crazyhzm 265

Dubbo源码解析（三十二）远程调用——thrift协议

dubbo java 阅读约 72 分钟

远程调用——thrift协议

> 目标：介绍thrift协议的设计和实现，介绍dubbo-rpc-thrift的源码。

前言

dubbo集成thrift协议，是基于Thrift来实现的，Thrift是一种轻量级，与语言无关的软件堆栈，具有用于点对点RPC的相关代码生成机制。Thrift为数据传输，数据序列化和应用程序级处理提供了清晰的抽象。代码生成系统采用简单的定义语言作为输入，并跨编程语言生成代码，使用抽象堆栈构建可互操作的RPC客户端和服务器。

源码分析

（一）MultiServiceProcessor

该类对输入流进行操作并写入某些输出流。它实现了TProcessor接口，关键的方法是process。

```
@Override
public boolean process(TProtocol in, TProtocol out) throws TException {

    // 获得十六进制的魔数
    short magic = in.readI16();

    // 如果不是规定的魔数，则打印错误日志，返回false
    if (magic != ThriftCodec.MAGIC) {
        logger.error("Unsupported magic " + magic);
        return false;
    }

    // 获得三十二进制魔数
    in.readI32();
    // 获得十六进制魔数
    in.readI16();
    // 获得版本
    byte version = in.readByte();
    // 获得服务名
    String serviceName = in.readString();
    // 获得id
    long id = in.readI64();

    ByteArrayOutputStream bos = new ByteArrayOutputStream(1024);

    // 创建基础运输TIOStreamTransport对象
```

（二）RandomAccessByteArrayOutputStream

该类是随机访问数组的输出流，比较简单，我就不多叙述，有兴趣的可以直接看源码，不看影响也不大。

（三）ClassNameGenerator

```
@SPI(DubboClassNameGenerator.NAME)
public interface ClassNameGenerator {

    /**
     * 生成参数的类名
     */
    public String generateArgsClassName(String serviceName, String methodName);

    /**
     * 生成结果的类名
     * @param serviceName
     * @param methodName
     * @return
     */
    public String generateResultClassName(String serviceName, String methodName);

}
```

该接口是是可扩展接口，定义了两个方法。有两个实现类，下面讲述。

（四）DubboClassNameGenerator

该类实现了ClassNameGenerator接口，是dubbo相关的类名生成实现。

```
public class DubboClassNameGenerator implements ClassNameGenerator {

    public static final String NAME = "dubbo";

    @Override
    public String generateArgsClassName(String serviceName, String methodName) {
        return ThriftUtils.generateMethodArgsClassName(serviceName, methodName);
    }

    @Override
    public String generateResultClassName(String serviceName, String methodName) {
        return ThriftUtils.generateMethodResultClassName(serviceName, methodName);
    }

}
```

（五）ThriftClassNameGenerator

该类实现了ClassNameGenerator接口，是Thrift相关的类名生成实现。

```
public class ThriftClassNameGenerator implements ClassNameGenerator {

    public static final String NAME = "thrift";

    @Override
    public String generateArgsClassName(String serviceName, String methodName) {
        return ThriftUtils.generateMethodArgsClassNameThrift(serviceName, methodName);
    }

    @Override
    public String generateResultClassName(String serviceName, String methodName) {
        return ThriftUtils.generateMethodResultClassNameThrift(serviceName, methodName);
    }

}
```

以上两个都调用了ThriftUtils中的方法。

（六）ThriftUtils

该类中封装的方法比较简单，就一些字符串的拼接，有兴趣的可以直接查看我下面贴出来的注释连接。

（七）ThriftCodec

该类是基于Thrift实现的编解码器。 这里需要大家看一下该类的注释，关于协议的数据：

```
* |<-                                 message header                                  ->|<- message body ->|
* +----------------+----------------------+-----------------+--------------------------+------------------+
* | magic (2 bytes)|message size (4 bytes)|head size(2 bytes)| version (1 byte) | header |   message body   |
* +----------------+----------------------+-----------------+--------------------------+------------------+
* |<-
```

1.属性

```
/**
 * 消息长度索引
 */
public static final int MESSAGE_LENGTH_INDEX = 2;
/**
 * 消息头长度索引
 */
public static final int MESSAGE_HEADER_LENGTH_INDEX = 6;
/**
 * 消息最短长度
 */
public static final int MESSAGE_SHORTEST_LENGTH = 10;
public static final String NAME = "thrift";
/**
 * 类名生成参数
 */
public static final String PARAMETER_CLASS_NAME_GENERATOR = "class.name.generator";
/**
 * 版本
 */
public static final byte VERSION = (byte) 1;
/**
 * 魔数
 */
public static final short MAGIC = (short) 0xdabc;
/**
```

2.encode

```
@Override
public void encode(Channel channel, ChannelBuffer buffer, Object message)
        throws IOException {

    // 如果消息是Request类型
    if (message instanceof Request) {
        // Request类型消息编码
        encodeRequest(channel, buffer, (Request) message);
    } else if (message instanceof Response) {
        // Response类型消息编码
        encodeResponse(channel, buffer, (Response) message);
    } else {
        throw new UnsupportedOperationException("Thrift codec only support encode "
                + Request.class.getName() + " and " + Response.class.getName());
    }

}
```

该方法是编码的逻辑，具体的编码操作根据请求类型不同分别调用不同的方法。

3.encodeRequest

```
private void encodeRequest(Channel channel, ChannelBuffer buffer, Request request)
        throws IOException {

    // 获得会话域
    RpcInvocation inv = (RpcInvocation) request.getData();

    // 获得下一个id
    int seqId = nextSeqId();

    // 获得服务名
    String serviceName = inv.getAttachment(Constants.INTERFACE_KEY);

    // 如果是空的 则抛出异常
    if (StringUtils.isEmpty(serviceName)) {
        throw new IllegalArgumentException("Could not find service name in attachment with key "
                + Constants.INTERFACE_KEY);
    }

    // 创建TMessage对象
    TMessage message = new TMessage(
            inv.getMethodName(),
            TMessageType.CALL,
            seqId);

    // 获得方法参数
    String methodArgs = ExtensionLoader.getExtensionLoader(ClassNameGenerator.class)
```

该方法是对request类型的消息进行编码。

4.encodeResponse

```
private void encodeResponse(Channel channel, ChannelBuffer buffer, Response response)
        throws IOException {

    // 获得结果
    RpcResult result = (RpcResult) response.getResult();

    // 获得请求
    RequestData rd = cachedRequest.get(response.getId());

    // 获得结果的类名
    String resultClassName = ExtensionLoader.getExtensionLoader(ClassNameGenerator.class).getExtension(
            channel.getUrl().getParameter(ThriftConstants.CLASS_NAME_GENERATOR_KEY,
ThriftClassNameGenerator.NAME))
            .generateResultClassName(rd.serviceName, rd.methodName);

    // 如果为空，则序列化失败
    if (StringUtils.isEmpty(resultClassName)) {
        throw new RpcException(RpcException.SERIALIZATION_EXCEPTION,
                "Could not encode response, the specified interface may be incorrect.");
    }

    // 获得类型
    Class clazz = cachedClass.get(resultClassName);

    // 如果为空，则重新获取
    if (clazz == null) {
```

该方法是对response类型的请求消息进行编码。

5.decode

```
    @Override
    public Object decode(Channel channel, ChannelBuffer buffer) throws IOException {

        int available = buffer.readableBytes();

        // 如果小于最小的长度，则还需要更多的输入
        if (available < MESSAGE_SHORTEST_LENGTH) {

            return DecodeResult.NEED_MORE_INPUT;

        } else {

            TIOStreamTransport transport = new TIOStreamTransport(new ChannelBufferInputStream(buffer));

            TBinaryProtocol protocol = new TBinaryProtocol(transport);

            short magic;
            int messageLength;

            // 对协议头中的魔数进行比对
            try {
//                protocol.readI32(); // skip the first message length
                byte[] bytes = new byte[4];
                transport.read(bytes, 0, 4);
                magic = protocol.readI16();
```

该方法是对解码的逻辑。对于消息分为REPLY、EXCEPTION和CALL三种情况来分别进行解码。

6.RequestData

```
static class RequestData {
    /**
     * 请求id
     */
    int id;
    /**
     * 服务名
     */
    String serviceName;
    /**
     * 方法名
     */
    String methodName;

    static RequestData create(int id, String sn, String mn) {
        RequestData result = new RequestData();
        result.id = id;
        result.serviceName = sn;
        result.methodName = mn;
        return result;
    }

}
```

该内部类是请求参数实体。

（八）ThriftInvoker

该类是thrift协议的Invoker实现。

1.属性

```
/**
 * 客户端集合
 */
private final ExchangeClient[] clients;

/**
 * 活跃的客户端索引
 */
private final AtomicPositiveInteger index = new AtomicPositiveInteger();

/**
 * 销毁锁
 */
private final ReentrantLock destroyLock = new ReentrantLock();

/**
 * invoker集合
 */
private final Set<Invoker<?>> invokers;
```

2.doInvoke

```
@Override
protected Result doInvoke(Invocation invocation) throws Throwable {

    RpcInvocation inv = (RpcInvocation) invocation;

    final String methodName;

    // 获得方法名
    methodName = invocation.getMethodName();

    // 设置附加值 path
    inv.setAttachment(Constants.PATH_KEY, getUrl().getPath());

    // for thrift codec
    inv.setAttachment(ThriftCodec.PARAMETER_CLASS_NAME_GENERATOR, getUrl().getParameter(
            ThriftCodec.PARAMETER_CLASS_NAME_GENERATOR, DubboClassNameGenerator.NAME));

    ExchangeClient currentClient;

    // 如果只有一个连接的客户端，则直接返回
    if (clients.length == 1) {
        currentClient = clients[0];
    } else {
        // 否则，取出下一个客户端，循环数组取
        currentClient = clients[index.getAndIncrement() % clients.length];
    }
```

该方法是thrift协议的调用链处理逻辑。

（九）ThriftProtocol

该类是thrift协议的主要实现逻辑，分别实现了服务引用和服务调用的逻辑。

1.属性

```
/**
 * 默认端口号
 */
public static final int DEFAULT_PORT = 40880;

/**
 * 扩展名
 */
public static final String NAME = "thrift";

// ip:port -> ExchangeServer
/**
 * 服务集合，key为ip:port
 */
private final ConcurrentMap<String, ExchangeServer> serverMap =
        new ConcurrentHashMap<String, ExchangeServer>();

private ExchangeHandler handler = new ExchangeHandlerAdapter() {

    @Override
    public Object reply(ExchangeChannel channel, Object msg) throws RemotingException {

        // 如果消息是Invocation类型的
        if (msg instanceof Invocation) {
            Invocation inv = (Invocation) msg;
            // 获得服务名
```

2.export

```
@Override
public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {

    // can use thrift codec only
    // 只能使用thrift编解码器
    URL url = invoker.getUrl().addParameter(Constants.CODEC_KEY, ThriftCodec.NAME);
    // find server.
    // 获得服务地址
    String key = url.getAddress();
    // client can expose a service for server to invoke only.
    // 客户端可以为服务器暴露服务以仅调用
    boolean isServer = url.getParameter(Constants.IS_SERVER_KEY, true);
    if (isServer && !serverMap.containsKey(key)) {
        // 加入到集合
        serverMap.put(key, getServer(url));
    }
    // export service.
    // 得到服务key
    key = serviceKey(url);
    // 创建暴露者
    DubboExporter<T> exporter = new DubboExporter<T>(invoker, key, exporterMap);
    // 加入集合
    exporterMap.put(key, exporter);

    return exporter;
}
```

该方法是服务暴露的逻辑实现。

3.refer

```
@Override
public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {

    // 创建ThriftInvoker
    ThriftInvoker<T> invoker = new ThriftInvoker<T>(type, url, getClients(url), invokers);

    // 加入到集合
    invokers.add(invoker);

    return invoker;

}
```

该方法是服务引用的逻辑实现。

4.getClients

```
private ExchangeClient[] getClients(URL url) {

    // 获得连接数
    int connections = url.getParameter(Constants.CONNECTIONS_KEY, 1);

    // 创建客户端集合
    ExchangeClient[] clients = new ExchangeClient[connections];

    // 创建客户端
    for (int i = 0; i < clients.length; i++) {
        clients[i] = initClient(url);
    }
    return clients;
}
```

该方法是获得客户端集合。

5.initClient

```
private ExchangeClient initClient(URL url) {

    ExchangeClient client;

    // 加上编解码器
    url = url.addParameter(Constants.CODEC_KEY, ThriftCodec.NAME);

    try {
        // 创建客户端
        client = Exchangers.connect(url);
    } catch (RemotingException e) {
        throw new RpcException("Fail to create remoting client for service(" + url
                + "): " + e.getMessage(), e);
    }

    return client;

}
```

该方法是创建客户端的逻辑。

6.getServer

```
private ExchangeServer getServer(URL url) {
    // enable sending readonly event when server closes by default
    // 加入只读事件
    url = url.addParameterIfAbsent(Constants.CHANNEL_READONLYEVENT_SENT_KEY, Boolean.TRUE.toString());
    // 获得服务的实现方式
    String str = url.getParameter(Constants.SERVER_KEY, Constants.DEFAULT_REMOTING_SERVER);

    // 如果该实现方式不是dubbo支持的方式，则抛出异常
    if (str != null && str.length() > 0 &&
!ExtensionLoader.getExtensionLoader(Transporter.class).hasExtension(str))
        throw new RpcException("Unsupported server type: " + str + ", url: " + url);

    ExchangeServer server;
    try {
        // 获得服务器
        server = Exchangers.bind(url, handler);
    } catch (RemotingException e) {
        throw new RpcException("Fail to start server(url: " + url + ") " + e.getMessage(), e);
    }
    // 获得实现方式
    str = url.getParameter(Constants.CLIENT_KEY);
    // 如果客户端实现方式不是dubbo支持的方式，则抛出异常。
    if (str != null && str.length() > 0) {
        Set<String> supportedTypes =
ExtensionLoader.getExtensionLoader(Transporter.class).getSupportedExtensions();
        if (!supportedTypes.contains(str)) {
```

该方法是获得server的逻辑实现。

后记

> 该部分相关的源码解析地址：https://github.com/CrazyHZM/i...

该文章讲解了远程调用中关于thrift协议实现的部分，要对Thrift。接下来我将开始对rpc模块关于webservice协议部分进行讲解。

阅读 793 • 更新于 11月8日

赞 2　收藏 1　¥ 赞赏　分享

本作品系 原创 ， 作者保留所有权利，未经作者允许，禁止转载和演绎

crazyhzm
265

关注作者

0 条评论

得票 • 时间

撰写评论 ...

提交评论

推荐阅读

聊聊Dubbo - Dubbo可扩展机制源码解析

摘要： 在Dubbo可扩展机制实战中，我们了解了Dubbo扩展机制的一些概念，初探了Dubbo中LoadBalance的实现，并自己实现了...

猫耳 • 阅读 21