crazyhzm 265

Dubbo源码解析（二十四）远程调用——dubbo协议

dubbo java 阅读约 149 分钟

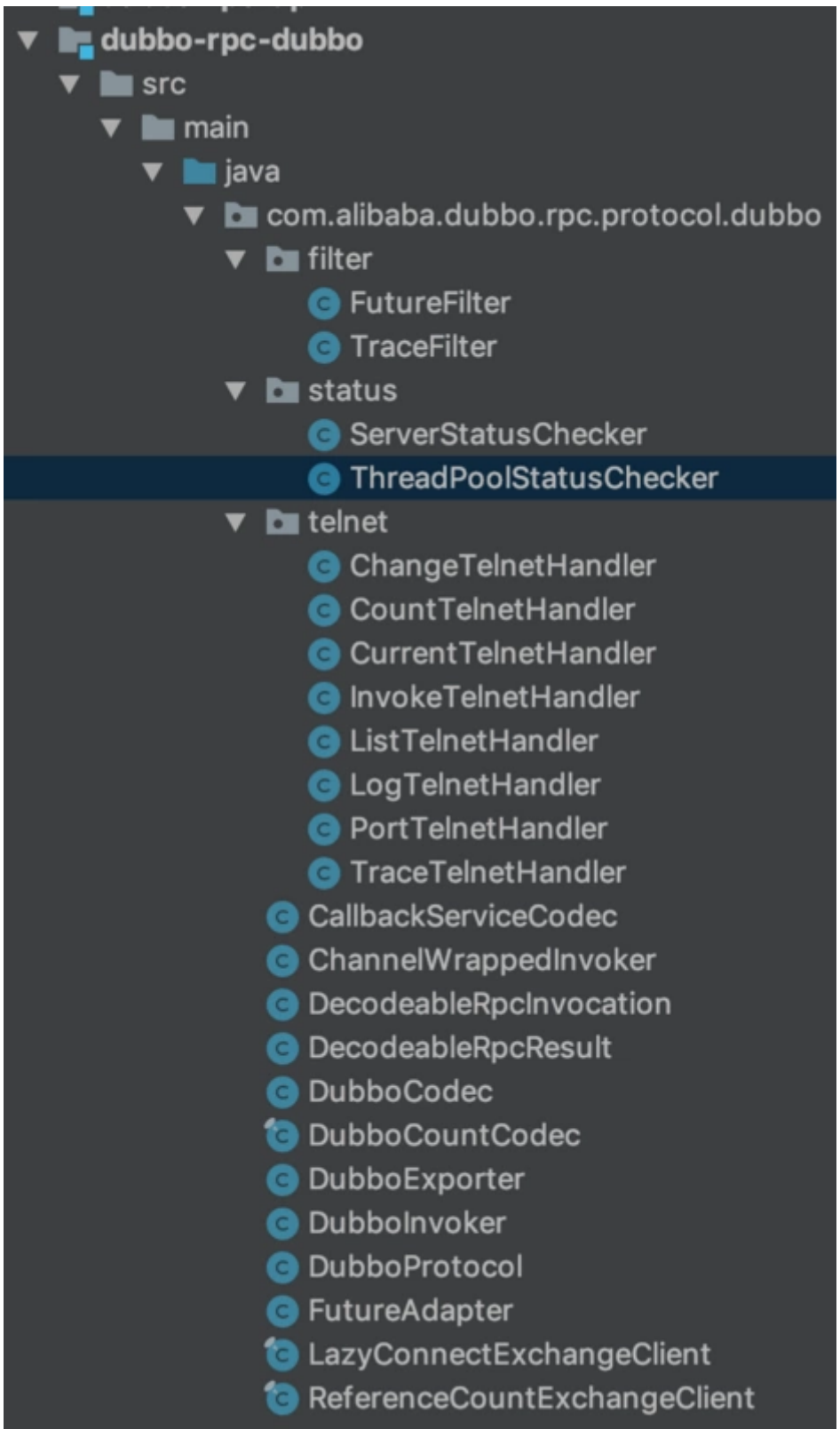
远程调用——dubbo协议

> 目标：介绍远程调用中跟dubbo协议相关的设计和实现，介绍dubbo-rpc-dubbo的源码。

前言

Dubbo 缺省协议采用单一长连接和 NIO 异步通讯，适合于小数据量大并发的服务调用，以及服务消费者机器数远大于服务提供者机器数的情况。反之，Dubbo 缺省协议不适合传送大数据量的服务，比如传文件，传视频等，除非请求量很低。这是官方文档的原话，并且官方文档还介绍了为什么使用单一长连接和 NIO 异步通讯以及为什么不适合传输大数据的服务。我就不赘述了。

我们先来看看dubbo-rpc-dubbo下的包结构：



1. filter：该包下面是对于dubbo协议独有的两个过滤器
2. status：该包下是做了对于服务和线程池状态的检测
3. telnet：该包下是对于telnet命令的支持
4. 最外层：最外层是dubbo协议的核心

源码分析

（一）DubboInvoker

该类是dubbo协议独自实现的的invoker，其中实现了调用方法的三种模式，分别是异步发送、单向发送和同步发送，具体在下面介绍。

1.属性

```
/**
 * 信息交换客户端数组
 */
private final ExchangeClient[] clients;

/**
 * 客户端数组位置
 */
private final AtomicPositiveInteger index = new AtomicPositiveInteger();

/**
 * 版本号
 */
private final String version;

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

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

2.doInvoke

```
@Override
protected Result doInvoke(final Invocation invocation) throws Throwable {
    // rpc会话域
    RpcInvocation inv = (RpcInvocation) invocation;
    // 获得方法名
    final String methodName = RpcUtils.getMethodName(invocation);
    // 把path放入到附加值中
    inv.setAttachment(Constants.PATH_KEY, getUrl().getPath());
    // 把版本号放入到附加值
    inv.setAttachment(Constants.VERSION_KEY, version);

    // 当前的客户端
    ExchangeClient currentClient;
    // 如果数组内就一个客户端，则直接取出
    if (clients.length == 1) {
        currentClient = clients[0];
    } else {
        // 取模轮询 从数组中取，当取到最后一个时，从头开始
        currentClient = clients[index.getAndIncrement() % clients.length];
    }
    try {
        // 是否启用异步
        boolean isAsync = RpcUtils.isAsync(getUrl(), invocation);
        // 是否是单向发送
        boolean isOneway = RpcUtils.isOneway(getUrl(), invocation);
        // 获得超时时间
```

在调用invoker的时候，通过远程通信将Invocation信息传递给服务端，服务端在接收到该invocation信息后，要找到对应的本地方法，然后通过反射执行该方法，将方法的执行结果返回给客户端，在这里，客户端发送有三种模式：

1. 异步发送，也就是当我发送调用后，我不阻塞等待结果，直接返回，将返回的future保存到上下文，方便后期使用。
2. 单向发送，执行方法不需要返回结果。
3. 同步发送，执行方法后，等待结果返回，否则一直阻塞。

3.isAvailable

```
@Override
public boolean isAvailable() {
    if (!super.isAvailable())
        return false;
    for (ExchangeClient client : clients) {
        // 只要有一个客户端连接并且不是只读，则表示存活
        if (client.isConnected() && !client.hasAttribute(Constants.CHANNEL_ATTRIBUTE_READONLY_KEY)) {
            //cannot write == not Available ?
            return true;
        }
    }
    return false;
}
```

该方法是检查服务端是否存活。

4.destroy

```
@Override
public void destroy() {
    // in order to avoid closing a client multiple times, a counter is used in case of connection per jvm, every
    // time when client.close() is called, counter counts down once, and when counter reaches zero, client will
be
    // closed.
    if (super.isDestroyed()) {
        return;
    } else {
        // double check to avoid dup close
        // 获得销毁锁
        destroyLock.lock();
        try {
            if (super.isDestroyed()) {
                return;
            }
            // 销毁
            super.destroy();
            // 从集合中移除
            if (invokers != null) {
                invokers.remove(this);
            }
            for (ExchangeClient client : clients) {
                try {
                    // 关闭每一个客户端
                    client.close(ConfigUtils.getServerShutdownTimeout());
```

该方法是销毁服务端，关闭所有连接到远程通信客户端。

（二）DubboExporter

该类继承了AbstractExporter，是dubbo协议中独有的服务暴露者。

```
/**
 * 服务key
 */
private final String key;

/**
 * 服务暴露者集合
 */
private final Map<String, Exporter<?>> exporterMap;

public DubboExporter(Invoker<T> invoker, String key, Map<String, Exporter<?>> exporterMap) {
    super(invoker);
    this.key = key;
    this.exporterMap = exporterMap;
}

@Override
public void unexport() {
    super.unexport();
    // 从集合中移除该key
    exporterMap.remove(key);
}
```

其中对于服务暴露者用集合做了缓存，并且只重写了了unexport。

（三）DubboProtocol

该类是dubbo协议的核心实现，其中增加了比如延迟加载等处理。 并且其中还包括了对服务暴露和服务引用的逻辑处理。

1.属性

```
public static final String NAME = "dubbo";

/**
 * 默认端口号
 */
public static final int DEFAULT_PORT = 20880;
/**
 * 回调名称
 */
private static final String IS_CALLBACK_SERVICE_INVOKE = "_isCallBackServiceInvoke";
/**
 * dubbo协议的单例
 */
private static DubboProtocol INSTANCE;
/**
 * 信息交换服务器集合 key：host:port  value：ExchangeServer
 */
private final Map<String, ExchangeServer> serverMap = new ConcurrentHashMap<String, ExchangeServer>(); //
<host:port,Exchanger>
/**
 * 信息交换客户端集合
 */
private final Map<String, ReferenceCountExchangeClient> referenceClientMap = new ConcurrentHashMap<String,
ReferenceCountExchangeClient>(); // <host:port,Exchanger>
/**
 * 懒加载的客户端集合
```

该属性中关键的是实例化了一个请求处理器，其中实现了基于dubbo协议等连接、取消连接、回复请求结果等方法。

2.export

```
@Override
public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
    URL url = invoker.getUrl();

    // export service.
    // 得到服务key  group+"/"+serviceName+":"+serviceVersion+":"+port
    String key = serviceKey(url);
    // 创建exporter
    DubboExporter<T> exporter = new DubboExporter<T>(invoker, key, exporterMap);
    // 加入到集合
    exporterMap.put(key, exporter);

    //export an stub service for dispatching event
    Boolean isStubSupportEvent = url.getParameter(Constants.STUB_EVENT_KEY, Constants.DEFAULT_STUB_EVENT);
    Boolean isCallbackservice = url.getParameter(Constants.IS_CALLBACK_SERVICE, false);
    // 如果是本地存根事件而不是回调服务
    if (isStubSupportEvent && !isCallbackservice) {
        // 获得本地存根的方法
        String stubServiceMethods = url.getParameter(Constants.STUB_EVENT_METHODS_KEY);
        // 如果为空，则抛出异常
        if (stubServiceMethods == null || stubServiceMethods.length() == 0) {
            if (logger.isWarnEnabled()) {
                logger.warn(new IllegalStateException("consumer [" + url.getParameter(Constants.INTERFACE_KEY) +
                        "], has set stubproxy support event ,but no stub methods founded."));
            }
        } else {
```

该方法是基于dubbo协议的服务暴露，除了对于存根服务和本地服务进行标记以外，打开服务和序列化分别在openServer和optimizeSerialization中实现。

3.openServer

```
private void openServer(URL url) {
    // find server.
    String key = url.getAddress();
    //client can export a service which's only for server to invoke
    // 客户端是否可以暴露仅供服务器调用的服务
    boolean isServer = url.getParameter(Constants.IS_SERVER_KEY, true);
    // 如果是的话
    if (isServer) {
        // 获得信息交换服务器
        ExchangeServer server = serverMap.get(key);
        if (server == null) {
            // 重新创建服务器对象，然后放入集合
            serverMap.put(key, createServer(url));
        } else {
            // server supports reset, use together with override
            // 重置
            server.reset(url);
        }
    }
}
```

该方法就是打开服务。其中的逻辑其实是把服务对象放入集合中进行缓存，如果该地址对应的服务器不存在，则调用createServer创建一个服务器对象。

4.createServer

```
private ExchangeServer createServer(URL url) {
    // send readonly event when server closes, it's enabled by default
    // 服务器关闭时发送readonly事件，默认情况下启用
    url = url.addParameterIfAbsent(Constants.CHANNEL_READONLYEVENT_SENT_KEY, Boolean.TRUE.toString());
    // enable heartbeat by default
    // 心跳默认间隔一分钟
    url = url.addParameterIfAbsent(Constants.HEARTBEAT_KEY, String.valueOf(Constants.DEFAULT_HEARTBEAT));
    // 获得远程通讯服务端实现方式，默认用netty3
    String str = url.getParameter(Constants.SERVER_KEY, Constants.DEFAULT_REMOTING_SERVER);

    /**
     * 如果没有该配置，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);

    /**
     * 添加编解码器DubboCodec实现
     */
    url = url.addParameter(Constants.CODEC_KEY, DubboCodec.NAME);
    ExchangeServer server;
    try {
        // 启动服务器
        server = Exchangers.bind(url, requestHandler);
    } catch (RemotingException e) {
```

该方法就是根据url携带的远程通信实现方法来创建一个服务器对象。

5.optimizeSerialization

```
private void optimizeSerialization(URL url) throws RpcException {
    // 获得类名
    String className = url.getParameter(Constants.OPTIMIZER_KEY, "");
    if (StringUtils.isEmpty(className) || optimizers.contains(className)) {
        return;
    }

    logger.info("Optimizing the serialization process for Kryo, FST, etc...");

    try {
        // 加载类
        Class clazz = Thread.currentThread().getContextClassLoader().loadClass(className);
        if (!SerializationOptimizer.class.isAssignableFrom(clazz)) {
            throw new RpcException("The serialization optimizer " + className + " isn't an instance of " +
SerializationOptimizer.class.getName());
        }

        // 强制类型转化为SerializationOptimizer
        SerializationOptimizer optimizer = (SerializationOptimizer) clazz.newInstance();

        if (optimizer.getSerializableClasses() == null) {
            return;
        }

        // 遍历序列化的类，把该类放入到集合进行缓存
        for (Class c : optimizer.getSerializableClasses()) {
```

该方法是把序列化的类放入到集合，以便进行序列化

6.refer

```
@Override
public <T> Invoker<T> refer(Class<T> serviceType, URL url) throws RpcException {
    // 序列化
    optimizeSerialization(url);
    // create rpc invoker. 创建一个DubboInvoker对象
    DubboInvoker<T> invoker = new DubboInvoker<T>(serviceType, url, getClients(url), invokers);
    // 把该invoker放入集合
    invokers.add(invoker);
    return invoker;
}
```

该方法是服务引用，其中就是新建一个DubboInvoker对象后把它放入到集合。

7.getClients

```
private ExchangeClient[] getClients(URL url) {
    // whether to share connection
    // 一个连接是否对于一个服务
    boolean service_share_connect = false;
    // 获得url中欢愉连接共享的配置 默认为0
    int connections = url.getParameter(Constants.CONNECTIONS_KEY, 0);
    // if not configured, connection is shared, otherwise, one connection for one service
    // 如果为0，则是共享类，并且连接数为1
    if (connections == 0) {
        service_share_connect = true;
        connections = 1;
    }

    // 创建数组
    ExchangeClient[] clients = new ExchangeClient[connections];
    for (int i = 0; i < clients.length; i++) {
        // 如果共享，则获得共享客户端对象，否则新建客户端
        if (service_share_connect) {
            clients[i] = getSharedClient(url);
        } else {
            clients[i] = initClient(url);
        }
    }
    return clients;
}
```

该方法是获得客户端集合的方法，分为共享客户端和非共享客户端。共享客户端是共用同一个连接，非共享客户端是每个客户端都有自己的一个连接。

8.getSharedClient

```
private ExchangeClient getSharedClient(URL url) {
    String key = url.getAddress();
    // 从集合中取出客户端对象
    ReferenceCountExchangeClient client = referenceClientMap.get(key);
    // 如果不为空并且没关闭连接，则计数器加1，返回
    if (client != null) {
        if (!client.isClosed()) {
            client.incrementAndGetCount();
            return client;
        } else {
            // 如果连接断开，则从集合中移除
            referenceClientMap.remove(key);
        }
    }

    locks.putIfAbsent(key, new Object());
    synchronized (locks.get(key)) {
        // 如果集合中有该key
        if (referenceClientMap.containsKey(key)) {
            // 则直接返回client
            return referenceClientMap.get(key);
        }

        // 否则新建一个连接
        ExchangeClient exchangeClient = initClient(url);
        client = new ReferenceCountExchangeClient(exchangeClient, ghostClientMap);
```

该方法是获得分享的客户端连接。

9.initClient

```
private ExchangeClient initClient(URL url) {

    // client type setting.
    // 获得客户端的实现方法 默认netty3
    String str = url.getParameter(Constants.CLIENT_KEY, url.getParameter(Constants.SERVER_KEY,
Constants.DEFAULT_REMOTING_CLIENT));

    // 添加编码器
    url = url.addParameter(Constants.CODEC_KEY, DubboCodec.NAME);
    // enable heartbeat by default
    // 默认开启心跳
    url = url.addParameterIfAbsent(Constants.HEARTBEAT_KEY, String.valueOf(Constants.DEFAULT_HEARTBEAT));

    // BIO is not allowed since it has severe performance issue.
    if (str != null && str.length() > 0 &&
!ExtensionLoader.getExtensionLoader(Transporter.class).hasExtension(str)) {
        throw new RpcException("Unsupported client type: " + str + "," +
                " supported client type is " +
StringUtils.join(ExtensionLoader.getExtensionLoader(Transporter.class).getSupportedExtensions(), " "));
    }

    ExchangeClient client;
    try {
        // connection should be lazy
        // 是否需要延迟连接，，默认不开启
        if (url.getParameter(Constants.LAZY_CONNECT_KEY, false)) {
```

该方法是新建一个客户端连接

10.destroy

```
@Override
public void destroy() {
    // 遍历服务器逐个关闭
    for (String key : new ArrayList<String>(serverMap.keySet())) {
        ExchangeServer server = serverMap.remove(key);
        if (server != null) {
            try {
                if (logger.isInfoEnabled()) {
                    logger.info("Close dubbo server: " + server.getLocalAddress());
                }
                server.close(ConfigUtils.getServerShutdownTimeout());
            } catch (Throwable t) {
                logger.warn(t.getMessage(), t);
            }
        }
    }

    // 遍历客户端集合逐个关闭
    for (String key : new ArrayList<String>(referenceClientMap.keySet())) {
        ExchangeClient client = referenceClientMap.remove(key);
        if (client != null) {
            try {
                if (logger.isInfoEnabled()) {
                    logger.info("Close dubbo connect: " + client.getLocalAddress() + "-->" +
client.getRemoteAddress());
                }
```

该方法是销毁的方法重写。

（四）ChannelWrappedInvoker

该类是对当前通道内的客户端调用消息进行包装

1.属性

```
/**
 * 通道
 */
private final Channel channel;
/**
 * 服务key
 */
private final String serviceKey;
/**
 * 当前的客户端
 */
private final ExchangeClient currentClient;
```

2.doInvoke

```
@Override
protected Result doInvoke(Invocation invocation) throws Throwable {
    RpcInvocation inv = (RpcInvocation) invocation;
    // use interface's name as service path to export if it's not found on client side
    // 设置服务path，默认用接口名称
    inv.setAttachment(Constants.PATH_KEY, getInterface().getName());
    // 设置回调的服务key
    inv.setAttachment(Constants.CALLBACK_SERVICE_KEY, serviceKey);

    try {
        // 如果是异步的
        if (getUrl().getMethodParameter(invocation.getMethodName(), Constants.ASYNC_KEY, false)) { // may have
concurrency issue
            // 直接发送请求消息
            currentClient.send(inv, getUrl().getMethodParameter(invocation.getMethodName(), Constants.SENT_KEY,
false));
            return new RpcResult();
        }
        // 获得超时时间
        int timeout = getUrl().getMethodParameter(invocation.getMethodName(), Constants.TIMEOUT_KEY,
Constants.DEFAULT_TIMEOUT);
        if (timeout > 0) {
            return (Result) currentClient.request(inv, timeout).get();
        } else {
            return (Result) currentClient.request(inv).get();
        }
```

该方法是在invoker调用的时候对发送请求消息进行了包装。

3.ChannelWrapper

该类是个内部没，继承了ClientDelegate，其中将编码器变成了dubbo的编码器，其他方法比较简单。

（五）DecodeableRpcInvocation

该类主要做了对于会话域内的数据进行序列化和解码。

1.属性

```
private static final Logger log = LoggerFactory.getLogger(DecodeableRpcInvocation.class);

/**
 * 通道
 */
private Channel channel;

/**
 * 序列化类型
 */
private byte serializationType;

/**
 * 输入流
 */
private InputStream inputStream;

/**
 * 请求
 */
private Request request;

/**
 * 是否解码
 */
private volatile boolean hasDecoded;
```

2.decode

```
@Override
public void decode() throws Exception {
    // 如果没有解码，则进行解码
    if (!hasDecoded && channel != null && inputStream != null) {
        try {
            decode(channel, inputStream);
        } catch (Throwable e) {
            if (log.isWarnEnabled()) {
                log.warn("Decode rpc invocation failed: " + e.getMessage(), e);
            }
            request.setBroken(true);
            request.setData(e);
        } finally {
            // 设置已经解码
            hasDecoded = true;
        }
    }
}

@Override
public Object decode(Channel channel, InputStream input) throws IOException {
    // 对数据进行反序列化
    ObjectInput in = CodecSupport.getSerialization(channel.getUrl(), serializationType)
            .deserialize(channel.getUrl(), input);

    // dubbo版本
```

该方法就是处理Invocation内数据的逻辑，其中主要是做了序列化和解码。把读取出来的设置放入对对应位置传递给后面的调用。

（六）DecodeableRpcResult

该类是做了基于dubbo协议对prc结果的解码

1.属性

```
private static final Logger log = LoggerFactory.getLogger(DecodeableRpcResult.class);

/**
 * 通道
 */
private Channel channel;

/**
 * 序列化类型
 */
private byte serializationType;

/**
 * 输入流
 */
private InputStream inputStream;

/**
 * 响应
 */
private Response response;

/**
 * 会话域
 */
private Invocation invocation;
```

2.decode

```
@Override
public Object decode(Channel channel, InputStream input) throws IOException {
    // 反序列化
    ObjectInput in = CodecSupport.getSerialization(channel.getUrl(), serializationType)
            .deserialize(channel.getUrl(), input);

    byte flag = in.readByte();
    // 根据返回的不同结果来进行处理
    switch (flag) {
        case DubboCodec.RESPONSE_NULL_VALUE:
            // 返回结果为空
            break;
        case DubboCodec.RESPONSE_VALUE:
            //
            try {
                // 获得返回类型数组
                Type[] returnType = RpcUtils.getReturnTypes(invocation);
                // 根据返回类型读取返回结果并且放入RpcResult
                setValue(returnType == null || returnType.length == 0 ? in.readObject() :
                        (returnType.length == 1 ? in.readObject((Class<?>) returnType[0])
                                : in.readObject((Class<?>) returnType[0], returnType[1])));
            } catch (ClassNotFoundException e) {
                throw new IOException(StringUtils.toString("Read response data failed.", e));
            }
            break;
        case DubboCodec.RESPONSE_WITH_EXCEPTION:
```

该方法是对响应结果的解码，其中根据不同的返回结果来对RpcResult设置不同的值。

（七）LazyConnectExchangeClient

该类实现了ExchangeClient接口，是ExchangeClient的装饰器，用到了装饰模式，是延迟连接的客户端实现类。

1.属性

```
// when this warning rises from invocation, program probably have bug.
/**
 * 延迟连接请求错误key
 */
static final String REQUEST_WITH_WARNING_KEY = "lazyclient_request_with_warning";
private final static Logger logger = LoggerFactory.getLogger(LazyConnectExchangeClient.class);
/**
 * 是否在延迟连接请求时错误
 */
protected final boolean requestWithWarning;
/**
 * url对象
 */
private final URL url;
/**
 * 请求处理器
 */
private final ExchangeHandler requestHandler;
/**
 * 连接锁
 */
private final Lock connectLock = new ReentrantLock();
// lazy connect, initial state for connection
/**
 * 初始化状态
 */
```

可以看到有属性ExchangeClient client，该类中很多方法就直接调用了client的方法。

2.构造方法

```
public LazyConnectExchangeClient(URL url, ExchangeHandler requestHandler) {
    // lazy connect, need set send.reconnect = true, to avoid channel bad status.
    // 默认有重连
    this.url = url.addParameter(Constants.SEND_RECONNECT_KEY, Boolean.TRUE.toString());
    this.requestHandler = requestHandler;
    // 默认延迟连接初始化成功
    this.initialState = url.getParameter(Constants.LAZY_CONNECT_INITIAL_STATE_KEY,
Constants.DEFAULT_LAZY_CONNECT_INITIAL_STATE);
    // 默认没有错误
    this.requestWithWarning = url.getParameter(REQUEST_WITH_WARNING_KEY, false);
}
```

3.initClient

```
private void initClient() throws RemotingException {
    // 如果客户端已经初始化，则直接返回
    if (client != null)
        return;
    if (logger.isInfoEnabled()) {
        logger.info("Lazy connect to " + url);
    }
    // 获得连接锁
    connectLock.lock();
    try {
        // 二次判空
        if (client != null)
            return;
        // 新建一个客户端
        this.client = Exchangers.connect(url, requestHandler);
    } finally {
        // 释放锁
        connectLock.unlock();
    }
}
```

该方法是初始化客户端的方法。

4.request

```
@Override
public ResponseFuture request(Object request) throws RemotingException {
    warning(request);
    initClient();
    return client.request(request);
}
```

该方法在调用client.request前调用了前面两个方法，initClient我在上面讲到了，就是用来初始化客户端的。而warning是用来报错的。

5.warning

```
private void warning(Object request) {
    if (requestWithWarning) {
        // 每5000次报错一次
        if (warningcount.get() % 5000 == 0) {
            logger.warn(new IllegalStateException("safe guard client , should not be called ,must have a bug."));
        }
        warningcount.incrementAndGet();
    }
}
```

每5000次记录报错一次。

（八）ReferenceCountExchangeClient

该类也是对ExchangeClient的装饰，其中增强了调用次数多功能。

1.属性

```
/**
 * url对象
 */
private final URL url;
/**
 * 计数
 */
private final AtomicInteger refenceCount = new AtomicInteger(0);

//    private final ExchangeHandler handler;
/**
 * 延迟连接客户端集合
 */
private final ConcurrentMap<String, LazyConnectExchangeClient> ghostClientMap;
/**
 * 客户端对象
 */
private ExchangeClient client;
```

2.replaceWithLazyClient

```
// ghost client
private LazyConnectExchangeClient replaceWithLazyClient() {
    // this is a defensive operation to avoid client is closed by accident, the initial state of the client is false
    // 设置延迟连接初始化状态、是否重连、是否已经重连等配置
    URL lazyUrl = url.addParameter(Constants.LAZY_CONNECT_INITIAL_STATE_KEY, Boolean.FALSE)
            .addParameter(Constants.RECONNECT_KEY, Boolean.FALSE)
            .addParameter(Constants.SEND_RECONNECT_KEY, Boolean.TRUE.toString())
            .addParameter("warning", Boolean.TRUE.toString())
            .addParameter(LazyConnectExchangeClient.REQUEST_WITH_WARNING_KEY, true)
            .addParameter("_client_memo", "referencecounthandler.replacewithlazyclient");

    // 获得服务地址
    String key = url.getAddress();
    // in worst case there's only one ghost connection.
    // 从集合中获取客户端
    LazyConnectExchangeClient gclient = ghostClientMap.get(key);
    // 如果对应等客户端不存在或者已经关闭连接，则重新创建一个延迟连接等客户端，并且放入集合
    if (gclient == null || gclient.isClosed()) {
        gclient = new LazyConnectExchangeClient(lazyUrl, client.getExchangeHandler());
        ghostClientMap.put(key, gclient);
    }
    return gclient;
}
```

该方法是用延迟连接替代，该方法在close方法中被调用。

3.close

```
@Override
public void close(int timeout) {
    if (refenceCount.decrementAndGet() <= 0) {
        if (timeout == 0) {
            client.close();
        } else {
            client.close(timeout);
        }
        client = replaceWithLazyClient();
    }
}
```

（九）FutureAdapter

该类实现了Future接口，是响应的Future适配器。其中是基于ResponseFuture做适配。其中比较简单，我就不多讲解了。

（十）CallbackServiceCodec

该类是针对回调服务的编解码器。

1.属性

```
/**
 * 代理工厂
 */
private static final ProxyFactory proxyFactory =
ExtensionLoader.getExtensionLoader(ProxyFactory.class).getAdaptiveExtension();
/**
 * dubbo协议
 */
private static final DubboProtocol protocol = DubboProtocol.getDubboProtocol();
/**
 * 回调的标志
 */
private static final byte CALLBACK_NONE = 0x0;
/**
 * 回调的创建标志
 */
private static final byte CALLBACK_CREATE = 0x1;
/**
 * 回调的销毁标志
 */
private static final byte CALLBACK_DESTROY = 0x2;
/**
 * 回调参数key
 */
private static final String INV_ATT_CALLBACK_KEY = "sys_callback_arg-";
```

2.encodeInvocationArgument

```
public static Object encodeInvocationArgument(Channel channel, RpcInvocation inv, int paraIndex) throws
IOException {
    // get URL directly
    // 直接获得url
    URL url = inv.getInvoker() == null ? null : inv.getInvoker().getUrl();
    // 设置回调标志
    byte callbackstatus = isCallBack(url, inv.getMethodName(), paraIndex);
    // 获得参数集合
    Object[] args = inv.getArguments();
    // 获得参数类型集合
    Class<?>[] pts = inv.getParameterTypes();
    // 根据不同的回调状态来设置附加值和返回参数
    switch (callbackstatus) {
        case CallbackServiceCodec.CALLBACK_NONE:
            return args[paraIndex];
        case CallbackServiceCodec.CALLBACK_CREATE:
            inv.setAttachment(INV_ATT_CALLBACK_KEY + paraIndex, exportOrunexportCallbackService(channel, url,
pts[paraIndex], args[paraIndex], true));
            return null;
        case CallbackServiceCodec.CALLBACK_DESTROY:
            inv.setAttachment(INV_ATT_CALLBACK_KEY + paraIndex, exportOrunexportCallbackService(channel, url,
pts[paraIndex], args[paraIndex], false));
            return null;
        default:
            return args[paraIndex];
    }
```

该方法是对会话域的信息进行编码。

3.decodeInvocationArgument

```
public static Object decodeInvocationArgument(Channel channel, RpcInvocation inv, Class<?>[] pts, int paraIndex,
Object inObject) throws IOException {
    // if it's a callback, create proxy on client side, callback interface on client side can be invoked through
channel
    // need get URL from channel and env when decode
    URL url = null;
    try {
        // 获得url
        url = DubboProtocol.getDubboProtocol().getInvoker(channel, inv).getUrl();
    } catch (RemotingException e) {
        if (logger.isInfoEnabled()) {
            logger.info(e.getMessage(), e);
        }
        return inObject;
    }
    // 获得回调状态
    byte callbackstatus = isCallBack(url, inv.getMethodName(), paraIndex);
    // 根据回调状态来返回结果
    switch (callbackstatus) {
        case CallbackServiceCodec.CALLBACK_NONE:
            return inObject;
        case CallbackServiceCodec.CALLBACK_CREATE:
            try {
                return referOrdestroyCallbackService(channel, url, pts[paraIndex], inv,
Integer.parseInt(inv.getAttachment(INV_ATT_CALLBACK_KEY + paraIndex)), true);
            } catch (Exception e) {
```

该方法是对会话域内的信息进行解码。

4.isCallBack

```
private static byte isCallBack(URL url, String methodName, int argIndex) {
    // parameter callback rule: method-name.parameter-index(starting from 0).callback
    // 参数的规则：ethod-name.parameter-index(starting from 0).callback
    byte isCallback = CALLBACK_NONE;
    if (url != null) {
        // 获得回调的值
        String callback = url.getParameter(methodName + "." + argIndex + ".callback");
        if (callback != null) {
            // 如果为true，则设置为创建标志
            if (callback.equalsIgnoreCase("true")) {
                isCallback = CALLBACK_CREATE;
                // 如果为false，则设置为销毁标志
            } else if (callback.equalsIgnoreCase("false")) {
                isCallback = CALLBACK_DESTROY;
            }
        }
    }
    return isCallback;
}
```

该方法是根据url携带的参数设置回调的标志，以供执行不同的编解码逻辑。

5.exportOrunexportCallbackService

```
private static String exportOrunexportCallbackService(Channel channel, URL url, Class clazz, Object inst, Boolean
export) throws IOException {
    // 返回对象的hashCode
    int instid = System.identityHashCode(inst);

    Map<String, String> params = new HashMap<String, String>(3);
    // no need to new client again
    // 设置不是服务端标志为否
    params.put(Constants.IS_SERVER_KEY, Boolean.FALSE.toString());
    // mark it's a callback, for troubleshooting
    // 设置是回调服务标志为true
    params.put(Constants.IS_CALLBACK_SERVICE, Boolean.TRUE.toString());
    String group = url.getParameter(Constants.GROUP_KEY);
    if (group != null && group.length() > 0) {
        // 设置是消费侧还是提供侧
        params.put(Constants.GROUP_KEY, group);
    }
    // add method, for verifying against method, automatic fallback (see dubbo protocol)
    // 添加方法，在dubbo的协议里面用到
    params.put(Constants.METHODS_KEY, StringUtils.join(Wrapper.getWrapper(clazz).getDeclaredMethodNames(), ","));

    Map<String, String> tmpmap = new HashMap<String, String>(url.getParameters());
    tmpmap.putAll(params);
    // 移除版本信息
    tmpmap.remove(Constants.VERSION_KEY);// doesn't need to distinguish version for callback
    // 设置接口名
```

该方法是在客户端侧暴露服务和取消暴露服务。

6.referOrdestroyCallbackService

```
private static Object referOrdestroyCallbackService(Channel channel, URL url, Class<?> clazz, Invocation inv, int
instid, boolean isRefer) {
    Object proxy = null;
    // 获得服务调用的缓存key
    String invokerCacheKey = getServerSideCallbackInvokerCacheKey(channel, clazz.getName(), instid);
    // 获得代理缓存key
    String proxyCacheKey = getServerSideCallbackServiceCacheKey(channel, clazz.getName(), instid);
    // 从通道内获得代理对象
    proxy = channel.getAttribute(proxyCacheKey);
    // 获得计数器key
    String countkey = getServerSideCountKey(channel, clazz.getName());
    // 如果是服务引用
    if (isRefer) {
        // 如果代理对象为空
        if (proxy == null) {
            // 获得服务引用的url
            URL referurl = URL.valueOf("callback://" + url.getAddress() + "/" + clazz.getName() + "?" +
Constants.INTERFACE_KEY + "=" + clazz.getName());
            referurl =
referurl.addParametersIfAbsent(url.getParameters()).removeParameter(Constants.METHODS_KEY);
            if (!isInstancesOverLimit(channel, referurl, clazz.getName(), instid, true)) {
                @SuppressWarnings("rawtypes")
                Invoker<?> invoker = new ChannelWrappedInvoker(clazz, channel, referurl, String.valueOf(instid));
                // 获得代理类
                proxy = proxyFactory.getProxy(invoker);
                // 设置代理类
```

该方法是在服务端侧进行服务引用或者销毁回调服务。

（十一）DubboCodec

该类是dubbo的编解码器，分别针对dubbo协议的request和response进行编码和解码。

1.属性

```
/**
 * dubbo名称
 */
public static final String NAME = "dubbo";
/**
 * 协议版本号
 */
public static final String DUBBO_VERSION = Version.getProtocolVersion();
/**
 * 响应携带着异常
 */
public static final byte RESPONSE_WITH_EXCEPTION = 0;
/**
 * 响应
 */
public static final byte RESPONSE_VALUE = 1;
/**
 * 响应结果为空
 */
public static final byte RESPONSE_NULL_VALUE = 2;
/**
 * 响应结果有异常并且带有附加值
 */
public static final byte RESPONSE_WITH_EXCEPTION_WITH_ATTACHMENTS = 3;
/**
 * 响应结果有附加值
```

2.decodeBody

```
@Override
protected Object decodeBody(Channel channel, InputStream is, byte[] header) throws IOException {
    byte flag = header[2], proto = (byte) (flag & SERIALIZATION_MASK);
    // get request id.
    long id = Bytes.bytes2long(header, 4);
    // 如果是response
    if ((flag & FLAG_REQUEST) == 0) {
        // decode response.
        // 创建一个response
        Response res = new Response(id);
        // 如果是事件，则设置事件，这里有个问题，我提交了pr在新版本已经修复
        if ((flag & FLAG_EVENT) != 0) {
            res.setEvent(Response.HEARTBEAT_EVENT);
        }
        // get status.
        // 设置状态
        byte status = header[3];
        res.setStatus(status);
        try {
            // 反序列化
            ObjectInput in = CodecSupport.deserialize(channel.getUrl(), is, proto);
            // 如果状态是响应成功
            if (status == Response.OK) {
                Object data;
                // 如果是心跳事件，则按照心跳事件解码
                if (res.isHeartbeat()) {
```

该方法是对request和response进行解码，用位运算来进行解码，其中的逻辑跟我在[《dubbo源码解析（十）远程通信——Exchange层》](#)中讲到的编解码器逻辑差不多。

3.encodeRequestData

```
@Override
protected void encodeRequestData(Channel channel, ObjectOutput out, Object data, String version) throws IOException {
    RpcInvocation inv = (RpcInvocation) data;

    // 输出版本
    out.writeUTF(version);
    // 输出path
    out.writeUTF(inv.getAttachment(Constants.PATH_KEY));
    // 输出版本号
    out.writeUTF(inv.getAttachment(Constants.VERSION_KEY));

    // 输出方法名称
    out.writeUTF(inv.getMethodName());
    // 输出参数类型
    out.writeUTF(ReflectUtils.getDesc(inv.getParameterTypes()));
    // 输出参数
    Object[] args = inv.getArguments();
    if (args != null)
        for (int i = 0; i < args.length; i++) {
            out.writeObject(encodeInvocationArgument(channel, inv, i));
        }
    // 输出附加值
    out.writeObject(inv.getAttachments());
}
```

该方法是对请求数据的编码。

4.encodeResponseData

```
@Override
protected void encodeResponseData(Channel channel, ObjectOutput out, Object data, String version) throws
IOException {
    Result result = (Result) data;
    // currently, the version value in Response records the version of Request
    boolean attach = Version.isSupportResponseAttatchment(version);
    // 获得异常
    Throwable th = result.getException();
    if (th == null) {
        Object ret = result.getValue();
        // 根据结果的不同输出不同的值
        if (ret == null) {
            out.writeByte(attach ? RESPONSE_NULL_VALUE_WITH_ATTACHMENTS : RESPONSE_NULL_VALUE);
        } else {
            out.writeByte(attach ? RESPONSE_VALUE_WITH_ATTACHMENTS : RESPONSE_VALUE);
            out.writeObject(ret);
        }
    } else {
        // 如果有异常，则输出异常
        out.writeByte(attach ? RESPONSE_WITH_EXCEPTION_WITH_ATTACHMENTS : RESPONSE_WITH_EXCEPTION);
        out.writeObject(th);
    }

    if (attach) {
        // returns current version of Response to consumer side.
        // 在附加值中加入版本号
```

该方法是对响应数据的编码。

（十二）DubboCountCodec

该类是对DubboCodec的功能增强，增加了消息长度的限制。

```
public final class DubboCountCodec implements Codec2 {

    private DubboCodec codec = new DubboCodec();

    @Override
    public void encode(Channel channel, ChannelBuffer buffer, Object msg) throws IOException {
        codec.encode(channel, buffer, msg);
    }

    @Override
    public Object decode(Channel channel, ChannelBuffer buffer) throws IOException {
        // 保存读取的标志
        int save = buffer.readerIndex();
        MultiMessage result = MultiMessage.create();
        do {
            Object obj = codec.decode(channel, buffer);
            // 粘包拆包
            if (Codec2.DecodeResult.NEED_MORE_INPUT == obj) {
                buffer.readerIndex(save);
                break;
            } else {
                // 增加消息
                result.addMessage(obj);
                // 记录消息长度
                logMessageLength(obj, buffer.readerIndex() - save);
                save = buffer.readerIndex();
```

（十三）TraceFilter

该过滤器是增强的功能是通道的跟踪，会在通道内把最大的调用次数和现在的调用数量放进去。方便使用telnet来跟踪服务的调用次数等。

1.属性

```
/**
 * 跟踪数量的最大值key
 */
private static final String TRACE_MAX = "trace.max";

/**
 * 跟踪的数量
 */
private static final String TRACE_COUNT = "trace.count";

/**
 * 通道集合
 */
private static final ConcurrentMap<String, Set<Channel>> tracers = new ConcurrentHashMap<String, Set<Channel>>();
```

2.addTracer

```
public static void addTracer(Class<?> type, String method, Channel channel, int max) {
    // 设置最大的数量
    channel.setAttribute(TRACE_MAX, max);
    // 设置当前的数量
    channel.setAttribute(TRACE_COUNT, new AtomicInteger());
    // 获得key
    String key = method != null && method.length() > 0 ? type.getName() + "." + method : type.getName();
    // 获得通道集合
    Set<Channel> channels = tracers.get(key);
    // 如果为空，则新建
    if (channels == null) {
        tracers.putIfAbsent(key, new ConcurrentHashSet<Channel>());
        channels = tracers.get(key);
    }
    channels.add(channel);
}
```

该方法是对某一个通道进行跟踪，把现在的调用数量放到属性里面

3.removeTracer

```
public static void removeTracer(Class<?> type, String method, Channel channel) {
    // 移除最大值属性
    channel.removeAttribute(TRACE_MAX);
    // 移除数量属性
    channel.removeAttribute(TRACE_COUNT);
    String key = method != null && method.length() > 0 ? type.getName() + "." + method : type.getName();
    Set<Channel> channels = tracers.get(key);
    if (channels != null) {
        // 集合中移除该通道
        channels.remove(channel);
    }
}
```

该方法是移除通道的跟踪。

4.invoke

```
@Override
public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
    // 开始时间
    long start = System.currentTimeMillis();
    // 调用下一个调用链 获得结果
    Result result = invoker.invoke(invocation);
    // 调用结束时间
    long end = System.currentTimeMillis();
    // 如果通道跟踪大小大于0
    if (tracers.size() > 0) {
        // 服务key
        String key = invoker.getInterface().getName() + "." + invocation.getMethodName();
        // 获得通道集合
        Set<Channel> channels = tracers.get(key);
        if (channels == null || channels.isEmpty()) {
            key = invoker.getInterface().getName();
            channels = tracers.get(key);
        }
        if (channels != null && !channels.isEmpty()) {
            // 遍历通道集合
            for (Channel channel : new ArrayList<Channel>(channels)) {
                // 如果通道是连接的
                if (channel.isConnected()) {
                    try {
                        // 获得跟踪的最大数
                        int max = 1;
```

该方法是当服务被调用时，进行跟踪或者取消跟踪的处理逻辑，是核心的功能增强逻辑。

（十四）FutureFilter

该类是处理异步和同步调用结果的过滤器。

1.invoke

```
@Override
public Result invoke(final Invoker<?> invoker, final Invocation invocation) throws RpcException {
    // 是否是异步的调用
    final boolean isAsync = RpcUtils.isAsync(invoker.getUrl(), invocation);

    fireInvokeCallback(invoker, invocation);
    // need to configure if there's return value before the invocation in order to help invoker to judge if it's
    // necessary to return future.
    Result result = invoker.invoke(invocation);
    if (isAsync) {
        // 调用异步处理
        asyncCallback(invoker, invocation);
    } else {
        // 调用同步结果处理
        syncCallback(invoker, invocation, result);
    }
    return result;
}
```

该方法中根据是否为异步调用来分别执行asyncCallback和syncCallback方法。

2.syncCallback

```
private void syncCallback(final Invoker<?> invoker, final Invocation invocation, final Result result) {
    // 如果有异常
    if (result.hasException()) {
        // 则调用异常的结果处理
        fireThrowCallback(invoker, invocation, result.getException());
    } else {
        // 调用正常的结果处理
        fireReturnCallback(invoker, invocation, result.getValue());
    }
}
```

该方法是同步调用的返回结果处理，比较简单。

3.asyncCallback

```
private void asyncCallback(final Invoker<?> invoker, final Invocation invocation) {
    Future<?> f = RpcContext.getContext().getFuture();
    if (f instanceof FutureAdapter) {
        ResponseFuture future = ((FutureAdapter<?>) f).getFuture();
        // 设置回调
        future.setCallback(new ResponseCallback() {
            @Override
            public void done(Object rpcResult) {
                // 如果结果为空，则打印错误日志
                if (rpcResult == null) {
                    logger.error(new IllegalStateException("invalid result value : null, expected " +
Result.class.getName()));
                    return;
                }
                ///must be rpcResult
                // 如果不是Result则打印错误日志
                if (!(rpcResult instanceof Result)) {
                    logger.error(new IllegalStateException("invalid result type :" + rpcResult.getClass() + ",
expected " + Result.class.getName()));
                    return;
                }
                Result result = (Result) rpcResult;
                if (result.hasException()) {
                    // 如果有异常，则调用异常处理方法
                    fireThrowCallback(invoker, invocation, result.getException());
                } else {
```

该方法是异步调用的结果处理，把异步返回结果的逻辑写在回调函数里面。

4.fireInvokeCallback

```
private void fireInvokeCallback(final Invoker<?> invoker, final Invocation invocation) {
    // 获得调用的方法
    final Method onInvokeMethod = (Method)
StaticContext.getSystemContext().get(StaticContext.getKey(invoker.getUrl(), invocation.getMethodName(),
Constants.ON_INVOKE_METHOD_KEY));
    // 获得调用的服务
    final Object onInvokeInst = StaticContext.getSystemContext().get(StaticContext.getKey(invoker.getUrl(),
invocation.getMethodName(), Constants.ON_INVOKE_INSTANCE_KEY));

    if (onInvokeMethod == null && onInvokeInst == null) {
        return;
    }
    if (onInvokeMethod == null || onInvokeInst == null) {
        throw new IllegalStateException("service:" + invoker.getUrl().getServiceKey() + " has a onreturn callback
config , but no such " + (onInvokeMethod == null ? "method" : "instance") + " found. url:" + invoker.getUrl());
    }
    // 如果不可以访问，则设置为可访问
    if (!onInvokeMethod.isAccessible()) {
        onInvokeMethod.setAccessible(true);
    }

    // 获得参数数组
    Object[] params = invocation.getArguments();
    try {
        // 调用方法
        onInvokeMethod.invoke(onInvokeInst, params);
```

该方法是调用方法的执行。

5.fireReturnCallback

```
private void fireReturnCallback(final Invoker<?> invoker, final Invocation invocation, final Object result) {
    final Method onReturnMethod = (Method)
StaticContext.getSystemContext().get(StaticContext.getKey(invoker.getUrl(), invocation.getMethodName(),
Constants.ON_RETURN_METHOD_KEY));
    final Object onReturnInst = StaticContext.getSystemContext().get(StaticContext.getKey(invoker.getUrl(),
invocation.getMethodName(), Constants.ON_RETURN_INSTANCE_KEY));

    //not set onreturn callback
    if (onReturnMethod == null && onReturnInst == null) {
        return;
    }

    if (onReturnMethod == null || onReturnInst == null) {
        throw new IllegalStateException("service:" + invoker.getUrl().getServiceKey() + " has a onreturn callback
config , but no such " + (onReturnMethod == null ? "method" : "instance") + " found. url:" + invoker.getUrl());
    }
    if (!onReturnMethod.isAccessible()) {
        onReturnMethod.setAccessible(true);
    }

    Object[] args = invocation.getArguments();
    Object[] params;
    // 获得返回结果类型
    Class<?>[] rParaTypes = onReturnMethod.getParameterTypes();
    // 设置参数和返回结果
    if (rParaTypes.length > 1) {
```

该方法是正常的返回结果的处理。

6.fireThrowCallback

```
private void fireThrowCallback(final Invoker<?> invoker, final Invocation invocation, final Throwable exception)
{
    final Method onthrowMethod = (Method)
StaticContext.getSystemContext().get(StaticContext.getKey(invoker.getUrl(), invocation.getMethodName(),
Constants.ON_THROW_METHOD_KEY));
    final Object onthrowInst = StaticContext.getSystemContext().get(StaticContext.getKey(invoker.getUrl(),
invocation.getMethodName(), Constants.ON_THROW_INSTANCE_KEY));

    //onthrow callback not configured
    if (onthrowMethod == null && onthrowInst == null) {
        return;
    }
    if (onthrowMethod == null || onthrowInst == null) {
        throw new IllegalStateException("service:" + invoker.getUrl().getServiceKey() + " has a onthrow callback
config , but no such " + (onthrowMethod == null ? "method" : "instance") + " found. url:" + invoker.getUrl());
    }
    if (!onthrowMethod.isAccessible()) {
        onthrowMethod.setAccessible(true);
    }
    // 获得抛出异常的类型
    Class<?>[] rParaTypes = onthrowMethod.getParameterTypes();
    if (rParaTypes[0].isAssignableFrom(exception.getClass())) {
        try {
            Object[] args = invocation.getArguments();
            Object[] params;
```

该方法是异常抛出时的结果处理。

（十五）ServerStatusChecker

该类是对于服务状态的监控设置。

```
public class ServerStatusChecker implements StatusChecker {

    @Override
    public Status check() {
        // 获得服务集合
        Collection<ExchangeServer> servers = DubboProtocol.getDubboProtocol().getServers();
        // 如果为空则返回UNKNOWN的状态
        if (servers == null || servers.isEmpty()) {
            return new Status(Status.Level.UNKNOWN);
        }
        // 设置状态为ok
        Status.Level level = Status.Level.OK;
        StringBuilder buf = new StringBuilder();
        // 遍历集合
        for (ExchangeServer server : servers) {
            // 如果服务没有绑定到本地端口
            if (!server.isBound()) {
                // 状态改为error
                level = Status.Level.ERROR;
                // 加入服务本地地址
                buf.setLength(0);
                buf.append(server.getLocalAddress());
                break;
            }
            if (buf.length() > 0) {
                buf.append(",");
```

（十六）ThreadPoolStatusChecker

该类是对于线程池的状态进行监控。

```java
@Activate
public class ThreadPoolStatusChecker implements StatusChecker {

    @Override
    public Status check() {
        // 获得数据中心
        DataStore dataStore = ExtensionLoader.getExtensionLoader(DataStore.class).getDefaultExtension();
        // 获得线程池集合
        Map<String, Object> executors = dataStore.get(Constants.EXECUTOR_SERVICE_COMPONENT_KEY);

        StringBuilder msg = new StringBuilder();
        // 设置为ok
        Status.Level level = Status.Level.OK;
        // 遍历线程池集合
        for (Map.Entry<String, Object> entry : executors.entrySet()) {
            String port = entry.getKey();
            ExecutorService executor = (ExecutorService) entry.getValue();

            if (executor != null && executor instanceof ThreadPoolExecutor) {
                ThreadPoolExecutor tp = (ThreadPoolExecutor) executor;
                boolean ok = tp.getActiveCount() < tp.getMaximumPoolSize() - 1;
                Status.Level lvl = Status.Level.OK;
                // 如果活跃数量超过了最大的线程数量，则设置warn
                if (!ok) {
                    level = Status.Level.WARN;
                    lvl = Status.Level.WARN;
```

逻辑比较简单，我就不赘述了。

关于telnet下的相关实现请感兴趣的朋友直接查看，里面都是对于telnet命令的实现，内容比较独立。

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dubbo协议的部分，dubbo协议是官方推荐使用的协议，并且对于telnet命令也做了很好的支持，要看懂这部分的逻辑，必须先对于之前的一些接口设计了解的很清楚。接下来我将开始对rpc模块关于hessian协议部分进行讲解。

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

dubbo负载均衡策略及对应源码分析