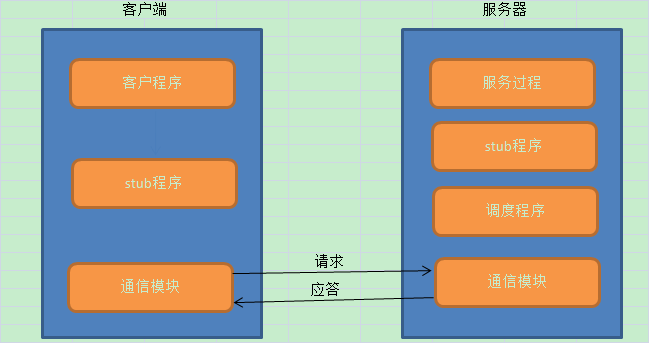
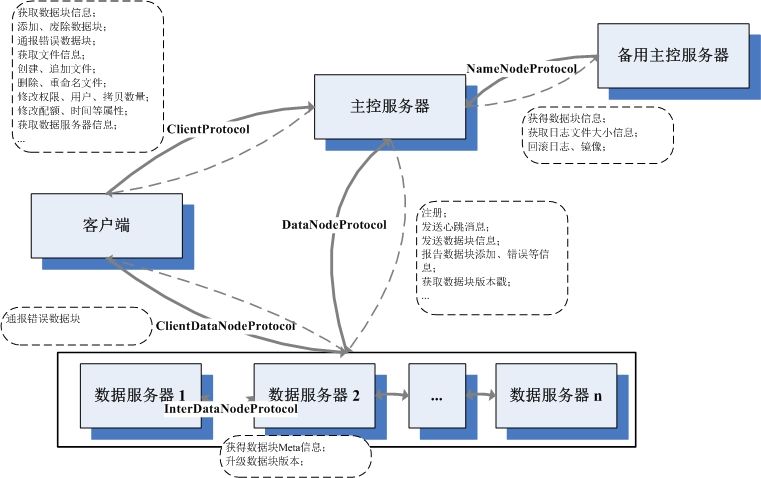
Hadoop RPC及优化

Hadoop的进程间交互是·通过RPC进行的，比如NameNode与Datanode的交互，RM与NM之间的交互。RPC的基本流程如下图所示：



* RPC采用C/S模式
* Client端发送一个带有参数的请求信息到Server
* Server接收到这个请求后，根据发送过来的参数调用相应的程序，然后把处理的结果发送给Client端
* Client端接收到结果后继续运行

Hadoop各组件之间的RPC交互系统框架如下图所示：



# Hadoop RPC使用示例

使用Hadoop RPC的四个步骤如下：

1. 定义RPC protocol，RPC协议是客户端和服务器端之间的通信接口，它定义了服务器端对外提供的服务接口
2. 实现RPC协议，RPC protocol通常只是一个Java接口，用户需要实现该接口
3. 构造和启动RPC Server，直接使用静态类Builder构造一个RPC Server，并调用函数start启动Server
4. 构造RPC Client并发送请求，使用静态方法getProxy构造客户端代理对象，通过代理对象调用远程端的方法。

## 1.1 定义RPC协议

定义一个IProxyProtocol通信接口，声明一个add方法，如下所示：

*import org.apache.hadoop.ipc.VersionedProtocol;*

*public interface IProxyProtocol extends VersionedProtocol {*

*static final long versionID = 23234L;*

*int add(int first, int second);*

*}*

Hadoop中所有自定义的RPC接口都需要继承VersionedProtocol接口，它描述了协议的版本信息。默认情况下，不同版本号的RPC Client和Server之间不同相互通信，因此客户端和服务端通过版本号标志。

## 1.2 实现RPC协议

对IProxyProtocol接口进行实现如下所示：

*public class MyProxy implements IProxyProtocol {*

*public int add(int first, int second) {*

*System.out.println("Invoke add function");*

*return first + second;*

*}*

*public long getProtocolVersion(String s, long l) throws IOException {*

*System.out.println("MyProxy.ProtocolVersio=" + IProxyProtocol.versionID);*

*return IProxyProtocol.versionID;*

*}*

*public ProtocolSignature getProtocolSignature(String s, long l, int i) throws IOException {*

*return null;*

*}}*

## 1.3 构造RPC Server并启动服务

通过RPC的静态方法getServer来获得Server对象，如下代码所示：

*import org.apache.hadoop.conf.Configuration;*

*import org.apache.hadoop.ipc.RPC;*

*import org.apache.hadoop.ipc.Server;*

*public class MyServer {*

*public static int port = 5432;*

*public static String ipaddr = "192.168.246.128";*

*public static void main(String[] args) throws Exception {*

*Configuration conf = new Configuration();*

*Server server = new RPC.Builder(conf)*

*.setProtocol(IProxyProtocol.class)*

*.setInstance(new MyProxy())*

*.setBindAddress(ipaddr)*

*.setPort(port)*

*.build();*

*server.start();*

*}}*

这段代码的核心在RPC.Builder中，首先设置的是服务端处理的Hadoop协议，第二个参数是被调用的Java对象，最后是主机和端口。获得服务器对象后，启动服务器。这样服务器在指定端口监听客户端的请求。

## 1.4 构造RPC Client并发出请求

这里使用静态方法getProxy构造客户端代理对象，直接通过代理对象调用远程端的方法，具体如下：

*import org.apache.hadoop.conf.Configuration;*

*import org.apache.hadoop.ipc.RPC;*

*import java.io.IOException;*

*import java.net.InetSocketAddress;*

*public class MyClient {*

*public static int port = 5432;*

*public static String ipaddr = "192.168.246.128";*

*public static void main(String[] args) throws IOException {*

*IProxyProtocol proxy = RPC.getProxy(*

*IProxyProtocol.class,*

*IProxyProtocol.versionID,*

*new InetSocketAddress(ipaddr,port),*

*new Configuration()*

*);*

*System.out.println("result:"+proxy.add(12, 45));*

*System.out.println("proxy class:"+proxy.getClass().getName());*

*RPC.stopProxy(proxy);*

*}}*

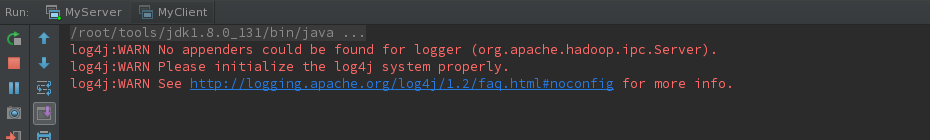
核心在于getProxy，该方法有四个参数

* 被调用的接口类
* 客户端版本号
* 服务端地址
* 配置

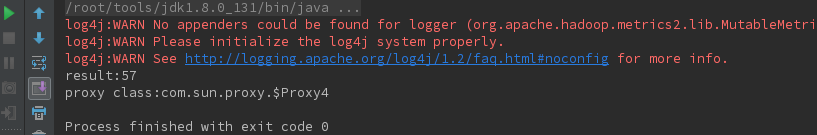
经过以上四步，Hadoop RPC搭建了高效的客户机-服务器网络模型。

## 1.5 查询运行结果

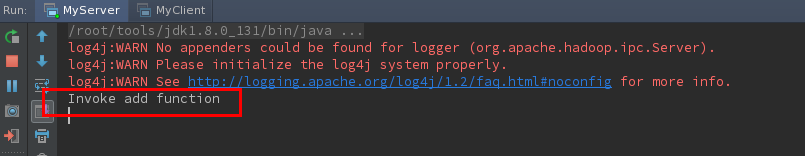
1）启动服务端，开始监听客户端请



2）启动客户端



3）服务端处理如下



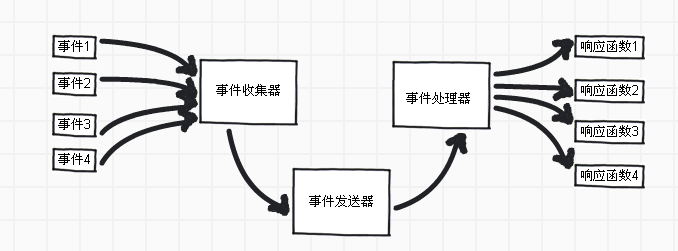
# Hadoop RPC分析

同其他RPC框架一样，Hadoop RPC分为四个部分：

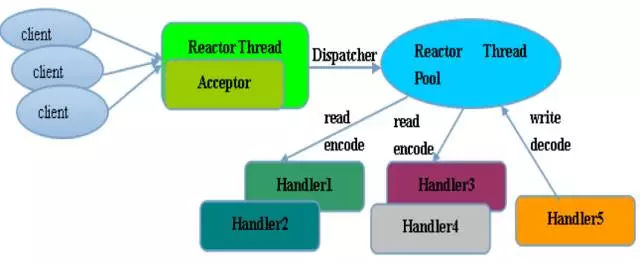
* 序列化层，Client与Server端通信传递的信息采用了Hadoop里提供的序列化类和自定义的Writable类型
* 函数调用层：Hadoop RPC采用动态代理以及Java反射实现函数调用
* 网络传输层：Hadoop RPC采用了基于TCP/IP的Socket机制
* 服务器端框架层，RPC Server利用Java NIO以及采用事件驱动的I/O模型，提高RPC Server的并发处理能力。

## 2.1 Reactor

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。Reactor是并发编程中一种基于事件驱动的设计模式，模式如下图所示：



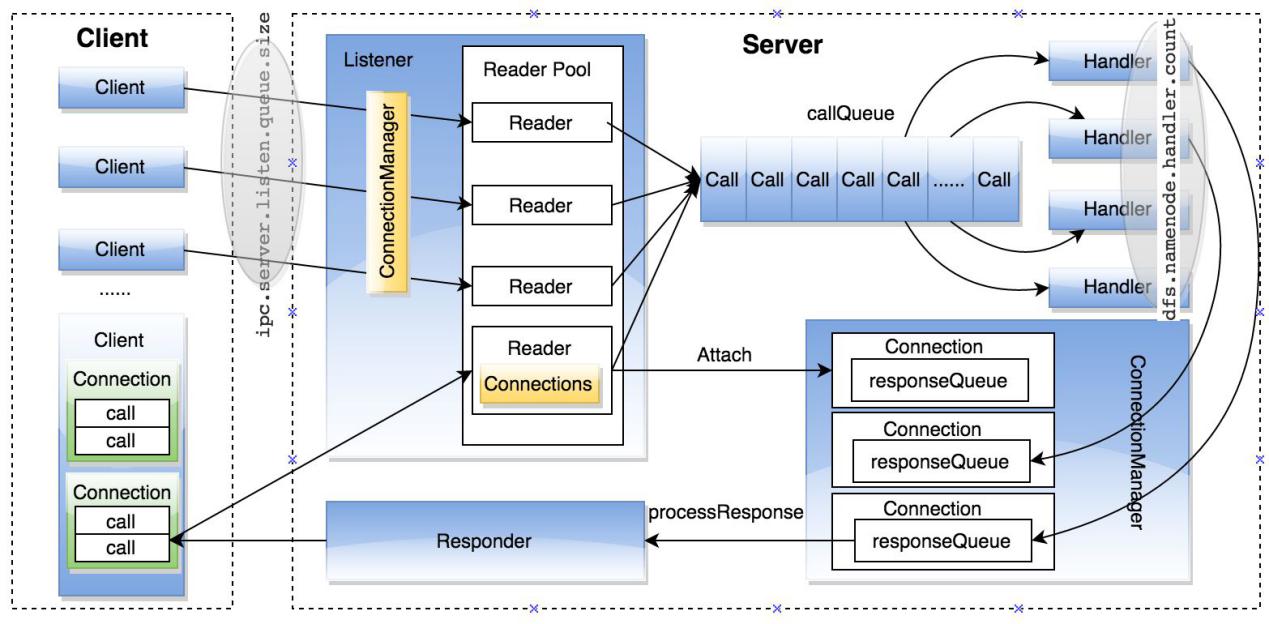
其中心思想是将所有要处理的事件注册到事件收集器上，然后通过事件发送器发送到事件处理器中。其实现模型如下图所示：



1. Reactor Thread，接收客户端发送过来的事件，然后将事件进行派发，相当于分发功能的Selector。
2. Acceptor接受Client连接，建立对应的handler，并行向Reactor注册此Handler
3. Handler，实现业务处理的逻辑

## 2.2 RPC总体框架

Hadoop RPC基于NIO实现，总体系统架构如下图所示：

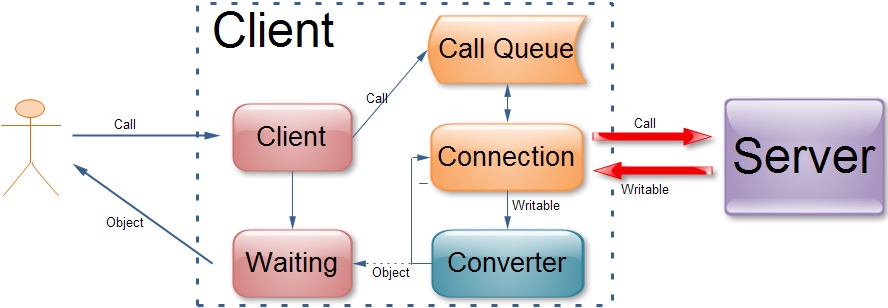


Hadoop RPC系统架构图

1. Listener，该线程用户监听来自客户端的链接，选用Selector模式处理Accept事件
2. 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。Server为了提高并发连接能力，启动多个Reader线程。
3. 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
4. Responder，一个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。
5. 客户端，在Client内部把每一个RPC连接Connection设计成后台线程，内部放置一个任务队列来存储待处理的RPC调用。

## 2.3 RPC Client

Client主要完成发送远程调用信息，并接受执行结果，流程图如下：



核心是将调用方法信息封装成Call，然后通过网络发送到服务端，并等待服务端的返回。

### 2.3.1 核心类

与Client相关的类如下所示：



1. Client类，RPC的核心对象：

*private ConcurrentMap<ConnectionId, Connection> connections =*

*new ConcurrentHashMap<>();*

*private Class<? extends Writable> valueClass; // class of call values*

*private AtomicBoolean running = new AtomicBoolean(true); // if client runs*

*final private Configuration conf;*

*private SocketFactory socketFactory; // how to create sockets*

Client每次与服务器的连接都会产生网络延迟，因此会占用大量的系统资源，因此在Client中设计了一个连接池，用来缓存与不同服务端的连接，通过对象ConnectionId来表示每一个连接。维护的连接相关参数：

|  |  |  |
| --- | --- | --- |
| 参数(ipc.client.xx) | 描述 | default(s) |
| connection.maxidletime | 连接的最大空闲期，超过该时间，则自动关闭与Server的链接 | 10 |
| connect.timeout | 设置客户端与服务器端的网络超时时间，超过该时间发生SocketTimeoutExcpetion | 20 |
| ping.interval | 自动向服务端发送ping包，测试客户端和服务端的链接是否正常 | 60 |
| connect.max.retries | 连接向服务器发起连接请求失败时，不断重新尝试，尝试次数为maxRetries | 10 |

当RPC结果返回后，Client需要将返回结果解析成用户真正需要的数据类型，所有通过valueClass来定义解析器，在WriteableRpcEngine中解析类型为ObjectWritable， 对于ProtobufRpcEngine使用的是解析类型为RpcWritable.Buffer。

1. Call类，维护的核心参数为：

*final int id; // call id*

*final int retry; // retry count*

*final Writable rpcRequest; // the serialized rpc request*

*Writable rpcResponse; // null if rpc has error*

*IOException error; // exception, null if success*

*final RPC.RpcKind rpcKind; // Rpc EngineKind*

*boolean done; // true when call is done*

*private final Object externalHandler;*

客户端的每一次RPC所涉及到的所有参数方法（方法名、输入参数、返回值）都被抽象到一个Call对象中，call中的rpcRequest属性包含了RPC调用的方法名和所有输出参数，rpcResponse是交互回调传回的结果。其中rpcRequest的具体类型为：Invocation，主要属性：

*private String methodName;*

*private Class<?>[] parameterClasses;*

*private Object[] parameters;*

*private Configuration conf;*

*private long clientVersion;*

*private int clientMethodsHash;*

*private String declaringClassProtocolName;*

1. ConnectionId类

*InetSocketAddress address;*

*UserGroupInformation ticket;*

*final Class<?> protocol;*

Client内部设计了一个RPC连接池，避免与服务器频繁的连接、关闭等操作，因此通过ConnectionID来标识一个RPC连接，主要通过服务地址、用户信息和协议类型来标识。

1. Connection类

*private InetSocketAddress server; // server ip:port*

*private final ConnectionId remoteId; // connection id*

*private AuthMethod authMethod; // authentication method*

*private AuthProtocol authProtocol;*

*private int serviceClass;*

*private SaslRpcClient saslRpcClient;*

*private Socket socket = null; // connected socket*

*private IpcStreams ipcStreams;*

在Client内部，每一个RPC连接Connection都被设计成后台线程，内部放置一个任务队列来存放待处理的RPC调用，当一个RPC连接的idle时间超过最大空闲值则自动关闭。为了保证客户端和服务器端底层通信协议的一致性，客户端在与服务器端建立网络连接之后马上向服务器端发送一个头部信息，以确保C/S两端所用的协议版本号是相同。

### 2.3.2 客户端启动

RPC客户端从调用RPC.getProtocolProxy开始，比如Datanode创建NameNode客户端，代码如下：

*ClientNamenodeProtocolPB proxy = RPC.getProtocolProxy(*

*ClientNamenodeProtocolPB.class, version, address, ugi, conf,*

*NetUtils.getDefaultSocketFactory(conf),*

*org.apache.hadoop.ipc.Client.getTimeout(conf), defaultPolicy,*

*fallbackToSimpleAuth).getProxy();*

getProtocolProxy方法用于返回一个协议接口的代理实例对象，在类定义中有多个变体（不同的参数组合）。getProtocolProxy最终调用RpcEngine中的getProxy方法：

*public <T> ProtocolProxy<T> getProxy(Class<T> protocol, long clientVersion,*

*InetSocketAddress addr, UserGroupInformation ticket,*

*Configuration conf, SocketFactory factory,*

*int rpcTimeout, RetryPolicy connectionRetryPolicy,*

*AtomicBoolean fallbackToSimpleAuth)throws IOException {*

*......*

*T proxy = (T) Proxy.newProxyInstance(protocol.getClassLoader(),*

*new Class[] { protocol }, new Invoker(protocol, addr, ticket, conf,*

*factory, rpcTimeout, fallbackToSimpleAuth));*

*return new ProtocolProxy<T>(protocol, proxy, true);*

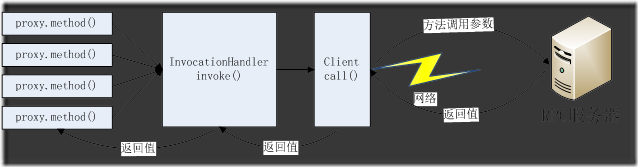
*}*

这里使用动态代理，调用Proxy.newProxyInstance方法生成一个protocol协议接口动态代理实例对象，这个代理对象代理protocol接口中的方法，而Invoker的构造参数仅有addr,ticket,conf和factory等。

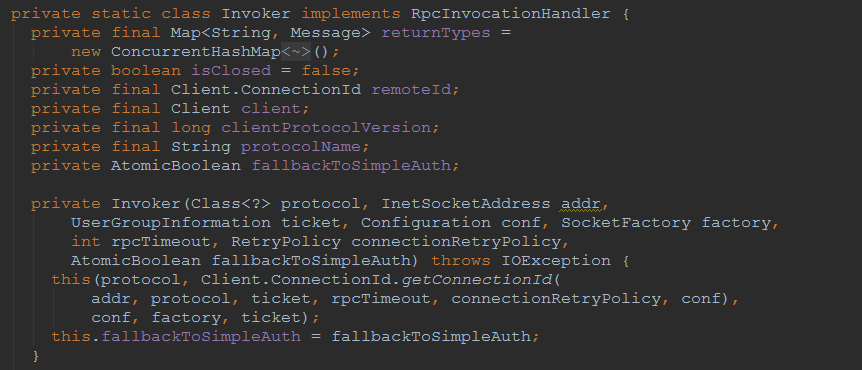
Invoker是InvocationHandler接口的实现类，参数都会传入接口的类的实例，以执行接口方法，参数当中没有传入protocol实例，这是因为真正的实例在Server端，Invoker的作用仅是将方法调用数据发送给远程RPC服务器。

### 2.3.3 客户端创建

根据Java动态代理，Proxy调用接口中的方法时，会被代理到Invoker的invoke方法上，Invoker类实现InvocationHandler接口，是RPC客户端向服务端发出调用的中转站，主要核心在Invoker.invoke方法中，流程如下所示：



其中Invoker的类结构如下所示：



成员变量：client、protocolName等，初始化时创建client对象:

*private Invoker(Class<?> protocol, Client.ConnectionId connId,*

*Configuration conf, SocketFactory factory, UserGroupInformation ticket) {*

*this.remoteId = connId;*

*this.client = CLIENTS.getClient(conf, factory, ticket, RpcWritable.Buffer.class);*

*this.protocolName = RPC.getProtocolName(protocol);*

*this.clientProtocolVersion = RPC*

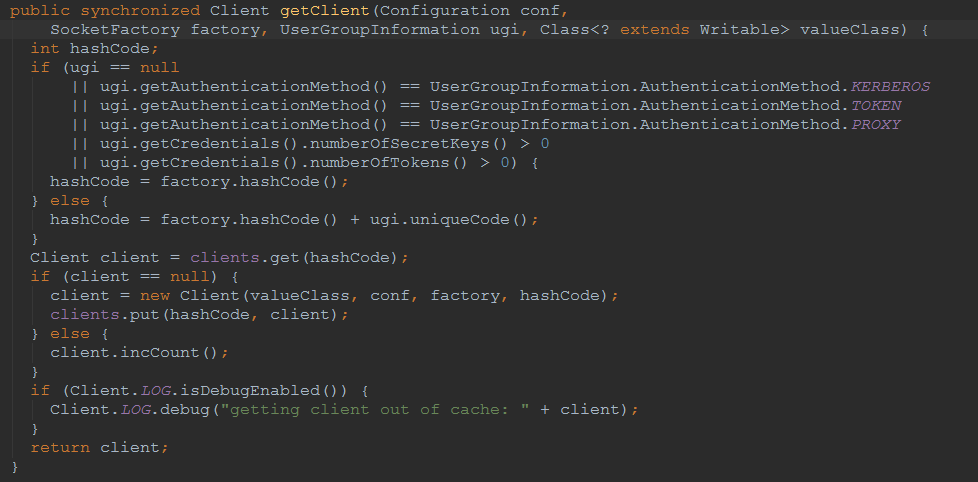
*.getProtocolVersion(protocol);*

*}*

构造方法中用conf和ticket初始化自己的对应成员变量，然后调用CLIENTS(ClientCache)的getClient方法生成客户端对象，在ClientCache中维护了

*private Map<Integer, Client> clients = new HashMap<Integer, Client>();*

Client表，其中保存了UGI/SocketFactory的hashCode与Client的映射表。getClient代码如下所示：



实际上Client的构造没有进行任何操作，也没有与服务器端建立Socket连接，也没有向服务器端以任何形式发送数据，仅初始化Client的主要成员valueClass,SocketFactory及conf等。

### 2.3.4 客户端调用

Proxy第一次调用getProtocolSignature为例（在调用方法前先验证方法在Server端是否支持），参数类型为：protocol,clientVersion及clientMethodsHash，这个方法先被Java动态代理模块利用反射转发到Invoker.invoke方法上，执行代码如下：

*@Override*

*public Message invoke(Object proxy, final Method method, Object[] args)*

*throws ServiceException {*

*long startTime = 0;*

*......*

*RequestHeaderProto rpcRequestHeader = constructRpcRequestHeader(method);*

*final Message theRequest = (Message) args[1];*

*final RpcWritable.Buffer val;*

*try {*

*val = (RpcWritable.Buffer) client.call(RPC.RpcKind.RPC\_PROTOCOL\_BUFFER,*

*new RpcProtobufRequest(rpcRequestHeader, theRequest), remoteId,*

*fallbackToSimpleAuth);*

*} .....*

*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));*

*}….*

*} else {*

*return getReturnMessage(method, val);*

*}*

*}*

该方法调用client.call执行调用，获取返回值Message，当invoke方法返回是proxy getProtocolSignature方法也就返回了。Client.Call方法分析如下：

*Writable call(RPC.RpcKind rpcKind, Writable rpcRequest,*

*ConnectionId remoteId, int serviceClass,*

*AtomicBoolean fallbackToSimpleAuth) throws IOException {*

*final Call call = createCall(rpcKind, rpcRequest); //创建Call对象*

*final Connection connection = getConnection(remoteId, call, serviceClass,*

*fallbackToSimpleAuth); //获取连接*

*connection.sendRpcRequest(call); //发送rpc请求*

*if (isAsynchronousMode()) {*

*final AsyncGet<Writable, IOException> asyncGet*

*= new AsyncGet<Writable, IOException>() {*

*....*

*final Writable w = getRpcResponse(call, connection, timeout, unit);*

*..... //异步获取结果*

*} else {*

*return getRpcResponse(call, connection, -1, null); //同步获取结果*

*}*

*}*

至此客户端的任务已经完成。

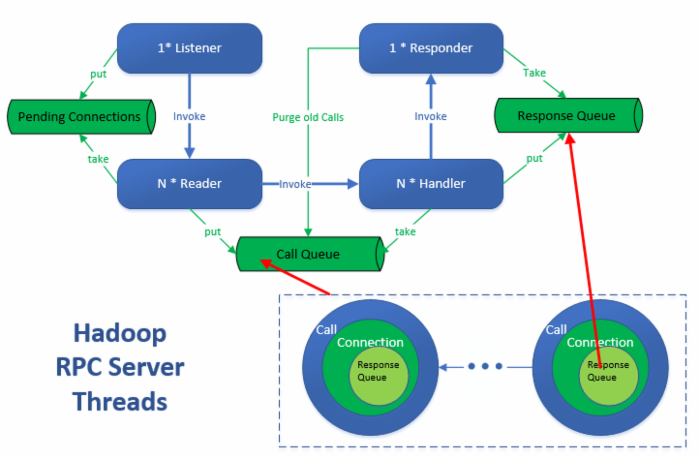
客户端分析：http://blog.csdn.net/xhh198781/article/details/7268298

https://www.csdn.net/article/2014-11-20/2822739?locationNum=8&fps=1

客户端启动流程：http://jingpin.jikexueyuan.com/article/20168.html

## 2.4 RPC Server

RPC Server的执行模型如下所示：



Server的处理划分为三个阶段：接受请求、处理请求和返回结果

1. **接受请求**，主要任务是接收来自客户端的RPC请求，将其封装成固定格式Call类后放入callQueue中，以便进行后续处理。该阶段分为建立连接和接收请求两个子阶段：

* 建立连接由线程Listener完成，整个Server中只有一个Listener线程，统一负责监听来自客户端的连接请求，收到请求后采用轮询的方式从线程池中选择一个Reader线程进行处理
* Reader线程可同时存在多个，接收Listener分配的RPC请求，将请求封装成Call对象，放到共享队列callQueue中

Listener和Reader线程内部各自包含一个Selector对象，分别监听SelectionKey.OP\_ACCEPT和SelectionKey.OP\_READ事件。

1. **处理请求，**该阶段主要任务是从共享队列callQueue中获取Call对象，执行对应的函数调用，这个阶段由Handler线程进行处理，Handler尝试着将直接将结果返回给客户端，如果函数调用的返回结果很大或网速慢，则将结果先放入ResponseQueue中，后续由Response线程处理
2. **返回结果**，该阶段有Response线程进行处理，当遇到Handler不能直接返回结果时（结果太大或者网络异常）将处理结果保存到Response Queue中。Response线程，监听SelectKey.OP\_WRITE事件，采用异步的方式发送结果给客户端。在Server中，仅包含一个Response线程。

RPC.Server端的可配置参数：

|  |  |  |
| --- | --- | --- |
| 参数(ipc.server.xx) | 描述 | default |
| read.threadpool.size | Reader线程数目 | 1 |
| read.connection-queue.size | 每个Reader线程中pending connections数目 | 100 |
| handler.queue.size | 每个handler对相应的最大Call数目 | 100 |
| dfs.datanode.handler.count  dfs.namenode.handler.count  yarn.resourcemanager.resource-  traker.client.thread-count | Handler线程数目， 在hadoop中RM和NN的RPC Server，handler配置参数均不同。默认情况下配置比较小。 | 10  10  50 |

RPC.Server端的类分析如下：



下面进行详细介绍，在执行流程过程中进行类分析。

### 2.4.1 Server初始化

在Hadoop各个组件中的initialize方法中创建和启动RPC服务器代码，例如NameNode RPC Server如下所示：

*this.serviceRpcServer = new RPC.Builder(conf)*

*.setProtocol(*

*org.apache.hadoop.hdfs.protocolPB.ClientNamenodeProtocolPB.class)*

*.setInstance(clientNNPbService)*

*.setBindAddress(bindHost)*

*.setPort(serviceRpcAddr.getPort()).setNumHandlers(serviceHandlerCount)*

*.setVerbose(false)*

*.setSecretManager(namesystem.getDelegationTokenSecretManager())*

*.build();*

其中build方法获取Server，如下所示：

*public Server build() throws IOException, HadoopIllegalArgumentException {*

*return getProtocolEngine(this.protocol, this.conf).getServer(*

*this.protocol, this.instance, this.bindAddress, this.port,*

*this.numHandlers, this.numReaders, this.queueSizePerHandler,*

*this.verbose, this.conf, this.secretManager, this.portRangeConfig);*

*}*

最终调用ProtocolEngine获取Server，如下所示：

*public RPC.Server getServer(Class<?> protocol, Object protocolImpl,*

*String bindAddress, int port, int numHandlers, int numReaders,*

*int queueSizePerHandler, boolean verbose, Configuration conf,*

*SecretManager<? extends TokenIdentifier> secretManager,*

*String portRangeConfig)*

*throws IOException {*

*return new Server(protocol, protocolImpl, conf, bindAddress, port,*

*numHandlers, numReaders, queueSizePerHandler, verbose, secretManager,*

*portRangeConfig);*

*}*

对于RPC.Server类型的对象，参数含义如下：

|  |  |
| --- | --- |
| 参数 | 说明 |
| bindAddress:String | NameNode RPC服务器监听IP地址 |
| port:int | NameNode RPC服务器监听port |
| numberHandlers | RPC服务器中Handler线程数 |
| numReaders | RPC服务器中的Reader线程数 |
| conf | 全局配置 |
| *protocolImpl：Object* | NameNode提供服务的实例 |
| *protocol* | NameNode提供服务的接口 |

### 2.4.2 Listener线程

Listener线程的初始化如下：

*public Listener() throws IOException {*

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

*acceptChannel = ServerSocketChannel.open();*

*acceptChannel.configureBlocking(false);*

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

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

*selector= Selector.open();*

*readers = new Reader[readThreads];*

*for (int i = 0; i < readThreads; i++) {*

*Reader reader = new Reader(*

*"Socket Reader #" + (i + 1) + " for port " + port);*

*readers[i] = reader;*

*reader.start();*

*}*

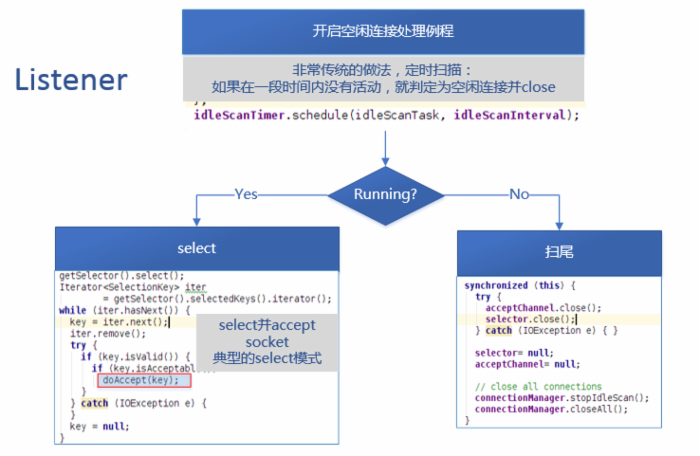
*acceptChannel.register(selector, SelectionKey.OP\_ACCEPT);*

*this.setName("IPC Server listener on " + port);*

*this.setDaemon(true);*

*}*

Listener是一个线程，用于监听客户端的链接和读取客户端发送过来的ConnectionHeader和远程过程调用数据。Listener使用NIO管理Socker连接和数据传输，因此在整个RPC服务器只需一个线程即可处理来自客户端的所有连接和读数据请求。在Listener中开启了一个空闲连接Idle Connection处理例程，如果有过期的空闲连接，就关闭，如下图所示：



当select操作调用时，可能会阻塞，其他线程可执行。当accept事件发生，就会被唤醒以处理全部的事件，处理事件是进行一个doAccept的调用。

doAccept的调用过程如下：

*void doAccept(SelectionKey key) throws InterruptedException, IOException, OutOfMemoryError {*

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

*SocketChannel channel;*

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

*channel.configureBlocking(false);*

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

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

*Reader reader = getReader();*

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

*if (c == null) {*

*if (channel.isOpen()) {*

*IOUtils.cleanup(null, channel);*

*}*

*connectionManager.droppedConnections.getAndIncrement();*

*continue;*

*}*

*key.attach(c); // so closeCurrentConnection can get the object*

*reader.addConnection(c);*

*}*

*}*

在getReader中，采用轮询的方式：

*Reader getReader() { //使用currentReader记录前一次使用的Reader*

*currentReader = (currentReader + 1) % readers.length;*

*return readers[currentReader];*

*}*

### 2.4.3 Reader

当一个新建立的连接被加入Reader缓冲队列pendingConnections之后，Reader也被唤醒，以处理此连接上的数据接收：

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

*pendingConnections.put(conn);*

*readSelector.wakeup();*

*}*

Server中配置了多个Reader线程，为了提高并发服务连接的能力。下面是Reader的主要逻辑：

*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(); //取出一个连接，可能阻塞*

*//向select注册一个读事件*

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

*}*

*readSelector.select(); //进行select，可能阻塞*

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

*while (iter.hasNext()) {*

*key = iter.next(); //依次读取数据*

*iter.remove();*

*try {*

*if (key.isReadable()) {*

*doRead(key);*

*}*

*} .....*

*}*

*}*

当Server还在运行时，Reader线程尽可能多处理缓冲队列中的连接，注册每个连接READ事件，采用select模式来获取连接上有数据接收的通知。当有数据需要接收时，尽最大可能读取select返回的连接上的数据，以防止Listener线程因为没有运行时间而发生饥饿。

在Reader的doRead调用中，其主要调用了readAndProcess方法，此方法循环处理数据，接收数据包的头部、上下文头部和真正的数据。

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

*while (!shouldClose()) { // stop if a fatal response has been sent.*

*if (!connectionHeaderRead) { //读取数据头*

*//Every connection is expected to send the header.*

*if (connectionHeaderBuf == null) {*

*connectionHeaderBuf = ByteBuffer.allocate(3);*

*}*

*count = channelRead(channel, connectionHeaderBuf);*

*......}}*

*if (data == null) { //读取请求数据*

*dataLengthBuffer.flip();*

*dataLength = dataLengthBuffer.getInt();*

*checkDataLength(dataLength);*

*data = ByteBuffer.allocate(dataLength);*

*}*

*count = channelRead(channel, data);*

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

*dataLengthBuffer.clear();*

*data.flip();*

*ByteBuffer requestData = data;*

*data = null;*

*boolean isHeaderRead = connectionContextRead;*

*processOneRpc(requestData); //处理接收到的数据*

*if (!isHeaderRead) {*

*continue;*

*}*

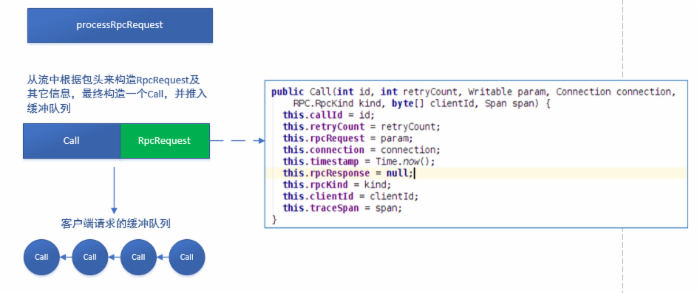
*}*

*return count;*

*}*

*return -1;}*

其中channelRead会判断数据接收数组buffer中剩余未读数据，如果大于一个临界值NIO\_BUFFER\_LIMIT，就采取分片的技巧来多次地读。当Reader读取一个完整的RpcRequest包之后，会调用processOneRpc方法，此调用将进入业务逻辑环节，这个方法中，会从接受到数据包中，反序列化出RpcRequest的头部和数据，依次构造一个RpcRequest对象，设置客户端需要的跟踪信息，然后构造一个Call对象，如下图所示：



序列化成一个Call后，放入CallQueue中，以供Handler处理，当callQueue满时，Reader会阻塞。

### 2.4.4 Handler

Handler是根据rpc请求中的方法call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的逻辑基本如下：

*public void run() {*

*while (running) {*

*TraceScope traceScope = null;*

*try {*

*final Call call = callQueue.take(); //从队列中获取Call对象，对该call进行处理*

*CurCall.set(call);*

*if (call.traceScope != null) {*

*call.traceScope.reattach();*

*traceScope = call.traceScope;*

*traceScope.getSpan().addTimelineAnnotation("called");*

*}*

*CallerContext.setCurrent(call.callerContext);*

*UserGroupInformation remoteUser = call.getRemoteUser();*

*if (remoteUser != null) {*

*remoteUser.doAs(call); //执行call方法*

*} else {*

*call.run();*

*}*

*} ......*

*}}*

可见，Handler从callQueue中取出一个Call，然后调用call.run方法，其中RpcCall.run方法如下图所示：

*public Void run() throws Exception {*

*....*

*value = call(*

*rpcKind, connection.protocolName, rpcRequest, timestamp);*

*setupResponse(this, returnStatus, detailedErr,*

*value, errorClass, error);*

*sendResponse();*

*return null;*

*}*

其中sendResponse中方法中将数据结果发送给connection，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);*

*}*

Connection#sendResponse，将call传递给Responder中，如下所示：

*private void sendResponse(RpcCall call) throws IOException {*

*responder.doRespond(call);*

*}*

### 2.4.5 Responder

一个Server中只有1个Responder线程，此线程不断进行如下几个重要调用以和Handler协调并发送数据：

*private void doRunLoop() {*

*while (running) {*

*try {*

*waitPending(); //这个wait是同步作用*

*writeSelector.select(PURGE\_INTERVAL); //开始select，或许会阻塞*

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

*while (iter.hasNext()) {*

*SelectionKey key = iter.next();*

*iter.remove();*

*try {*

*if (key.isWritable()) { //如果selectKeys有数据，就依次异步发送数据*

*doAsyncWrite(key); //处理Response 的RpcCall*

*} } .....}*

*//当到到丢失时间，会从selectedKeys构造calls，并依次丢弃*

*for (RpcCall call : calls) {*

*doPurge(call, now);*

*}} ......}*

当Handler调用doResponse方法后，handler处理的结果被加入到responseQueue的对尾，而不是立即发送回客户端：

*void doRespond(RpcCall call) throws IOException {*

*synchronized (call.connection.responseQueue) {*

*if (call.connection.useWrap) {*

*wrapWithSasl(call);*

*}*

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

*if (call.connection.responseQueue.size() == 1) {*

*processResponse(call.connection.responseQueue, true);*

*}*

*}*

Handler是生产者，将结果加入队列，Responder是消费者从队列种取出结果并发送。在这个方法中processResponse方法将启动Responder进行发送，首先从responseQueue中以非阻塞方法取出一个call，然后以非阻塞方式尽力发送call.rpcResponse，如果发送完毕，则返回。

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

*boolean inHandler) throws IOException {*

*.....*

*try {*

*synchronized (responseQueue) {*

*numElements = responseQueue.size();*

*call = responseQueue.removeFirst();*

*SocketChannel channel = call.connection.channel;*

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

*if (inHandler) {*

*incPending();*

*try {*

*writeSelector.wakeup();*

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

*} …..*

*}*

通过channelWrite，将处理结果发送给客户端。

# 参考文献：

1）线程模型分析（代码描述结构比较赞）：

https://wenku.baidu.com/view/b0a8acc95ebfc77da26925c52cc58bd631869304.html

2）Server端分析：http://blog.csdn.net/xhh198781/article/details/7280084

3）Server端类分析：http://blog.csdn.net/Androidlushangderen/article/details/41785711

http://blog.csdn.net/u010039929/article/details/76436664

http://www.cnblogs.com/ahhuiyang/p/3854366.html

4）RPC框架：http://blog.csdn.net/thomas0yang/article/details/41211259

5）RPC example: https://github.com/sz-zzm/rpc/blob/master/pom.xml

6）github：

https://github.com/loull521/hadoop-yarn-src-read/blob/master/docs/hadoop\_rpc\_1.md

http://www.lxway.com/995500624.htm

http://blog.csdn.net/Androidlushangderen/article/details/41785711

http://www.linkedkeeper.com/detail/blog.action?bid=156

https://www.cnblogs.com/edisonchou/p/4285817.html