 默认 false  this.supportServerMethodCheck = supportServerMethodCheck;  } |

#### 1.4.2 调用协议接口方法执行 RPC 请求调用 Invoker.invoke()

|  |
| --- |
| @Override  public Message invoke(Object proxy, final Method method, Object[] args)  throws ServiceException {  long startTime = 0;  if (LOG.isDebugEnabled()) {  startTime = Time.now();  }  if (args.length != 2) { // RpcController + Message  throw new ServiceException(  "Too many or few parameters for request. Method: ["  + method.getName() + "]" + ", Expected: 2, Actual: "  + args.length);  }  if (args[1] == null) {  throw new ServiceException("null param while calling Method: ["  + method.getName() + "]");  }  // if Tracing is on then start a new span for this rpc.  // guard it in the if statement to make sure there isn't  // any extra string manipulation.  Tracer tracer = Tracer.curThreadTracer();  TraceScope traceScope = null;  if (tracer != null) {  traceScope = tracer.newScope(RpcClientUtil.methodToTraceString(method));  }  // 构建 Rpc Request 头部 Header 信息 (执行方法名称、协议接口名称、协议版本)  RequestHeaderProto rpcRequestHeader = constructRpcRequestHeader(method);  if (LOG.isTraceEnabled()) {  LOG.trace(Thread.currentThread().getId() + ": Call -> " +  remoteId + ": " + method.getName() +  " {" + TextFormat.shortDebugString((Message) args[1]) + "}");  }  // 请求参数强转为 Message 对象  final Message theRequest = (Message) args[1];  final RpcWritable.Buffer val;  try {  // 执行 Rpc Request  val = (RpcWritable.Buffer) client.call(  RPC.RpcKind.RPC\_PROTOCOL\_BUFFER,  // 创建 Rpc Request RpcProtobufRequest 对象  new RpcProtobufRequest(rpcRequestHeader, theRequest),  remoteId, fallbackToSimpleAuth, alignmentContext);  } catch (Throwable e) {  if (LOG.isTraceEnabled()) {  LOG.trace(Thread.currentThread().getId() + ": Exception <- " +  remoteId + ": " + method.getName() +  " {" + e + "}");  }  if (traceScope != null) {  traceScope.addTimelineAnnotation("Call got exception: " +  e.toString());  }  throw new ServiceException(e);  } finally {  if (traceScope != null) traceScope.close();  }  if (LOG.isDebugEnabled()) {  long callTime = Time.now() - startTime;  LOG.debug("Call: " + method.getName() + " took " + callTime + "ms");  }  // 如果 Client 端异步发送 Rpc Request 则异步处理返回结果  if (Client.isAsynchronousMode()) {  final AsyncGet<RpcWritable.Buffer, IOException> arr  = Client.getAsyncRpcResponse();  final AsyncGet<Message, Exception> asyncGet  = new AsyncGet<Message, Exception>() {  @Override  public Message get(long timeout, TimeUnit unit) throws Exception {  return getReturnMessage(method, arr.get(timeout, unit));  }  @Override  public boolean isDone() {  return arr.isDone();  }  };  ASYNC\_RETURN\_MESSAGE.set(asyncGet);  return null;  } else {  return getReturnMessage(method, val);  }  } |

|  |
| --- |
| Writable call(RPC.RpcKind rpcKind, Writable rpcRequest,  ConnectionId remoteId, int serviceClass,  AtomicBoolean fallbackToSimpleAuth, AlignmentContext alignmentContext)  throws IOException {  // 创建 Call 对象  final Call call = createCall(rpcKind, rpcRequest);  call.setAlignmentContext(alignmentContext);  // 创建 Connection 对象 (继承 Thread 类)  final Connection connection = getConnection(remoteId, call, serviceClass,  fallbackToSimpleAuth);  try {  checkAsyncCall();  try {  // 发送 Rpc Request  connection.sendRpcRequest(call); // send the rpc request  } catch (RejectedExecutionException e) {  throw new IOException("connection has been closed", e);  } catch (InterruptedException ie) {  Thread.currentThread().interrupt();  IOException ioe = new InterruptedIOException(  "Interrupted waiting to send RPC request to server");  ioe.initCause(ie);  throw ioe;  }  } catch (Exception e) {  if (isAsynchronousMode()) {  releaseAsyncCall();  }  throw e;  }  // 判断是否异步处理 Rpc Request 返回接口  if (isAsynchronousMode()) {  final AsyncGet<Writable, IOException> asyncGet  = new AsyncGet<Writable, IOException>() {  @Override  public Writable get(long timeout, TimeUnit unit)  throws IOException, TimeoutException {  boolean done = true;  try {  // 异步处理返回结果  final Writable w = getRpcResponse(call, connection, timeout, unit);  if (w == null) {  done = false;  throw new TimeoutException(call + " timed out "  + timeout + " " + unit);  }  return w;  } finally {  if (done) {  releaseAsyncCall();  }  }  }  @Override  public boolean isDone() {  synchronized (call) {  return call.done;  }  }  };  ASYNC\_RPC\_RESPONSE.set(asyncGet);  return null;  } else {  // 同步处理返回结果 (阻塞等待 将由 Connection.run() 接收到数据后唤醒)  return getRpcResponse(call, connection, -1, null);  }  } |

|  |
| --- |
| private Connection getConnection(ConnectionId remoteId,  Call call, int serviceClass, AtomicBoolean fallbackToSimpleAuth)  throws IOException {  if (!running.get()) {  // the client is stopped  throw new IOException("The client is stopped");  }  Connection connection;  /\* we could avoid this allocation for each RPC by having a  \* connectionsId object and with set() method. We need to manage the  \* refs for keys in HashMap properly. For now its ok.  \*/  while (true) {  // These lines below can be shorten with computeIfAbsent in Java8  connection = connections.get(remoteId);  if (connection == null) {  // 创建 Connection 对象  connection = new Connection(remoteId, serviceClass);  Connection existing = connections.putIfAbsent(remoteId, connection);  if (existing != null) {  connection = existing;  }  }  if (connection.addCall(call)) {  break;  } else {  // This connection is closed, should be removed. But other thread could  // have already known this closedConnection, and replace it with a new  // connection. So we should call conditional remove to make sure we only  // remove this closedConnection.  connections.remove(remoteId, connection);  }  }  // If the server happens to be slow, the method below will take longer to  // establish a connection.  // 跟 Server 端建立连接  connection.setupIOstreams(fallbackToSimpleAuth);  return connection;  } |

|  |
| --- |
| private synchronized void setupIOstreams(  AtomicBoolean fallbackToSimpleAuth) {  if (socket != null || shouldCloseConnection.get()) {  return;  }  UserGroupInformation ticket = remoteId.getTicket();  if (ticket != null) {  final UserGroupInformation realUser = ticket.getRealUser();  if (realUser != null) {  ticket = realUser;  }  }  try {  connectingThread.set(Thread.currentThread());  if (LOG.isDebugEnabled()) {  LOG.debug("Connecting to " + server);  }  Span span = Tracer.getCurrentSpan();  if (span != null) {  span.addTimelineAnnotation("IPC client connecting to " + server);  }  short numRetries = 0;  Random rand = null;  while (true) {  // 启动 Client 端 SocketChannel 去连接 Server 端 ServerSocketChannel  setupConnection(ticket);  // 创建 IpcStreams 对象  ipcStreams = new IpcStreams(socket, maxResponseLength);  // 往 IpcStreams 流写入 Rpc Request 的头部信息  writeConnectionHeader(ipcStreams);  if (authProtocol == AuthProtocol.SASL) {  try {  authMethod = ticket  .doAs(new PrivilegedExceptionAction<AuthMethod>() {  @Override  public AuthMethod run()  throws IOException, InterruptedException {  return setupSaslConnection(ipcStreams);  }  });  } catch (IOException ex) {  if (saslRpcClient == null) {  // whatever happened -it can't be handled, so rethrow  throw ex;  }  // otherwise, assume a connection problem  authMethod = saslRpcClient.getAuthMethod();  if (rand == null) {  rand = new Random();  }  handleSaslConnectionFailure(numRetries++, maxRetriesOnSasl, ex,  rand, ticket);  continue;  }  if (authMethod != AuthMethod.SIMPLE) {  // Sasl connect is successful. Let's set up Sasl i/o streams.  ipcStreams.setSaslClient(saslRpcClient);  // for testing  remoteId.saslQop =  (String) saslRpcClient.getNegotiatedProperty(Sasl.QOP);  LOG.debug("Negotiated QOP is :" + remoteId.saslQop);  if (fallbackToSimpleAuth != null) {  fallbackToSimpleAuth.set(false);  }  } else if (UserGroupInformation.isSecurityEnabled()) {  if (!fallbackAllowed) {  throw new IOException("Server asks us to fall back to SIMPLE " +  "auth, but this client is configured to only allow secure " +  "connections.");  }  if (fallbackToSimpleAuth != null) {  fallbackToSimpleAuth.set(true);  }  }  }  if (doPing) {  ipcStreams.setInputStream(new PingInputStream(ipcStreams.in));  }  // 往 IpcStreams 流写入当前 Connection 上下文信息  writeConnectionContext(remoteId, authMethod);  // update last activity time  touch();  span = Tracer.getCurrentSpan();  if (span != null) {  span.addTimelineAnnotation("IPC client connected to " + server);  }  // start the receiver thread after the socket connection has been set  // up  // 调用 Connection.run()  start();  return;  }  } catch (Throwable t) {  if (t instanceof IOException) {  markClosed((IOException) t);  } else {  markClosed(new IOException("Couldn't set up IO streams: " + t, t));  }  close();  } finally {  connectingThread.set(null);  }  } |

|  |
| --- |
| public void run() {  if (LOG.isDebugEnabled())  LOG.debug(getName() + ": starting, having connections "  + connections.size());  try {  while (waitForWork()) {//wait here for work - read or close connection  // 等待 Rpc Request 结果返回  receiveRpcResponse();  }  } catch (Throwable t) {  // This truly is unexpected, since we catch IOException in receiveResponse  // -- this is only to be really sure that we don't leave a client hanging  // forever.  LOG.warn("Unexpected error reading responses on connection " + this, t);  markClosed(new IOException("Error reading responses", t));  }  close();  if (LOG.isDebugEnabled())  LOG.debug(getName() + ": stopped, remaining connections "  + connections.size());  } |
| /\* Receive a response.  \* Because only one receiver, so no synchronization on in.  \*/  private void receiveRpcResponse() {  if (shouldCloseConnection.get()) {  return;  }  touch();  try {  // 读取 Rpc Request 返回数据  ByteBuffer bb = ipcStreams.readResponse();  RpcWritable.Buffer packet = RpcWritable.Buffer.wrap(bb);  RpcResponseHeaderProto header =  packet.getValue(RpcResponseHeaderProto.getDefaultInstance());  checkResponse(header);  int callId = header.getCallId();  if (LOG.isDebugEnabled())  LOG.debug(getName() + " got value #" + callId);  RpcStatusProto status = header.getStatus();  if (status == RpcStatusProto.SUCCESS) {  Writable value = packet.newInstance(valueClass, conf);  final Call call = calls.remove(callId);  // 赋值操作  call.setRpcResponse(value);  if (call.alignmentContext != null) {  call.alignmentContext.receiveResponseState(header);  }  }  // verify that packet length was correct  if (packet.remaining() > 0) {  throw new RpcClientException("RPC response length mismatch");  }  if (status != RpcStatusProto.SUCCESS) { // Rpc Request failed  final String exceptionClassName = header.hasExceptionClassName() ?  header.getExceptionClassName() :  "ServerDidNotSetExceptionClassName";  final String errorMsg = header.hasErrorMsg() ?  header.getErrorMsg() : "ServerDidNotSetErrorMsg";  final RpcErrorCodeProto erCode =  (header.hasErrorDetail() ? header.getErrorDetail() : null);  if (erCode == null) {  LOG.warn("Detailed error code not set by server on rpc error");  }  RemoteException re = new RemoteException(exceptionClassName, errorMsg, erCode);  if (status == RpcStatusProto.ERROR) {  final Call call = calls.remove(callId);  call.setException(re);  } else if (status == RpcStatusProto.FATAL) {  // Close the connection  markClosed(re);  }  }  } catch (IOException e) {  markClosed(e);  }  } |