阿里巴巴 Dubbo 实现的源码分析

1. Dubbo 概述

Dubbo 是阿里巴巴开源出来的一个分布式服务框架，致力于提供高性能和透明化的 RPC 远程服务调用方案，以及作为 SOA 服务治理的方案。它的核心功能包括：

#remoting:远程通讯基础，提供对多种 NIO 框架抽象封装，包括“同步转异步”和“请求-响应”模式的信息交换方式。

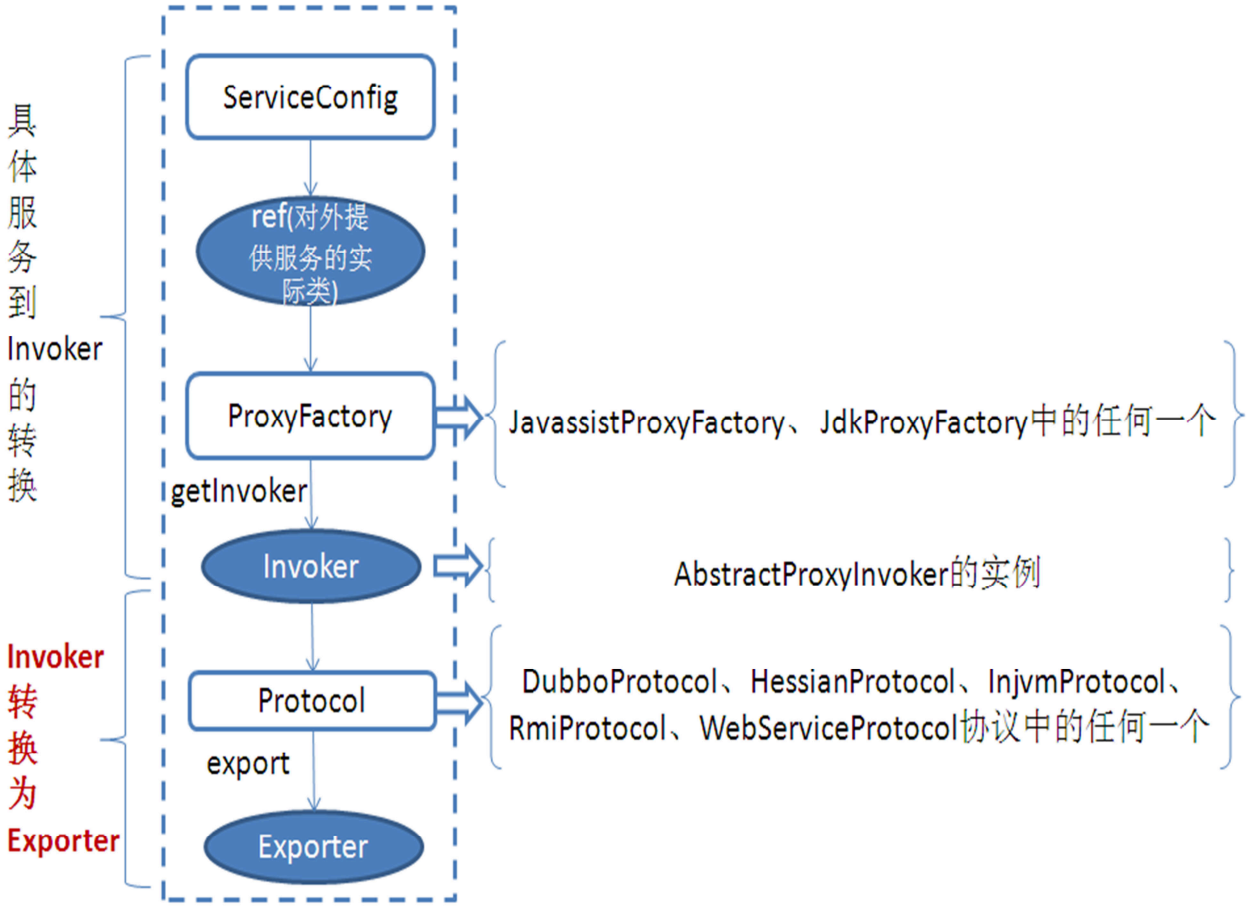
#Cluster: 服务框架核心，提供基于接口方法的远程过程调用，包括多协议支持，并提供软负载均衡和容错机制的集群支持。

#registry: 服务注册中心，使服务消费方能动态的查找服务提供方，使地址透明，使服务提供方可以平滑增加或减少机器。

由于 Dubbo 团队的文档和代码都非常优秀，所以更多关于 dubbo 的方方面面请参考网站 http://code.alibabatech.com/wiki/display/dubbo/Home-zh。

这里我们只是补充一下从源码具体实现角度来看的某些细节方面，包括 Invoker、ExtensionLoader 等方面。任何官方已经介绍过的细节，我们不做画蛇添足，官方文档已经足够详实了，这篇文档的定位是补充实现的相关细节，是基于我在往 Dubbo 添加 web service 协议过程中，所碰到过的一些困难。

2. 服务提供者暴露一个服务的详细过程



上图是服务提供者暴露服务的主过程：

首先 ServiceConfig 类拿到对外提供服务的实际类 ref(如：HelloWorldImpl),然后通过 ProxyFactory 类的 getInvoker 方法使用 ref 生成一个 AbstractProxyInvoker 实例，到这一步就完成具体服务到 Invoker 的转化。接下来就是 Invoker 转换到 Exporter 的过程。

Dubbo 处理服务暴露的关键就在 Invoker 转换到 Exporter 的过程(如上图中的红色部分)，下面我们以 Dubbo 和 RMI 这两种典型协议的实现来进行说明：

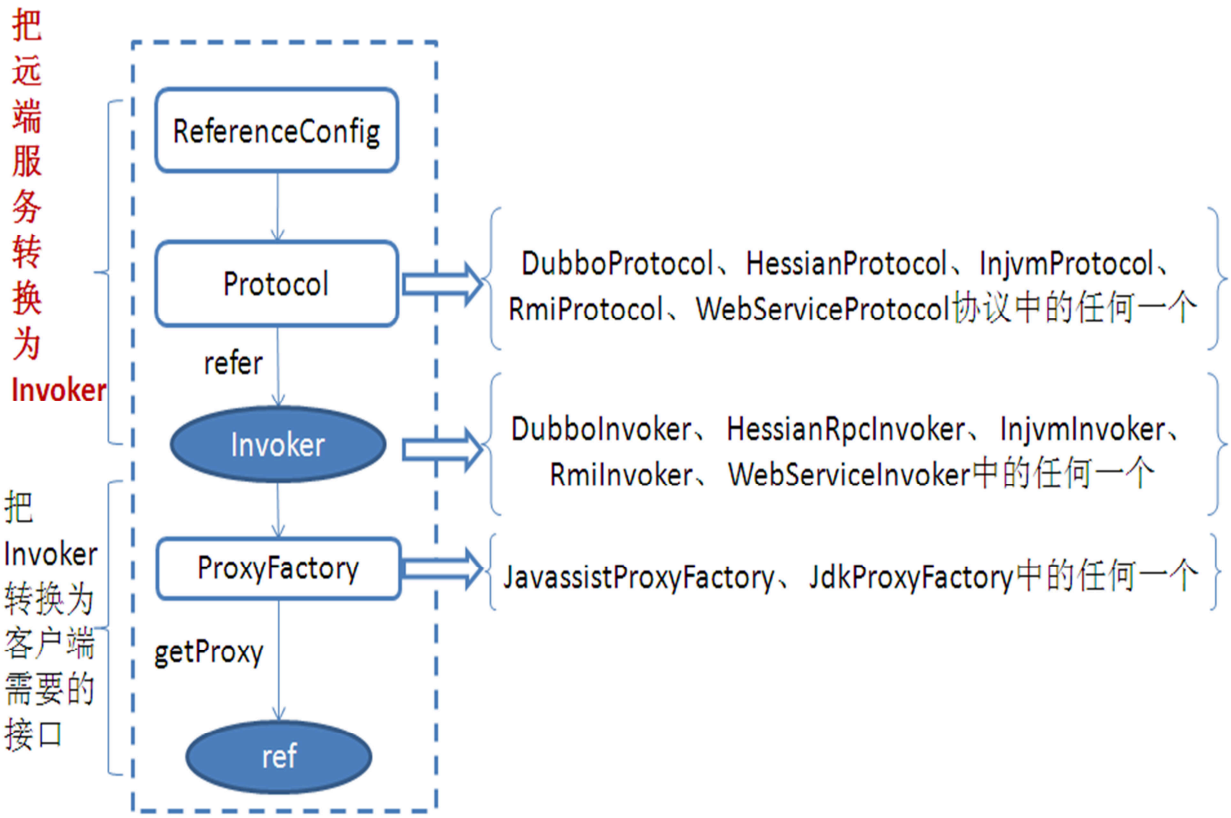
#Dubbo 的实现

Dubbo 协议的 Invoker 转为 Exporter 发生在 DubboProtocol 类的 export 方法，它主要是打开 socket 侦听服务，并接收客户端发来的各种请求，通讯细节由 Dubbo 自己实现。

#RMI 的实现

RMI 协议的 Invoker 转为 Exporter 发生在 RmiProtocol 类的 export 方法，它通过 Spring 或 Dubbo 或 JDK 来实现 RMI 服务，通讯细节这一块由 JDK 底层来实现，这就省了不少工作量。

3. 服务消费者消费一个服务的详细过程



上图是服务消费的主过程：

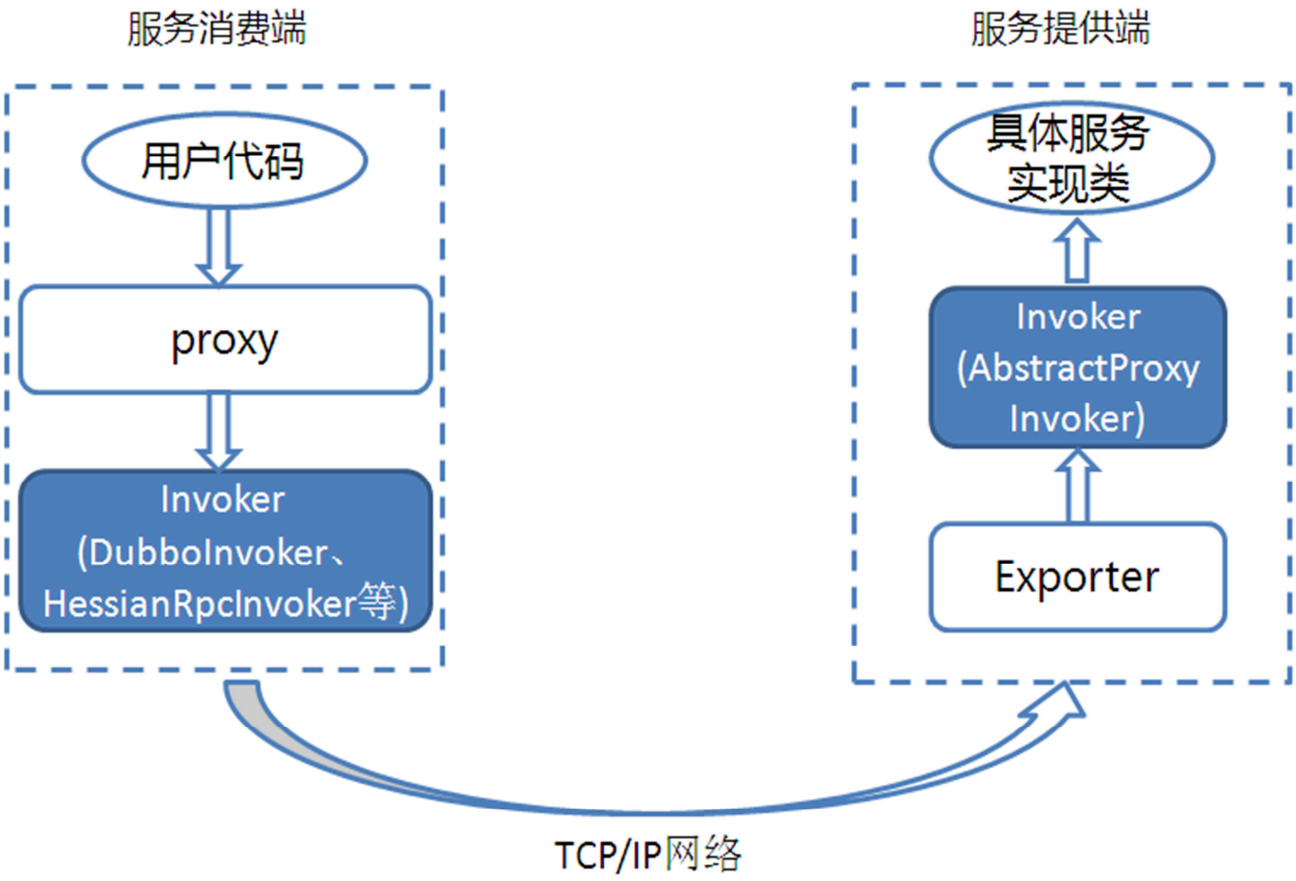
首先 ReferenceConfig 类的 init 方法调用 Protocol 的 refer 方法生成 Invoker 实例(如上图中的红色部分)，这是服务消费的关键。接下来把 Invoker 转换为客户端需要的接口(如：HelloWorld)。

关于每种协议如 RMI/Dubbo/Web service 等它们在调用 refer 方法生成 Invoker 实例的细节和上一章节所描述的类似。

4. 满眼都是 Invoker

由于 Invoker 是 Dubbo 领域模型中非常重要的一个概念，很多设计思路都是向它靠拢。这就使得 Invoker 渗透在整个实现代码里，对于刚开始接触 Dubbo 的人，确实容易给搞混了。

下面我们用一个精简的图来说明最重要的两种 Invoker：服务提供 Invoker 和服务消费 Invoker：



为了更好的解释上面这张图，我们结合服务消费和提供者的代码示例来进行说明：

```
#服务消费者代码
public class DemoClientAction {
    private DemoService demoService;
    public void setDemoService(DemoService demoService) {
        this.demoService = demoService;
}
    public void start() {
            String hello = demoService.sayHello("world" + i);
     }
}
```

上面代码中的'DemoService'就是上图中服务消费端的 proxy，用户代码通过这个 proxy 调用其对应的 Invoker(DubboInvoker、 HessianRpcInvoker、 InjvmInvoker、 RmiInvoker、 WebServiceInvoker 中的任何一个)，而该 Invoker 实现了真正的远程服务调用。

```
#服务提供者代码
public class DemoServiceImpl
  implements DemoService
{
  public String sayHello(String name) throws RemoteException
  {
    return "Hello " + name;
  }
}
```

上面这个类会被封装成为一个 AbstractProxyInvoker 实例，并新生成一个 Exporter 实例。这样当网络通讯层收到一个请求后，会找到对应的 Exporter 实例，并调用它所对应的 AbstractProxyInvoker 实例，从而真正调用了服务提供者的代码。

Dubbo 里还有一些其他的 Invoker 类，但上面两种是最重要的。

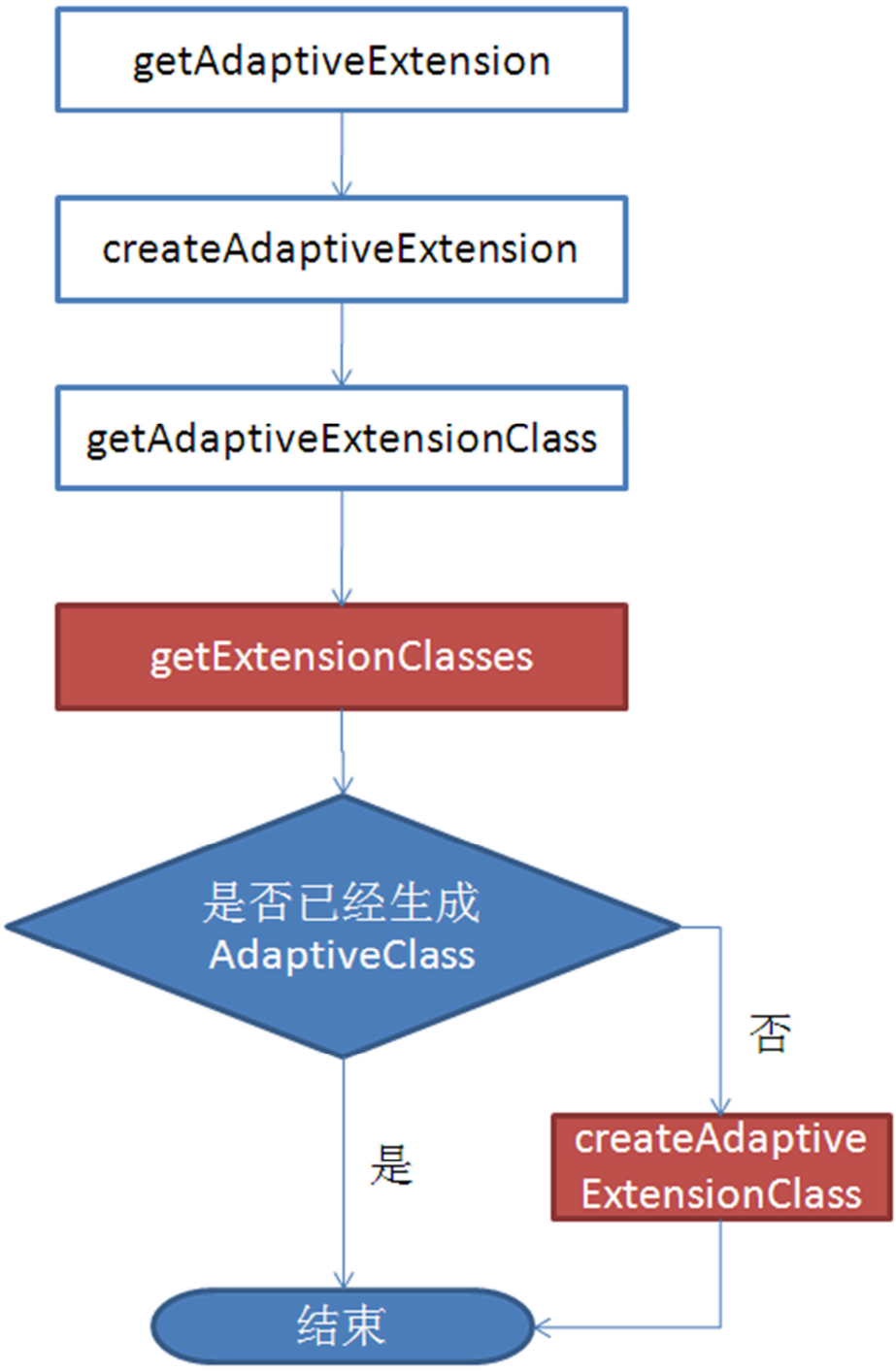
5. ExtensionLoader 的完整分析

ExtensionLoader 是 Dubbo 中一个非常重要的类，刚接触 Dubbo 源码的人看这个类的时候也多少会有点困惑，这个类非常重要，它就像是厨房里的“大厨”，按照用户的随时需要把各种“食材”烹调出来。

我们结合具体代码详细说一下 ExtensionLoader 的实现，下面是 ServiceConfig 类里的一行代码：

```
private static final Protocol protocol =
ExtensionLoader.getExtensionLoader(Protocol.class).getAdaptiveExtension();
```

上面代码的程序流程图如下所示(假定是第一次执行这行代码)：



在这个过程中最重要的两个方法是 getExtensionClasses 和 createAdaptiveExtensionClass(图中红色部分)，下面详细对这两个方法进行分析：

＃**getExtensionClasses**

这个方法主要读取 META-INF/services/目录下对应文件内容，在本示例代码中，是读取 META-INF/services/com.alibaba.dubbo.rpc.Protocol 文件中的内容，具体内容如下：

com.alibaba.dubbo.registry.support.RegistryProtocol

com.alibaba.dubbo.rpc.protocol.ProtocolFilterWrapper

com.alibaba.dubbo.rpc.protocol.ProtocolListenerWrapper

com.alibaba.dubbo.rpc.protocol.dubbo.DubboProtocol

com.alibaba.dubbo.rpc.protocol.injvm.InjvmProtocol

com.alibaba.dubbo.rpc.protocol.rmi.RmiProtocol

com.alibaba.dubbo.rpc.protocol.hessian.HessianProtocol

com.alibaba.dubbo.rpc.protocol.webservice.WebServiceProtocol

它分析该文件中的每一行(每一行对应一个类)，分析这些类，如果发现有哪个类的 Annotation 是@Adaptive，则找到对应的 AdaptiveClass 了，但由于 Protocol 文件里没有哪个类的 Annotation 是@Adaptive，所以在这个例子中该方法没找到对应的 AdaptiveClass。

＃**createAdaptiveExtensionClass**

该方法是在 getExtensionClasses 方法找不到 AdaptiveClass 的情况下被调用，该方法主要是通过字节码的方式在内存中新生成一个类，它具有 AdaptiveClass 的功能，Protocol 就是通过这种方式获得 AdaptiveClass 类的。

AdaptiveClass 类的作用是能在运行时动态判断具体是要调用哪个类的方法，更多关于 AdaptiveClass 的内容请参考 Dubbo 官方文档。

关键代码：

```
com.taobao.remoting.impl.DefaultClient.java
//同步调用远程接口
public Object invokeWithSync(Object appRequest, RequestControl control) throws
RemotingException, InterruptedException {
        byte protocol = getProtocol(control);
        if (!TRConstants.isValidProtocol(protocol)) {
            throw new RemotingException("Invalid serialization protocol [" + protocol + "]
on invokeWithSync.");
        }
        ResponseFuture future = invokeWithFuture(appRequest, control);
        return future.get();  //获取结果时让当前线程等待，ResponseFuture 其实就是前
面说的 callback
}
public ResponseFuture invokeWithFuture(Object appRequest, RequestControl
control) {
         byte protocol = getProtocol(control);
         long timeout = getTimeout(control);
         ConnectionRequest request = new ConnectionRequest(appRequest);
```

```
        request.setSerializeProtocol(protocol);
        Callback2FutureAdapter adapter = new Callback2FutureAdapter(request);
        connection.sendRequestWithCallback(request, adapter, timeout);
        return adapter;
}
```

```
Callback2FutureAdapter implements ResponseFuture
public Object get() throws RemotingException, InterruptedException {
synchronized (this) {   // 旋锁
    while (!isDone) {   // 是否有结果了
wait(); //没结果是释放锁，让当前线程处于等待状态
    }
}
if (errorCode == TRConstants.RESULT_TIMEOUT) {
    throw new TimeoutException("Wait response timeout, request["
    + connectionRequest.getAppRequest() + "].");
}
else if (errorCode > 0) {
    throw new RemotingException(errorMsg);
}
else {
    return appResp;
}
}
客户端收到服务端结果后，回调时相关方法，即设置 isDone = true 并 notifyAll()
public void handleResponse(Object _appResponse) {
        appResp = _appResponse; //将远程调用结果设置到 callback 中来
        setDone();
}
public void onRemotingException(int _errorType, String _errorMsg) {
        errorCode = _errorType;
        errorMsg = _errorMsg;
        setDone();
}
private void setDone() {
```

```
        isDone = true;
        synchronized (this) { //获取锁，因为前面 wait()已经释放了 callback 的锁了
            notifyAll(); // 唤醒处于等待的线程
        }
}
```

```
com.taobao.remoting.impl.DefaultConnection.java

// 用来存放请求和回调的 MAP
private final ConcurrentHashMap<Long, Object[]> requestResidents;

//发送消息出去
void sendRequestWithCallback(ConnectionRequest connRequest,
ResponseCallback callback, long timeoutMs) {
        long requestId = connRequest.getId();
        long waitBegin = System.currentTimeMillis();
        long waitEnd = waitBegin + timeoutMs;
        Object[] queue = new Object[4];
        int idx = 0;
        queue[idx++] = waitEnd;
        queue[idx++] = waitBegin;   //用于记录日志
        queue[idx++] = connRequest; //用于记录日志
        queue[idx++] = callback;
       requestResidents.put(requestId, queue); // 记录响应队列
       write(connRequest);

        // 埋点记录等待响应的 Map 的大小
        StatLog.addStat("TBRemoting-ResponseQueues", "size",
requestResidents.size(),
                1L);
}
public void write(final Object connectionMsg) {
//mina 里的 IoSession.write()发送消息
        WriteFuture writeFuture = ioSession.write(connectionMsg);
        // 注册 FutureListener，当请求发送失败后，能够立即做出响应
```

```
        writeFuture.addListener(new MsgWrittenListener(this, connectionMsg));
}

/**
* 在得到响应后，删除对应的请求队列，并执行回调
* 调用者：MINA 线程
*/
public void putResponse(final ConnectionResponse connResp) {
        final long requestId = connResp.getRequestId();
        Object[] queue = requestResidents.remove(requestId);
        if (null == queue) {
            Object appResp = connResp.getAppResponse();
            String appRespClazz = (null == appResp) ? "null" :
appResp.getClass().getName();
            StringBuilder sb = new StringBuilder();
            sb.append("Not found response receiver for
requestId=[").append(requestId).append("],");
            sb.append("from [").append(connResp.getHost()).append("],");
            sb.append("response type [").append(appRespClazz).append("].");
            LOGGER.warn(sb.toString());
            return;
        }
        int idx = 0;
        idx++;
        long waitBegin = (Long) queue[idx++];
        ConnectionRequest connRequest = (ConnectionRequest) queue[idx++];
        ResponseCallback callback = (ResponseCallback) queue[idx++];
        // ** 把回调任务交给业务提供的线程池执行 **
        Executor callbackExecutor = callback.getExecutor();
        callbackExecutor.execute(new CallbackExecutorTask(connResp, callback));

        long duration = System.currentTimeMillis() - waitBegin; // 实际读响应时间
        logIfResponseError(connResp, duration, connRequest.getAppRequest());
}
```

```
CallbackExecutorTask
static private class CallbackExecutorTask implements Runnable {
        final ConnectionResponse resp;
        final ResponseCallback callback;
        final Thread createThread;

        CallbackExecutorTask(ConnectionResponse _resp, ResponseCallback _cb) {
            resp = _resp;
            callback = _cb;
            createThread = Thread.currentThread();
        }

        public void run() {
            // 预防这种情况：业务提供的 Executor，让调用者线程来执行任务
            if (createThread == Thread.currentThread()
                    && callback.getExecutor() != DIYExecutor.getInstance()) {
                StringBuilder sb = new StringBuilder();
                sb.append("The network callback task [" + resp.getRequestId() + "]
cancelled, cause:");
                sb.append("Can not callback task on the network io thhread.");
                LOGGER.warn(sb.toString());
                return;
            }

            if (TRConstants.RESULT_SUCCESS == resp.getResult()) {
                callback.handleResponse(resp.getAppResponse()); //设置调用结果
            }
            else {
                callback.onRemotingException(resp.getResult(), resp
                        .getErrorMsg());  //处理调用异常
            }
        }
}
```

另外：

1， 服务端在处理客户端的消息，然后再处理时，使用了线程池来并行处理，不用一个一个消息的处理

同样，客户端接收到服务端的消息，也是使用线程池来处理消息，再回调