**简洁高效RPC框架**

——刘向前 20170303

**一、分布式**

多线程是一台机器同时让多个cpu核心一起做一件事情，分布式是让多台机器共同做一件事。多线程能够共同做事是因为可以通信，多台机器上的进程可以通信自然也可以一起做事。所以，分布式与多线程只是通信方式不同罢了。

分布式通信就是网络通信，网络通信说到底就是基于socket的通信。所以，如何网络通信才是分布式的关键。

网络通信可以基于应用层进行，也可以基于网络层进行。

显然，基于应用层通信要慢于网络层通信。

所以，我们要做基于网络层的分布式框架，这篇文章分享的是最简化的RPC框架。

说明，本文的RPC框架通过该着hdfs的分布式框架而来，hdfs的RPC是完全融于hdfs的业务需求，而本文侧重于提取通用的东西，将精髓做成RPC供方便调用。

如果想多hdfs有更多了解，请参考徐鹏写的《Hadoop 2.x HDFS源码分析》。

**二、为什么要自己写**

这一段没有兴趣的请略过，只是给朋友分享为什么我没有用开源框架，而是选择自己写RPC；其实也是传达一下我对技术的认识，我认为技术人员要尽量有所长，所长一定要尽量追本溯源；深度越深，解决问题越简单有效。

当年是一个分布式白痴，老大是一个分布式大牛，当时我只会写一些单机版算法；自然就想着能够将单机版的东西分布式化，利用多台机器更高效工作；特别是想写一套分布式的垂直搜索引擎。于是，开始了分布式的追寻之旅。

首先，经一个苏海波兄弟推荐，读了几遍《大数据日志录》，系统了解了行业的开源软件。可是用一个开源软件可不代表你会分布式吧。

后来，学习java rmi，学习fourinone等等，可是我一直不明白，为什么要有个stub呢？

Fourinone算是一个不错的尝试，他用最少的代码，实现了不少hadoop要做的工作，而且更加高效，至少与我的思路相合，他告诉我们掌握的深，事情更简单，谢谢这位仁兄。

很多时候我们不需要通用的庞大代码融入我们的程序中，采用开源实现我们需要的一小点分布式功能，却要引入几十万行代码的包，可控性很差，而且我们还必须按照人家的规矩做，一旦规矩没学好，用起来问题还挺多，一有问题自己不知道哪里下手，毕竟代码量那么多，你很难完全明白吧。

如果你要深度学习别人的开源代码，学习代码风格，设计理念；这我真的很佩服，希望你能坚持下去，徐鹏对hadoop的理解就非常棒。

后来我开始阅读hadoop源码，后来网上查源码分析的资料，碰巧徐鹏写了《hdfs源码分析》，RPC那部分正好用来学习。Hdfs基于socket写的RPC框架，简单，高效，太合胃口了，因为我能很好的控制每行代码啊。于是，根据垂直搜索需要，写了一版RPC，经过压力测试，觉得还是不错的。

有了自己的钟爱的分布式框架，现在共享给大家，希望对你有点帮助。不过，我觉得理念更重要，追根溯源才是我想表达的，这样我们在一个领域内才能做到极致，才有更多价值。

显然，会有更多板砖拍来，因为中国的牛逼工程师更多善于使用开源，而且混的比我好多了。确实，看看招聘广告，我这些玩意恐怕真不受欢迎。不过，我还是比较喜欢用最简单最根本最可控的方法，写我的小东西，解决我的小问题。

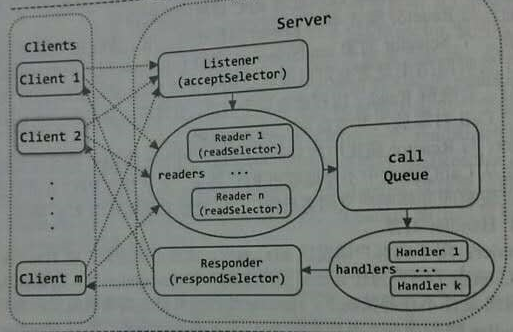
**三、RPC框架**

1、框架图



是不是大家觉得很熟悉，整个过程不过时一个client到server的访问，中间通道是socket;与纯粹的server和client不同的是添加了编解码过程，并根据参数进行函数调用。其实根据具体业务具体开发的话，只需要client和server约定好就可以了。当然，我们的框架并不是一个傻瓜式的框架，而是要client和server端约定好编解码和函数调用，这部分是需要根据业务进行重新开发的。唯一不变的是，server端的处理流程和client端的请求方法。

2、RPC.Server框架



基于selector的网络框架，这边才是RPC的核心，保证了高效处理请求。

步骤一：Listener创建AcceptChannel监听链接

address = new InetSocketAddress(bindAddress, port);  
// Create a new server socket and set to non blocking mode  
acceptChannel = ServerSocketChannel.*open*();  
acceptChannel.configureBlocking(false);

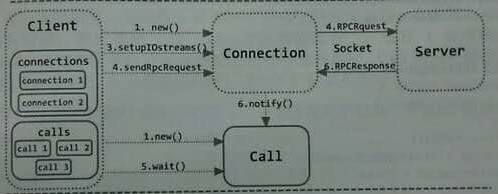
步骤二：Listener创建多个Reader类通过创建的readchannel进行请求获取

步骤三：Reader将获得的链接信息和链接包装成call结构存放到call queue队列中

步骤四：Handler线程对call queue队列进行处理，并将结果通过call中的链接返回给客户端

注意：client和server其实只有一个真实的socket的链接

3、RPC.Client框架



步骤一：client将请求存放到Connection的队列中，并等待结果

步骤二：connection不断将队列中的请求编码并发送到server端

步骤三：connection 获得请求结果，会唤醒相应的等待请求

步骤四：client获得了要的请求

注意：上面是没有问题的情况，因为请求过程会出现超时或server死掉的情况

Server死掉的情况，定期链接并不断扩大不访问时间

**超时情况就不同了，这是一个关键问题，也是这个框架的一个难点，真的不好调试啊！**

因为线程断掉时候，channel也会断掉，获取getConnection部分需要对线程是否在运行进行判断(斜体部分)，不然rpc的client会死掉。

为什么会死掉，且没有清除？而按框架说死掉会自动清掉。

这个问题我真心不是彻底明白，虽然是我写的，多线程访问确实有时候我也有点解释不清，哪位能够帮解释下，不胜感激。

//null: 表示没有连接还没有获取到  
private Connection getConnection(ConnectionId remoteId, Call call) {  
 if (!running.get()) {  
 // the client is stopped  
 return null;  
 }  
 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.  
 \*/  
 do {  
 synchronized (connections) {  
 *connection = connections.get(remoteId);  
 if (connection != null && !connection.isAlive()) {  
 connections.remove(remoteId);  
 connection = null;  
 }*  
  
 if (connection == null) {  
 connection = new Connection(remoteId);  
 connections.put(remoteId, connection);  
 try {  
 if (!connection.setupIOstreams()) {  
 connections.remove(remoteId);  
 return null;  
 }  
 } catch (IOException e) {  
 //说明服务还没有起来  
 connections.remove(remoteId);  
 return null;  
 }  
  
 connection.start();  
 } else if (!connection.isAlive()) {  
 connections.remove(remoteId);  
 }  
 }  
 } while (!connection.addCall(call));  
  
 return connection;  
}

**四、代码网址**