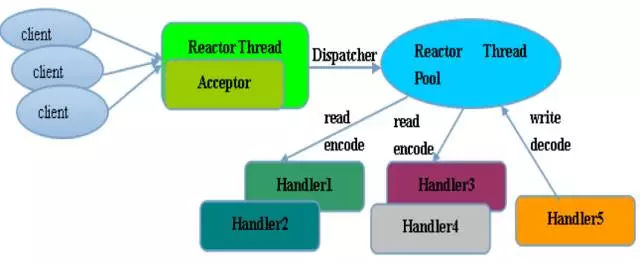
Hadoop Reactor模型

目前主流的Java网络程序基本上是在Java NIO的基础上实现Reactor模型，隐藏NIO底层的复杂细节，大大简化了NIO编程，其原理如下图所示：



1. Acceptor(Main Reactor)，负责接收客户端Socket发起的新建连接请求，并把该Socket绑定到Sub Reacor线程成
2. 客户端Socket随后的读写事件都有Reactor线程来处理
3. Reactor线程读取数据，对数据进行预处理(编解码成Java POJO)后交给程序的Handler来实现特定的业务逻辑处理。为了不影响Reactor线程，通常使用单独的线程池来异步执行Handler接口方法

在Hadoop中RPC通信的核心逻辑在ipc.Server中定义实现，ipc.Server采用了很多提高高并发处理能力的技术，主要包括线程池、事件驱动及Reactor设计模式等，这些技术均采用JDK自带的库实现。其中Reactor模型的实现结构图如下图所示：



# Listener

Listener在Hadoop Reactor中是Main Reactor的角色，该线程监听来自客户端的连接，通过多路复用Selector模式，处理Accept事件，其初始化核心代码分析：

*address = new InetSocketAddress(bindAddress, port);*

*// 创建ServerSocketChannel，客户端连接端口，默认nn的端口为8020*

*acceptChannel = ServerSocketChannel.open();*

*acceptChannel.configureBlocking(false); //配置非阻塞模式*

*// 绑定主机及端口*

*bind(acceptChannel.socket(), address, backlogLength, conf, portRangeConfig);*

*port = acceptChannel.socket().getLocalPort(); //Could be an ephemeral port*

*// 创建Selector*

*selector= Selector.open();*

*// 将Server Socket注册到Selector上，并监听OP\_ACCEPT*

*acceptChannel.register(selector, SelectionKey.OP\_ACCEPT);*

Listener线程在启动后，不停通过Selector获取客户端连接，并将其dispatch到Reader进行下一步处理，线程执行流程如下：

*while (running) {*

*SelectionKey key = null;*

*try {*

*getSelector().select();*

*Iterator<SelectionKey> iter = getSelector().selectedKeys().iterator();*

*while (iter.hasNext()) {*

*key = iter.next();*

*iter.remove();*

*try {*

*if (key.isValid()) {*

*if (key.isAcceptable()) //接收到OP\_ACCEPT客户端*

*doAccept(key); //，通过doAccept进行dispatch*

*}*

*}*

*key = null;*

*}*

*}*

在doAccept中核心逻辑是将SelectionKey对应的SocketChannel转发给Reader进行处理，核心逻辑如下：

*void doAccept(SelectionKey key)*

*ServerSocketChannel server = (ServerSocketChannel) key.channel();*

*SocketChannel channel;*

*while ((channel = server.accept()) != null) {*

*channel.configureBlocking(false);*

*channel.socket().setTcpNoDelay(tcpNoDelay);*

*channel.socket().setKeepAlive(true);*

currentReader =

(currentReader + 1) % readers.length;

return readers[currentReader];

*Reader reader = getReader();*

*//轮算算法，获取Reader*

*Connection c = connectionManager.register(channel);*

*//将SocketChannel封装到Connection中，并添加给Reader*

*key.attach(c); // 将Connection附加到相对应的SelectionKey上*

*reader.addConnection(c);*

*}*

# Reader

Listener接收到客户端连接请求后建立连接，将连接加入到Reader缓冲队列，然后使用Reader处理此连接上的数据请求，连接添加到Reader如下

*public void addConnection(Connection conn) throws InterruptedException {*

*pendingConnections.put(conn);*

*readSelector.wakeup();*

*}*

Reader是属于Sub Reactor Thread，其执行如下：

*private synchronized void doRunLoop() {*

*while (running) {*

*SelectionKey key = null;*

*try {*

*int size = pendingConnections.size();*

*for (int i=size; i>0; i--) {*

*Connection conn = pendingConnections.take();*

*//注册SocketChannel，到readSelector中，并监听OP\_READ*

*conn.channel.register(readSelector, SelectionKey.OP\_READ, conn);*

*}*

*readSelector.select();*

*//通过ReadSelector，获取OP\_READ事件，可以从Channel中读取数据*

*Iterator<SelectionKey> iter = readSelector.selectedKeys().iterator();*

*while (iter.hasNext()) {*

*key = iter.next();*

*iter.remove();*

*try {*

*if (key.isReadable()) {*

*doRead(key); //读取数据*

*}*

*}*

*}*

在Reader中，通过doRead从SocketChannel中读取数据

*void doRead(SelectionKey key) throws InterruptedException {*

*int count;*

*Connection c = (Connection)key.attachment();*

*count = c.readAndProcess();*

*}*

*Connection.readAndProcess，读取数据并处理*

*public int readAndProcess() throws IOException, InterruptedException {*

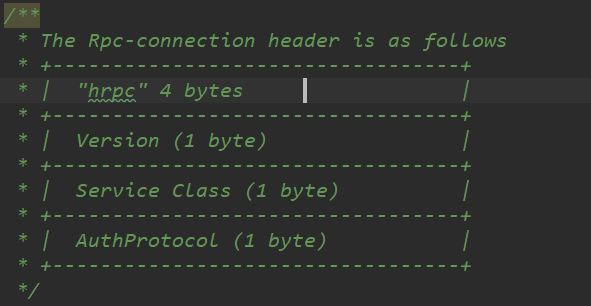
*while (!shouldClose()) { //读取RPC Request Header数据，其包括以下字段：*

*if (!connectionHeaderRead)*

*if (connectionHeaderBuf == null) {*

*connectionHeaderBuf = ByteBuffer.allocate(HEADER\_LEN\_AFTER\_HRPC\_PART);*

*}*



*count = channelRead(channel, connectionHeaderBuf);*

*....*

*}*

*//读取RPC Request 数据*

*if (data == null) { // just read 4 bytes - length of RPC packet*

*data = ByteBuffer.allocate(dataLength);*

*}*

*count = channelRead(channel, data);*

*if (data.remaining() == 0) {*

*ByteBuffer requestData = data;*

*data = null; // null out in case processOneRpc throws.*

*boolean isHeaderRead = connectionContextRead;*

*processOneRpc(requestData); //questData的处理*

*}*

*}*

将请求封装成RpcCall后，写入到CallQueue中

*RpcCall call = new RpcCall(this, header.getCallId(),*

*header.getRetryCount(), rpcRequest,*

*ProtoUtil.convert(header.getRpcKind()),*

*header.getClientId().toByteArray(), traceScope, callerContext);*

*internalQueueCall(call);*

Server为了提高并发连接能力，启动多个Reader线程，其配置参数为：

*public static final String IPC\_SERVER\_RPC\_READ\_THREADS\_KEY =  
 "ipc.server.read.threadpool.size";  
/\*\* Default value for IPC\_SERVER\_RPC\_READ\_THREADS\_KEY \*/  
public static final int IPC\_SERVER\_RPC\_READ\_THREADS\_DEFAULT = 1;*

默认值是1

# Handler

根据RPC请求中的方法及参数，来调用相应的业务逻辑接口来处理请求。一个Server中有多个Handler，对应多个业务接口。Handler从callQueue中取出Call对象，然后调用Server.call方法进行处理。Handler线程数量，以NN为例，其参数为：

*public static final String DFS\_NAMENODE\_HANDLER\_COUNT\_KEY = "dfs.namenode.handler.count";*

*public static final int DFS\_NAMENODE\_HANDLER\_COUNT\_DEFAULT = 10;*

Handler从CallQueue中取出RpcCall并进行处理，其执行如下：

*public void run() {*

*while (running) {*

*try {*

*final Call call = callQueue.take(); // pop the queue; maybe blocked here*

*CurCall.set(call);*

*UserGroupInformation remoteUser = call.getRemoteUser();*

*if (remoteUser != null) {*

*remoteUser.doAs(call);*

*} else {*

*call.run();*

*}*

*}*

*}*

RpcCall的成员变量如下：

*final Connection connection; // connection to client*

*final Writable rpcRequest; // Serialized Rpc request from client*

*ByteBuffer rpcResponse;*

RpcCall的执行如下：

*public Writable call(RPC.Server server, String connectionProtocolName,*

*Writable writableRequest, long receiveTime) throws Exception {*

*RpcProtobufRequest request = (RpcProtobufRequest) writableRequest;*

*RequestHeaderProto rpcRequest = request.getRequestHeader();*

*String methodName = rpcRequest.getMethodName(); //获取调用方法名*

*String declaringClassProtoName = rpcRequest.getDeclaringClassProtocolName();*

*ProtoClassProtoImpl protocolImpl = getProtocolImpl(server,*

*declaringClassProtoName, clientVersion);*

*BlockingService service = (BlockingService) protocolImpl.protocolImpl; //调用协议*

*MethodDescriptor methodDescriptor = service.getDescriptorForType()*

*.findMethodByName(methodName);*

*Message prototype = service.getRequestPrototype(methodDescriptor);*

*Message param = request.getValue(prototype);*

*Message result;*

*long startTime = Time.now();*

*int qTime = (int) (startTime - receiveTime);*

*Exception exception = null;*

*boolean isDeferred = false;*

*try {*

*server.rpcDetailedMetrics.init(protocolImpl.protocolClass);*

*currentCallInfo.set(new CallInfo(server, methodName));*

*result = service.callBlockingMethod(methodDescriptor, null, param); //执行*

*if (currentCallback.get() != null) {*

*Server.getCurCall().get().deferResponse();*

*isDeferred = true;*

*currentCallback.set(null);*

*return null;*

*}*

*return RpcWritable.wrap(result); //返回结果*

*}*

获取结果后，将其发送到Responder，然后发送给客户端

*Writable value = null;*

*ResponseParams responseParams = new ResponseParams();*

*value = call(rpcKind, connection.protocolName, rpcRequest, timestamp);*

*if (!isResponseDeferred()) {*

*setupResponse(this, responseParams.returnStatus,responseParams.detailedErr,*

*value, responseParams.errorClass, responseParams.error);*

*sendResponse();*

*}*

将结果存储到具体的Connection的ResponseQueue中，如下所示：

*void doResponse(Throwable t) throws IOException {*

*RpcCall call = this;*

*if (t != null) {*

*call = new RpcCall(this);*

*setupResponse(call,*

*RpcStatusProto.FATAL, RpcErrorCodeProto.ERROR\_RPC\_SERVER,*

*null, t.getClass().getName(), StringUtils.stringifyException(t));*

*}*

*connection.sendResponse(call); => responder.doRespond(call);*

*=> call.connection.responseQueue.addLast(call);*

*}*

# Responder

经过上述步骤，已经生成了相应结果，最后一步是将结果发送给Responder。在Server中只有1个Responder线程，当Handler调用doRespond方法后，handler的处理结果被加入到responseQueue队列中，Responser线程采用异步方式将结果发送给客户端，其核心执行流程如下：

*private boolean processResponse(LinkedList<RpcCall> responseQueue,*

*boolean inHandler) throws IOException {*

*RpcCall call = null;*

*try {*

*synchronized (responseQueue) {*

*numElements = responseQueue.size();*

*call = responseQueue.removeFirst(); //获取Call的SocketChannel*

*SocketChannel channel = call.connection.channel;*

*int numBytes = channelWrite(channel, call.rpcResponse);*

*if (numBytes < 0) {*

*return true;*

*}*

*......*

*} else {*

*call.connection.responseQueue.addFirst(call);*

*if (inHandler) {*

*call.timestamp = Time.now();*

*incPending();*

*try {*

*writeSelector.wakeup(); //将Channel注册到WriterSelector中*

*channel.register(writeSelector, SelectionKey.OP\_WRITE, call);*

*}*

*return done;*

*}*