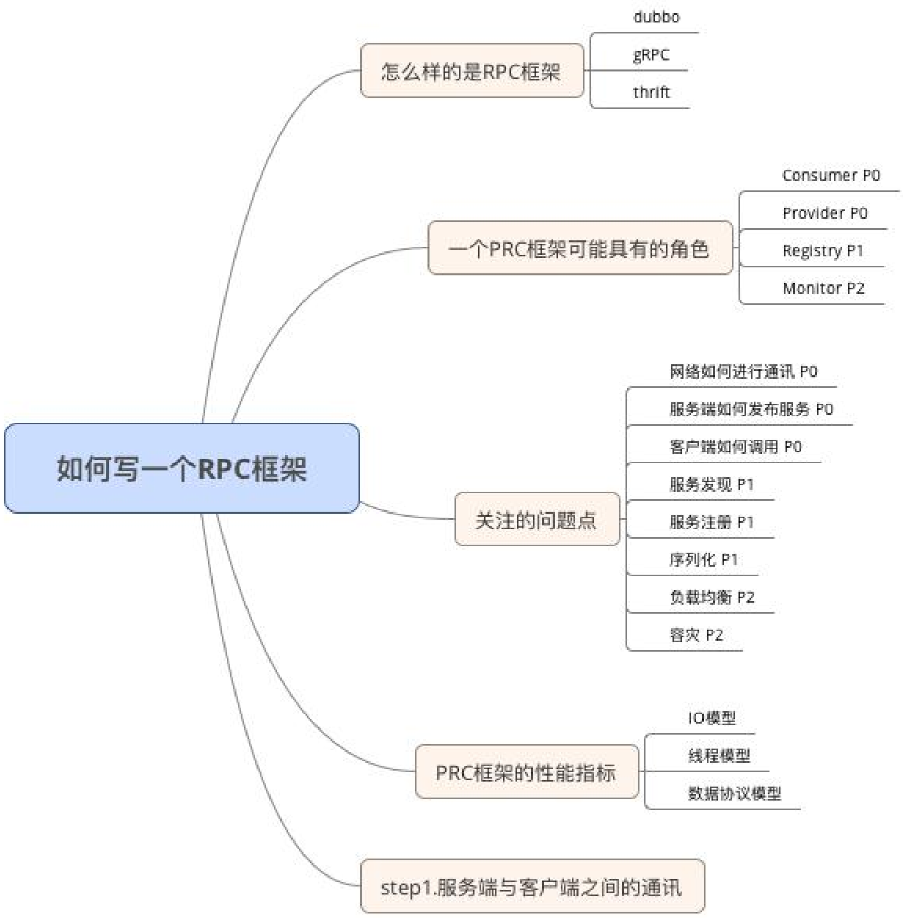
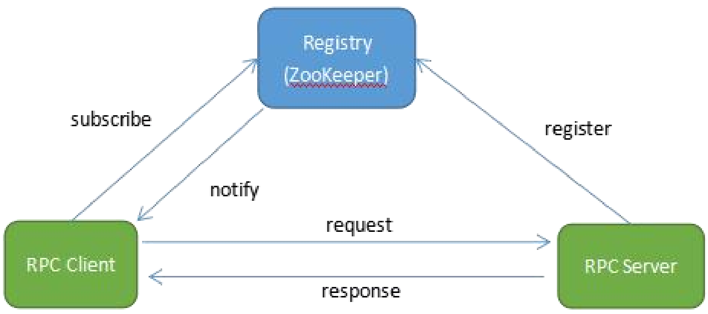
轻量级分布式 Rpc 框架的实现

Rpc 可以基于 HTTP 或者 TCP 协议，Web Service 就是基于 HTTP 协议的 RPC，但是其性能比如基于 TCP 协议。为了实现 RPC 框架，思路如下所示：



典型的 Rpc 整体框架如下图所示：



Rpc Server 将服务部署在分布式环境下的不同节点，然后通过服务注册的方式，让客户端自动来发现当前可用的服务，并调用这些服务。

实现的轻量级 Rpc 架构如上图，使用以下技术：

- Spring，强大的依赖注入框架
- Netty， NIO 编程技术库
- Protostuff，基于 Protobuf 序列化框架，面向 POJO，无需编写.proto 文件
- Zookeeper，提供服务注册与发现功能

具体过程如下

1. 定义 RPC 协议

通过协议定义客户端和服务端可以理解的消息传输结构

1) 客户端请求消息，RpcRequest，定义如下：

```
public class RpcRequest {
        private String requestId;    //请求标识
        private String className;    //请求类名
        private String methodName; //方法名
        private Class<?>[] parameterTypes;    //参数类型
        private Object[] parameters;      //参数值
}
```

2) 服务端响应消息，RpcResponse，定义如下：

```
public class RpcResponse {
        private String requestId;
        private String error;
        private Object result;
}
```

客户端（RpcClientHandler）通过动态代理将 RpcRequest 发送给服务端，如下：

```
public RPCFuture sendRequest(RpcRequest request) {
    final CountDownLatch latch = new CountDownLatch(1);
    RPCFuture rpcFuture = new RPCFuture(request);
    pendingRPC.put(request.getRequestId(), rpcFuture);
    channel.writeAndFlush(request).addListener(new ChannelFutureListener() {
            @Override
            public void operationComplete(ChannelFuture future) {
                    latch.countDown();
            }
    }); .....
    return rpcFuture;
}
```

服务端(RpcHandler)接收到 RpcRequest 后进行处理，并响应 RpcResponse，如下：

```
@Override
public void channelRead0(final ChannelHandlerContext ctx,final RpcRequest request)
throws Exception {
  RpcServer.submit(new Runnable() {
    @Override
    public void run() {
        RpcResponse response = new RpcResponse();
        response.setRequestId(request.getRequestId());
        try {
            Object result = handle(request); //处理 RpcRequest
            response.setResult(result);      //设置 RcpResponse
        } ...   //发送消息信息
        ctx.writeAndFlush(response).addListener(new ChannelFutureListener() {
            @Override
            public void operationComplete(ChannelFuture channelFuture)   {
                logger.debug("Send response for request " + request.getRequestId());
            }
        });
    }
  });
}
```

2. 序列化

RpcRequest 和 RpcResponse 的发送都要经过网路传输，因此要经过序列化后形成传输字节，本处使用 Netty 作为 NIO，其序列化及反序列化在 RpcEncoder 和 RpcDecoder 中进行处理，这里使用 Protostuff 作为序列化工具

1) 序列化-RpcEncoder

```
cp.addLast(new RpcEncoder(RpcRequest.class)); //定义其序列化类 RpcRequest
```

序列化操作如下：

```
@Override   //Object in <= RpcRequest
public void encode(ChannelHandlerContext ctx, Object in, ByteBuf out) throws Exception {
  if (genericClass.isInstance(in)) {
      byte[] data = SerializationUtil.serialize(in);
      out.writeInt(data.length);
      out.writeBytes(data);
  }
}
```

2) 反序列化-RpcDecoder，处理 RpcResponse

```
cp.addLast(new RpcDecoder(RpcResponse.class));
```

其反序列化如下：

```
@Override
public final void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out)   {
    if (in.readableBytes() < 4) {
        return;
    }
    in.markReaderIndex();
    int dataLength = in.readInt();
    if (in.readableBytes() < dataLength) {
        in.resetReaderIndex();
        return;
    }
    byte[] data = new byte[dataLength];
    in.readBytes(data);
    Object obj = SerializationUtil.deserialize(data, genericClass);
    out.add(obj);
}
```

3) 序列化操作在类中 SerializationUtil 中实现，使用 Protostuff 类库

```
public class SerializationUtil {
    private static Map<Class<?>, Schema<?>> cachedSchema = new
ConcurrentHashMap<>();
    private static Objenesis objenesis = new ObjenesisStd(true);

    @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;
}

    @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);
        } .....
    }

    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;
        } ......
    }
}
```

3. 网络通信层

SimpleRpc 使用 Netty 作为通信框架，其序列化及反序列化通过 RpcEncoder 和 RpcDecoder 来实现

1) Server 端初始化，核心是 RpcHandler，RpcServer 初始化如下：

```
public void start() throws Exception {
  if (bossGroup == null && workerGroup == null) {
    bossGroup = new NioEventLoopGroup();
    workerGroup = new NioEventLoopGroup();
    ServerBootstrap bootstrap = new ServerBootstrap();
    bootstrap.group(bossGroup, workerGroup).channel(NioServerSocketChannel.class)
            .childHandler(new ChannelInitializer<SocketChannel>() {
                @Override
                public void initChannel(SocketChannel channel) throws Exception {
                    channel.pipeline()
                      .addLast(new LengthFieldBasedFrameDecoder(65536, 0, 4, 0, 0))
                      .addLast(new RpcDecoder(RpcRequest.class))
                      .addLast(new RpcEncoder(RpcResponse.class))
                      .addLast(new RpcHandler(handlerMap));
                }
            })
            .option(ChannelOption.SO_BACKLOG, 128)
            .childOption(ChannelOption.SO_KEEPALIVE, true);

    String host = array[0]
    int port = Integer.parseInt(array[1]);
```

```
        ChannelFuture future = bootstrap.bind(host, port).sync();
        future.channel().closeFuture().sync();
    }
}
```

2) Client 端初始化

```
public class RpcClientInitializer extends ChannelInitializer<SocketChannel> {
    @Override
    protected void initChannel(SocketChannel socketChannel) throws Exception {
            ChannelPipeline cp = socketChannel.pipeline();
            cp.addLast(new RpcEncoder(RpcRequest.class));
            cp.addLast(new LengthFieldBasedFrameDecoder(65536, 0, 4, 0, 0));
            cp.addLast(new RpcDecoder(RpcResponse.class));
            cp.addLast(new RpcClientHandler());
    }
}
```

```
public void run() {
  Bootstrap b = new Bootstrap();
  b.group(eventLoopGroup)
      .channel(NioSocketChannel.class)
      .handler(new RpcClientInitializer());
  ChannelFuture channelFuture = b.connect(remotePeer);...
}
```

其处理过程不再详述，见源码

4. 服务注册与发现

使用 Zookeeper 实现服务注册与发现功能

1) 注册，Server 端启动时将其注册到 Zookeeper 中

```
public void start() throws Exception {
  ......
  if (serviceRegistry != null) {
        serviceRegistry.register(serverAddress);
  }
}
```

在 ZK 中，信息如下：

```
[zk:] get /registry/data0000000004
127.0.0.1:18866
```

2) 发现，客户端使用时从 Zookeeper 中获取服务地址

```
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() == Watcher.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));
    }
    this.dataList = dataList;
    updateConnectedServer();
  }
}
```

在 updateConnectedServer 中将服务端地址发布到 ConnectionManager 中

```
private void connectServerNode(final InetSocketAddress remotePeer) {
   threadPoolExecutor.submit(new Runnable() {
     @Override
     public void run() {
       Bootstrap b = new Bootstrap();
       b.group(eventLoopGroup)
              .channel(NioSocketChannel.class)
              .handler(new RpcClientInitializer());
       ChannelFuture channelFuture = b.connect(remotePeer);
       channelFuture.addListener(new ChannelFutureListener() {
         @Override
         public void operationComplete(final ChannelFuture channelFuture)
               RpcClientHandler handler =
                       channelFuture.channel().pipeline().get(RpcClientHandler.class);
               addHandler(handler);
           }
         } }); } });}
```

5. 负载均衡

启动多个 Server 后，客户端提交程序时从中选择某个 Server，从而实现负载均衡，

```
ObjectProxy#
@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);
    ......
    RpcClientHandler handler = ConnectManager.getInstance().chooseHandler();
    RPCFuture rpcFuture = handler.sendRequest(request);
    return rpcFuture.get();
}
```

在 ConnectManager#chooseHandler 中实现负载均衡

```
public RpcClientHandler chooseHandler() {
    int size = connectedHandlers.size();
    ......
    int index = (roundRobin.getAndAdd(1) + size) % size;
    return connectedHandlers.get(index);
}
```

参考链接:

http://www.cnblogs.com/LBSer/p/4853234.html
https://github.com/luxiaoxun/NettyRpc
http://www.cnblogs.com/luxiaoxun/p/5272384.html
https://my.oschina.net/huangyong/blog/361751
https://github.com/SimonHunag/simpleRPC