博客来源地址: http://www.importnew.com/24689.html

一个轻量级分布式 RPC 框架 — NettyRpc

2017/05/19 | 分类： [基础技术](http://www.importnew.com/cat/basic) | [2 条评论](http://www.importnew.com/24689.html#comments) | 标签： [NETTYRPC](http://www.importnew.com/tag/nettyrpc), [RPC](http://www.importnew.com/tag/rpc)

分享到：

原文出处： [阿凡卢](http://www.cnblogs.com/luxiaoxun/p/5272384.html)

1、背景

最近在搜索Netty和Zookeeper方面的文章时，看到了这篇文章《[轻量级分布式 RPC 框架](https://my.oschina.net/huangyong/blog/361751" \t "_blank)》，作者用Zookeeper、Netty和Spring写了一个轻量级的分布式RPC框架。花了一些时间看了下他的代码，写的干净简单，写的RPC框架可以算是一个简易版的[dubbo](http://dubbo.io/" \t "_blank)。这个RPC框架虽小，但是麻雀虽小，五脏俱全，有兴趣的可以学习一下。

本人在这个简易版的RPC上添加了如下特性：

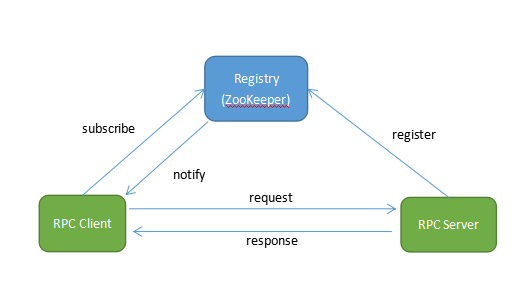
* 服务异步调用的支持，回调函数callback的支持
* 客户端使用长连接（在多次调用共享连接）
* 服务端异步多线程处理RPC请求

项目地址：https://github.com/luxiaoxun/NettyRpc

2、简介

RPC，即 Remote Procedure Call（远程过程调用），调用远程计算机上的服务，就像调用本地服务一样。RPC可以很好的解耦系统，如WebService就是一种基于Http协议的RPC。

这个RPC整体框架如下：

[](http://www.importnew.com/?attachment_id=24690)

这个RPC框架使用的一些技术所解决的问题：

服务发布与订阅：服务端使用Zookeeper注册服务地址，客户端从Zookeeper获取可用的服务地址。

通信：使用Netty作为通信框架。

Spring：使用Spring配置服务，加载Bean，扫描注解。

动态代理：客户端使用代理模式透明化服务调用。

消息编解码：使用Protostuff序列化和反序列化消息。

3、服务端发布服务

使用注解标注要发布的服务

服务注解

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Target({ElementType.TYPE})  @Retention(RetentionPolicy.RUNTIME)  @Component  public @interface RpcService {      Class<?> value();  } |

一个服务接口：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public interface HelloService {        String hello(String name);        String hello(Person person);  } |

一个服务实现：使用注解标注

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | @RpcService(HelloService.class)  public class HelloServiceImpl implements HelloService {        @Override      public String hello(String name) {          return "Hello! " + name;      }        @Override      public String hello(Person person) {          return "Hello! " + person.getFirstName() + " " + person.getLastName();      }  } |

服务在启动的时候扫描得到所有的服务接口及其实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @Override      public void setApplicationContext(ApplicationContext ctx) throws BeansException {          Map<String, Object> serviceBeanMap = ctx.getBeansWithAnnotation(RpcService.class);          if (MapUtils.isNotEmpty(serviceBeanMap)) {              for (Object serviceBean : serviceBeanMap.values()) {                  String interfaceName = serviceBean.getClass().getAnnotation(RpcService.class).value().getName();                  handlerMap.put(interfaceName, serviceBean);              }          }      } |

在Zookeeper集群上注册服务地址：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71 | public class ServiceRegistry {        private static final Logger LOGGER = LoggerFactory.getLogger(ServiceRegistry.class);        private CountDownLatch latch = new CountDownLatch(1);        private String registryAddress;        public ServiceRegistry(String registryAddress) {          this.registryAddress = registryAddress;      }        public void register(String data) {          if (data != null) {              ZooKeeper zk = connectServer();              if (zk != null) {                  AddRootNode(zk); // Add root node if not exist                  createNode(zk, data);              }          }      }        private ZooKeeper connectServer() {          ZooKeeper zk = null;          try {              zk = new ZooKeeper(registryAddress, Constant.ZK\_SESSION\_TIMEOUT, new Watcher() {                  @Override                  public void process(WatchedEvent event) {                      if (event.getState() == Event.KeeperState.SyncConnected) {                          latch.countDown();                      }                  }              });              latch.await();          } catch (IOException e) {              LOGGER.error("", e);          }          catch (InterruptedException ex){              LOGGER.error("", ex);          }          return zk;      }        private void AddRootNode(ZooKeeper zk){          try {              Stat s = zk.exists(Constant.ZK\_REGISTRY\_PATH, false);              if (s == null) {                  zk.create(Constant.ZK\_REGISTRY\_PATH, new byte[0], ZooDefs.Ids.OPEN\_ACL\_UNSAFE, CreateMode.PERSISTENT);              }          } catch (KeeperException e) {              LOGGER.error(e.toString());          } catch (InterruptedException e) {              LOGGER.error(e.toString());          }      }        private void createNode(ZooKeeper zk, String data) {          try {              byte[] bytes = data.getBytes();              String path = zk.create(Constant.ZK\_DATA\_PATH, bytes, ZooDefs.Ids.OPEN\_ACL\_UNSAFE, CreateMode.EPHEMERAL\_SEQUENTIAL);              LOGGER.debug("create zookeeper node ({} => {})", path, data);          } catch (KeeperException e) {              LOGGER.error("", e);          }          catch (InterruptedException ex){              LOGGER.error("", ex);          }      }  }    ServiceRegistry |

这里在原文的基础上加了AddRootNode()判断服务父节点是否存在，如果不存在则添加一个PERSISTENT的服务父节点，这样虽然启动服务时多了点判断，但是不需要手动命令添加服务父节点了。

关于Zookeeper的使用原理，可以看这里《[ZooKeeper基本原理](http://www.cnblogs.com/luxiaoxun/p/4887452.html" \t "_blank)》。

4、客户端调用服务

使用代理模式调用服务：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53 | public class RpcProxy {        private String serverAddress;      private ServiceDiscovery serviceDiscovery;        public RpcProxy(String serverAddress) {          this.serverAddress = serverAddress;      }        public RpcProxy(ServiceDiscovery serviceDiscovery) {          this.serviceDiscovery = serviceDiscovery;      }        @SuppressWarnings("unchecked")      public <T> T create(Class<?> interfaceClass) {          return (T) Proxy.newProxyInstance(                  interfaceClass.getClassLoader(),                  new Class<?>[]{interfaceClass},                  new InvocationHandler() {                      @Override                      public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {                          RpcRequest request = new RpcRequest();                          request.setRequestId(UUID.randomUUID().toString());                          request.setClassName(method.getDeclaringClass().getName());                          request.setMethodName(method.getName());                          request.setParameterTypes(method.getParameterTypes());                          request.setParameters(args);                            if (serviceDiscovery != null) {                              serverAddress = serviceDiscovery.discover();                          }                          if(serverAddress != null){                              String[] array = serverAddress.split(":");                              String host = array[0];                              int port = Integer.parseInt(array[1]);                                RpcClient client = new RpcClient(host, port);                              RpcResponse response = client.send(request);                                if (response.isError()) {                                  throw new RuntimeException("Response error.",new Throwable(response.getError()));                              } else {                                  return response.getResult();                              }                          }                          else{                              throw new RuntimeException("No server address found!");                          }                      }                  }          );      }  } |

这里每次使用代理远程调用服务，从Zookeeper上获取可用的服务地址，通过RpcClient send一个Request，等待该Request的Response返回。这里原文有个比较严重的bug，在原文给出的简单的Test中是很难测出来的，原文使用了obj的wait和notifyAll来等待Response返回，会出现“假死等待”的情况：一个Request发送出去后，在obj.wait()调用之前可能Response就返回了，这时候在channelRead0里已经拿到了Response并且obj.notifyAll()已经在obj.wait()之前调用了，这时候send后再obj.wait()就出现了假死等待，客户端就一直等待在这里。使用CountDownLatch可以解决这个问题。

注意：这里每次调用的send时候才去和服务端建立连接，使用的是短连接，这种短连接在高并发时会有连接数问题，也会影响性能。

从Zookeeper上获取服务地址：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75 | public class ServiceDiscovery {        private static final Logger LOGGER = LoggerFactory.getLogger(ServiceDiscovery.class);        private CountDownLatch latch = new CountDownLatch(1);        private volatile List<String> dataList = new ArrayList<>();        private String registryAddress;        public ServiceDiscovery(String registryAddress) {          this.registryAddress = registryAddress;          ZooKeeper zk = connectServer();          if (zk != null) {              watchNode(zk);          }      }        public String discover() {          String data = null;          int size = dataList.size();          if (size > 0) {              if (size == 1) {                  data = dataList.get(0);                  LOGGER.debug("using only data: {}", data);              } else {                  data = dataList.get(ThreadLocalRandom.current().nextInt(size));                  LOGGER.debug("using random data: {}", data);              }          }          return data;      }        private ZooKeeper connectServer() {          ZooKeeper zk = null;          try {              zk = new ZooKeeper(registryAddress, Constant.ZK\_SESSION\_TIMEOUT, new Watcher() {                  @Override                  public void process(WatchedEvent event) {                      if (event.getState() == Event.KeeperState.SyncConnected) {                          latch.countDown();                      }                  }              });              latch.await();          } catch (IOException | InterruptedException e) {              LOGGER.error("", e);          }          return zk;      }        private void watchNode(final ZooKeeper zk) {          try {              List<String> nodeList = zk.getChildren(Constant.ZK\_REGISTRY\_PATH, new Watcher() {                  @Override                  public void process(WatchedEvent event) {                      if (event.getType() == Event.EventType.NodeChildrenChanged) {                          watchNode(zk);                      }                  }              });              List<String> dataList = new ArrayList<>();              for (String node : nodeList) {                  byte[] bytes = zk.getData(Constant.ZK\_REGISTRY\_PATH + "/" + node, false, null);                  dataList.add(new String(bytes));              }              LOGGER.debug("node data: {}", dataList);              this.dataList = dataList;          } catch (KeeperException | InterruptedException e) {              LOGGER.error("", e);          }      }  }    ServiceDiscovery |

每次服务地址节点发生变化，都需要再次watchNode，获取新的服务地址列表。

5、消息编码

请求消息：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50 | public class RpcRequest {        private String requestId;      private String className;      private String methodName;      private Class<?>[] parameterTypes;      private Object[] parameters;        public String getRequestId() {          return requestId;      }        public void setRequestId(String requestId) {          this.requestId = requestId;      }        public String getClassName() {          return className;      }        public void setClassName(String className) {          this.className = className;      }        public String getMethodName() {          return methodName;      }        public void setMethodName(String methodName) {          this.methodName = methodName;      }        public Class<?>[] getParameterTypes() {          return parameterTypes;      }        public void setParameterTypes(Class<?>[] parameterTypes) {          this.parameterTypes = parameterTypes;      }        public Object[] getParameters() {          return parameters;      }        public void setParameters(Object[] parameters) {          this.parameters = parameters;      }  }    RpcRequest |

响应消息：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | public class RpcResponse {        private String requestId;      private String error;      private Object result;        public boolean isError() {          return error != null;      }        public String getRequestId() {          return requestId;      }        public void setRequestId(String requestId) {          this.requestId = requestId;      }        public String getError() {          return error;      }        public void setError(String error) {          this.error = error;      }        public Object getResult() {          return result;      }        public void setResult(Object result) {          this.result = result;      }  }    RpcResponse |

消息序列化和反序列化工具：（基于 Protostuff 实现）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54 | public class SerializationUtil {        private static Map<Class<?>, Schema<?>> cachedSchema = new ConcurrentHashMap<>();        private static Objenesis objenesis = new ObjenesisStd(true);        private SerializationUtil() {      }        @SuppressWarnings("unchecked")      private static <T> Schema<T> getSchema(Class<T> cls) {          Schema<T> schema = (Schema<T>) cachedSchema.get(cls);          if (schema == null) {              schema = RuntimeSchema.createFrom(cls);              if (schema != null) {                  cachedSchema.put(cls, schema);              }          }          return schema;      }        /\*\*       \* 序列化（对象 -> 字节数组）       \*/      @SuppressWarnings("unchecked")      public static <T> byte[] serialize(T obj) {          Class<T> cls = (Class<T>) obj.getClass();          LinkedBuffer buffer = LinkedBuffer.allocate(LinkedBuffer.DEFAULT\_BUFFER\_SIZE);          try {              Schema<T> schema = getSchema(cls);              return ProtostuffIOUtil.toByteArray(obj, schema, buffer);          } catch (Exception e) {              throw new IllegalStateException(e.getMessage(), e);          } finally {              buffer.clear();          }      }        /\*\*       \* 反序列化（字节数组 -> 对象）       \*/      public static <T> T deserialize(byte[] data, Class<T> cls) {          try {              T message = (T) objenesis.newInstance(cls);              Schema<T> schema = getSchema(cls);              ProtostuffIOUtil.mergeFrom(data, message, schema);              return message;          } catch (Exception e) {              throw new IllegalStateException(e.getMessage(), e);          }      }  }    SerializationUtil |

由于处理的是TCP消息，本人加了TCP的粘包处理Handler

|  |  |
| --- | --- |
| 1 | channel.pipeline().addLast(new LengthFieldBasedFrameDecoder(65536,0,4,0,0)) |

消息编解码时开始4个字节表示消息的长度，也就是消息编码的时候，先写消息的长度，再写消息。

6、性能改进

1）服务端请求异步处理

Netty本身就是一个高性能的网络框架，从网络IO方面来说并没有太大的问题。

从这个RPC框架本身来说，在原文的基础上把Server端处理请求的过程改成了多线程异步：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | public void channelRead0(final ChannelHandlerContext ctx,final RpcRequest request) throws Exception {          RpcServer.submit(new Runnable() {              @Override              public void run() {                  LOGGER.debug("Receive request " + request.getRequestId());                  RpcResponse response = new RpcResponse();                  response.setRequestId(request.getRequestId());                  try {                      Object result = handle(request);                      response.setResult(result);                  } catch (Throwable t) {                      response.setError(t.toString());                      LOGGER.error("RPC Server handle request error",t);                  }                  ctx.writeAndFlush(response).addListener(ChannelFutureListener.CLOSE).addListener(new ChannelFutureListener() {                      @Override                      public void operationComplete(ChannelFuture channelFuture) throws Exception {                          LOGGER.debug("Send response for request " + request.getRequestId());                      }                  });              }          });      } |

Netty 4中的Handler处理在IO线程中，如果Handler处理中有耗时的操作（如数据库相关），会让IO线程等待，影响性能。

2）服务端长连接的管理

客户端保持和服务进行长连接，不需要每次调用服务的时候进行连接，长连接的管理（通过Zookeeper获取有效的地址）。

通过监听Zookeeper服务节点值的变化，动态更新客户端和服务端保持的长连接。这个事情现在放在客户端在做，客户端保持了和所有可用服务的长连接，给客户端和服务端都造成了压力，需要解耦这个实现。

3）客户端请求异步处理

客户端请求异步处理的支持，不需要同步等待：发送一个异步请求，返回Feature，通过Feature的callback机制获取结果。

|  |  |
| --- | --- |
| 1  2  3 | IAsyncObjectProxy client = rpcClient.createAsync(HelloService.class);  RPCFuture helloFuture = client.call("hello", Integer.toString(i));  String result = (String) helloFuture.get(3000, TimeUnit.MILLISECONDS); |

个人觉得该RPC的待改进项：

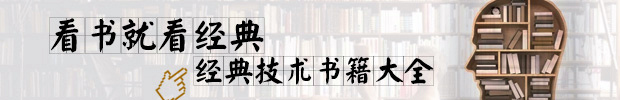
编码序列化的多协议支持。

项目持续更新中。

项目地址：https://github.com/luxiaoxun/NettyRpc

参考：

* 轻量级分布式 RPC 框架：http://my.oschina.net/huangyong/blog/361751
* 你应该知道的RPC原理：http://www.cnblogs.com/LBSer/p/4853234.html

[](http://blog.jobbole.com/106093/)

相关文章

* [RPC原理及RPC实例分析](http://www.importnew.com/22003.html)
* [远程过程调用(RPC)详解](http://www.importnew.com/21660.html)
* [轻量级分布式 RPC 框架](http://www.importnew.com/20327.html)
* [Java的内存回收机制](http://www.importnew.com/15330.html)
* [微信开发之入门教程](http://www.importnew.com/18769.html)
* [写程序很难之去除字符串的空白字符](http://www.importnew.com/22956.html)
* [七种WebSocket框架的性能比较](http://www.importnew.com/23286.html)
* [JVM源码分析之新生代DefNewGeneration的实现](http://www.importnew.com/23683.html)
* [Java Selenium (六) XPath 定位](http://www.importnew.com/25488.html)
* [G1 垃圾收集器之对象分配过程](http://www.importnew.com/27859.html)