crazyhzm 265

Dubbo源码解析（九）远程通信——Transport层

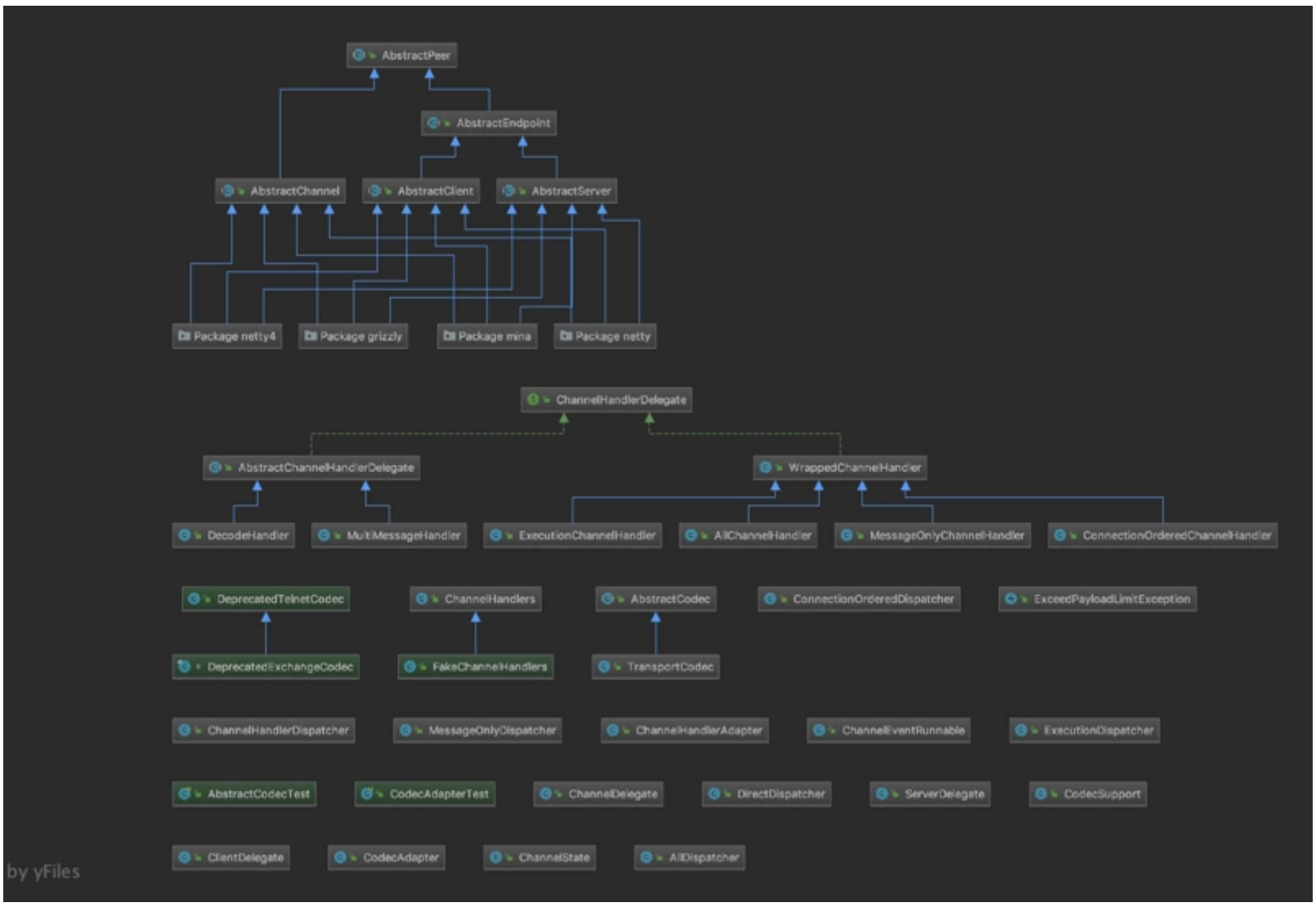
java dubbo 阅读约 77 分钟

远程通讯——Transport层

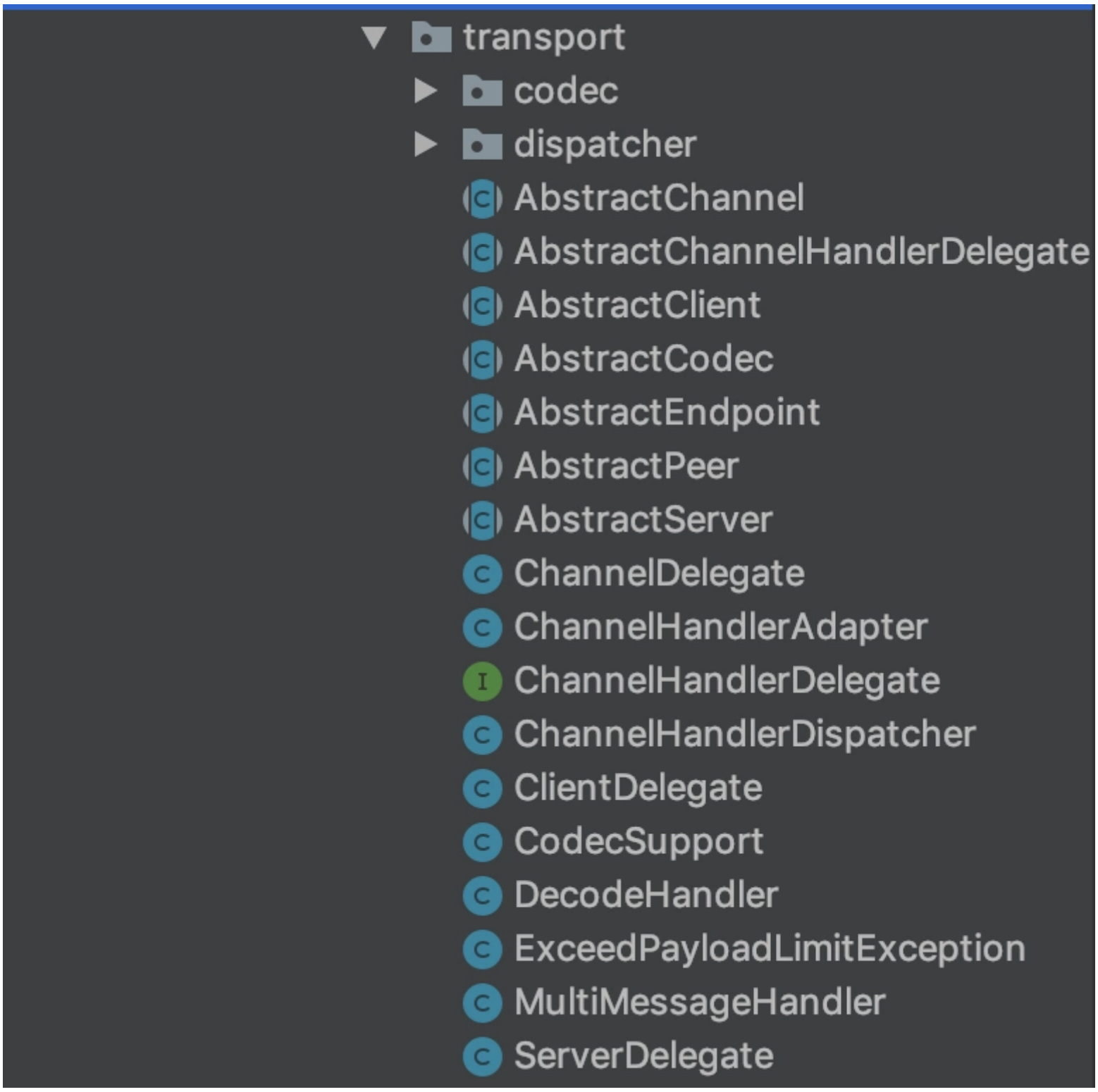
> 目标：介绍Transport层的相关设计和逻辑、介绍dubbo-remoting-api中的transport包内的源码解析。

前言

先预警一下，该文篇幅会很长，做好心理准备。Transport层也就是网络传输层，在远程通信中必然会涉及到传输。它在dubbo 的框架设计中也处于倒数第二层，当然最底层是序列化，这个后面介绍。官方文档对Transport层的解释是抽象 mina 和 netty 为统一接口，以 Message 为中心，扩展接口为 Channel、Transporter、Client、Server、Codec。那我们现在先来看这个包下面的类图：



可以看到有四个包继承了AbstractChannel、AbstractServer、AbstractClient。也就是说现在Transport层是抽象mina、netty以及grizzly为统一接口。看完类图，再来看看包结构：



下面的讲解大致会按照类图中类的顺序往下讲，尽量把client、server、channel、codec、dispacher五部分涉及到的内容一起讲解。

源码解析

（一）AbstractPeer

```
public abstract class AbstractPeer implements Endpoint, ChannelHandler {
    private final ChannelHandler handler;

    private volatile URL url;
    /**
     * 是否正在关闭
     */
    // closing closed means the process is being closed and close is finished
    private volatile boolean closing;
    /**
     * 是否关闭完成
     */
    private volatile boolean closed;

    public AbstractPeer(URL url, ChannelHandler handler) {
        if (url == null) {
            throw new IllegalArgumentException("url == null");
        }
        if (handler == null) {
            throw new IllegalArgumentException("handler == null");
        }
        this.url = url;
        this.handler = handler;
    }
}
```

该类实现了Endpoint和ChannelHandler两个接口，要关注的两个点：

1. 实现ChannelHandler接口并且有在属性中还有一个handler，下面很多实现方法也是直接调用了handler方法，这种模式叫做装饰模式，这样做可以对装饰对象灵活的增强功能。对装饰模式不懂的朋友可以google一下。有很多例子介绍。
2. 在该类中有closing和closed属性，在Endpoint中有很多关于关闭通道的操作，会有关闭中和关闭完成的状态区分，在该类中就缓存了这两个属性来判断关闭的状态。

下面我就介绍该类中的send方法，其他方法比较好理解，到时候可以直接看源码：

```
@Override
public void send(Object message) throws RemotingException {
    // url中sent的配置项
    send(message, url.getParameter(Constants.SENT_KEY, false));
}
```

该配置项是选择是否等待消息发出：

1. sent值为true，等待消息发出，消息发送失败将抛出异常。
2. sent值为false，不等待消息发出，将消息放入 IO 队列，即刻返回。

对该类还有点糊涂的朋友，记住在ChannelHandler接口，该类就做了装饰模式中装饰角色，在Endpoint接口，只是维护了通道的正在关闭和关闭完成两个状态。

（二）AbstractEndpoint

```
public abstract class AbstractEndpoint extends AbstractPeer implements Resetable {

    /**
     * 日志记录
     */
    private static final Logger logger = LoggerFactory.getLogger(AbstractEndpoint.class);

    /**
     * 编解码器
     */
    private Codec2 codec;

    /**
     * 超时时间
     */
    private int timeout;

    /**
     * 连接超时时间
     */
    private int connectTimeout;

    public AbstractEndpoint(URL url, ChannelHandler handler) {
        super(url, handler);
        this.codec = getChannelCodec(url);
        // 优先从url配置中取，如果没有，默认为1s
```

该类是端点的抽象类，其中封装了编解码器以及两个超时时间。基于dubbo 的SPI机制，获得相应的编解码器实现对象，编解码器优先从Codec2的扩展类中寻找。

下面来看看该类中的reset方法：

```
@Override
public void reset(URL url) {
    if (isClosed()) {
        throw new IllegalStateException("Failed to reset parameters "
                + url + ", cause: Channel closed. channel: " + getLocalAddress());
    }
    try {
        // 判断重置的url中有没有携带timeout，有的话重置
        if (url.hasParameter(Constants.TIMEOUT_KEY)) {
            int t = url.getParameter(Constants.TIMEOUT_KEY, 0);
            if (t > 0) {
                this.timeout = t;
            }
        }
    } catch (Throwable t) {
        logger.error(t.getMessage(), t);
    }
    try {
        // 判断重置的url中有没有携带connect.timeout，有的话重置
        if (url.hasParameter(Constants.CONNECT_TIMEOUT_KEY)) {
            int t = url.getParameter(Constants.CONNECT_TIMEOUT_KEY, 0);
            if (t > 0) {
                this.connectTimeout = t;
            }
        }
    } catch (Throwable t) {
```

这个方法是Resetable接口中的方法，可以看到以前的reset实现方法都加上了@Deprecated注解，不推荐使用了，因为这种实现方式重置太复杂，需要把所有参数都设置一遍，比如我只想重置一个超时时间，但是其他值不变，如果用以前的reset，我需要在url中把所有值都带上，就会很多余。现在用新的reset，每次只关心我需要重置的值，只更改为需要重置的值。比如上面的代码所示，只想修改超时时间，那我就只在url中携带超时时间的参数。

（三）AbstractServer

该类继承了AbstractEndpoint并且实现Server接口，是服务器抽象类。重点实现了服务器的公共逻辑，比如发送消息，关闭通道，连接通道，断开连接等。并且抽象了打开和关闭服务器两个方法。

1.属性

```
/**
 * 服务器线程名称
 */
protected static final String SERVER_THREAD_POOL_NAME = "DubboServerHandler";
private static final Logger logger = LoggerFactory.getLogger(AbstractServer.class);
/**
 * 线程池
 */
ExecutorService executor;
/**
 * 服务地址，也就是本地地址
 */
private InetSocketAddress localAddress;
/**
 * 绑定地址
 */
private InetSocketAddress bindAddress;
/**
 * 最大可接受的连接数
 */
private int accepts;
/**
 * 空闲超时时间，单位是s
 */
private int idleTimeout = 600; //600 seconds
```

该类的属性比较好理解，就是稍微注意一下idleTimeout的单位是s。

2.构造函数

```
public AbstractServer(URL url, ChannelHandler handler) throws RemotingException {
    super(url, handler);
    // 从url中获得本地地址
    localAddress = getUrl().toInetSocketAddress();

    // 从url配置中获得绑定的ip
    String bindIp = getUrl().getParameter(Constants.BIND_IP_KEY, getUrl().getHost());
    // 从url配置中获得绑定的端口号
    int bindPort = getUrl().getParameter(Constants.BIND_PORT_KEY, getUrl().getPort());
    // 判断url中配置anyhost是否为true或者判断host是否为不可用的本地Host
    if (url.getParameter(Constants.ANYHOST_KEY, false) || NetUtils.isInvalidLocalHost(bindIp)) {
        bindIp = NetUtils.ANYHOST;
    }
    bindAddress = new InetSocketAddress(bindIp, bindPort);
    // 从url中获取配置，默认值为0
    this.accepts = url.getParameter(Constants.ACCEPTS_KEY, Constants.DEFAULT_ACCEPTS);
    // 从url中获取配置，默认600s
    this.idleTimeout = url.getParameter(Constants.IDLE_TIMEOUT_KEY, Constants.DEFAULT_IDLE_TIMEOUT);
    try {
        // 开启服务器
        doOpen();
        if (logger.isInfoEnabled()) {
            logger.info("Start " + getClass().getSimpleName() + " bind " + getBindAddress() + ", export " +
getLocalAddress());
        }
    } catch (Throwable t) {
```

构造函数大部分逻辑就是从url中取配置，存到缓存中，并且做了开启服务器的操作。具体的看上面的注释，还是比较清晰的。

3.reset方法

```
@Override
public void reset(URL url) {
    if (url == null) {
        return;
    }
    try {
        // 重置accepts的值
        if (url.hasParameter(Constants.ACCEPTS_KEY)) {
            int a = url.getParameter(Constants.ACCEPTS_KEY, 0);
            if (a > 0) {
                this.accepts = a;
            }
        }
    } catch (Throwable t) {
        logger.error(t.getMessage(), t);
    }
    try {
        // 重置idle.timeout的值
        if (url.hasParameter(Constants.IDLE_TIMEOUT_KEY)) {
            int t = url.getParameter(Constants.IDLE_TIMEOUT_KEY, 0);
            if (t > 0) {
                this.idleTimeout = t;
            }
        }
    } catch (Throwable t) {
        logger.error(t.getMessage(), t);
```

该类中的reset方法做了三个值的重置，分别是最大可连接的客户端数量、空闲超时时间以及线程池的两个配置参数。其中要注意核心线程数和最大线程数的区别。举个例子，核心线程数就像是工厂正式工，最大线程数，就是工厂临时工作量加大，请了一批临时工，临时工加正式工的和就是最大线程数，等这批任务结束后，临时工要辞退的，而正式工会留下。

还有send、close、connected、disconnected等方法比较简单，如果有兴趣，可以到我的GitHub查看，地址文章末尾会给出。

（四）AbstractClient

该类是客户端的抽象类，继承了AbstractEndpoint类，实现了Client接口，该类中也是做了客户端公用的重连逻辑，抽象了打开客户端、关闭客户端、连接服务器、断开服务器连接以及获得通道方法，让子类去重点关注这几个方法。

1.属性

```
/**
 * 客户端线程名称
 */
protected static final String CLIENT_THREAD_POOL_NAME = "DubboClientHandler";
private static final Logger logger = LoggerFactory.getLogger(AbstractClient.class);
/**
 * 线程池id
 */
private static final AtomicInteger CLIENT_THREAD_POOL_ID = new AtomicInteger();
/**
 * 重连定时任务执行器
 */
private static final ScheduledThreadPoolExecutor reconnectExecutorService = new ScheduledThreadPoolExecutor(2,
new NamedThreadFactory("DubboClientReconnectTimer", true));
/**
 * 连接锁
 */
private final Lock connectLock = new ReentrantLock();
/**
 * 发送消息时，若断开，是否重连
 */
private final boolean send_reconnect;
/**
 * 重连次数
 */
private final AtomicInteger reconnect_count = new AtomicInteger(0);
```

上述属性大部分跟重连有关，该类最重要的也是封装了重连的逻辑。

2.构造函数

```
public AbstractClient(URL url, ChannelHandler handler) throws RemotingException {
    super(url, handler);

    // 从url中获得是否重连的配置，默认为false
    send_reconnect = url.getParameter(Constants.SEND_RECONNECT_KEY, false);

    // 从url中获得关闭超时时间，默认为900s
    shutdown_timeout = url.getParameter(Constants.SHUTDOWN_TIMEOUT_KEY, Constants.DEFAULT_SHUTDOWN_TIMEOUT);

    // The default reconnection interval is 2s, 1800 means warning interval is 1 hour.
    // 重连的默认值是2s，重连 warning 的间隔默认是1800，当出错的时候，每隔1800*2=3600s报警一次
    reconnect_warning_period = url.getParameter("reconnect.waring.period", 1800);

    try {
        // 打开客户端
        doOpen();
    } catch (Throwable t) {
        close();
        throw new RemotingException(url.toInetSocketAddress(), null,
                "Failed to start " + getClass().getSimpleName() + " " + NetUtils.getLocalAddress()
                        + " connect to the server " + getRemoteAddress() + ", cause: " + t.getMessage(), t);
    }
    try {
        // connect.
        // 连接服务器
        connect();
```

该构造函数中做了一些属性值的设置，并且做了打开客户端和连接服务器的操作。

3.wrapChannelHandler

```
protected static ChannelHandler wrapChannelHandler(URL url, ChannelHandler handler) {
    // 加入线程名称
    url = ExecutorUtil.setThreadName(url, CLIENT_THREAD_POOL_NAME);
    // 设置使用的线程池类型
    url = url.addParameterIfAbsent(Constants.THREADPOOL_KEY, Constants.DEFAULT_CLIENT_THREADPOOL);
    // 包装
    return ChannelHandlers.wrap(handler, url);
}
```

该方法是包装通道处理器，设置使用的线程池类型是可缓存线程池。

4.initConnectStatusCheckCommand

```
private synchronized void initConnectStatusCheckCommand() {
    //reconnect=false to close reconnect
    int reconnect = getReconnectParam(getUrl());
    // 有连接频率的值，并且当前没有连接任务
    if (reconnect > 0 && (reconnectExecutorFuture == null || reconnectExecutorFuture.isCancelled())) {
        Runnable connectStatusCheckCommand = new Runnable() {
            @Override
            public void run() {
                try {
                    if (!isConnected()) {
                        // 重连
                        connect();
                    } else {
                        // 记录最后一次重连的时间
                        lastConnectedTime = System.currentTimeMillis();
                    }
                } catch (Throwable t) {
                    String errorMsg = "client reconnect to " + getUrl().getAddress() + " find error . url: " +
getUrl();
                    // wait registry sync provider list
                    if (System.currentTimeMillis() - lastConnectedTime > shutdown_timeout) {
                        // 如果之前没有打印过重连的误日志
                        if (!reconnect_error_log_flag.get()) {
                            reconnect_error_log_flag.set(true);
                            // 打印日志
                            logger.error(errorMsg, t);
```

该方法是初始化重连线程，其中做了重连失败后的告警日志和错误日志打印策略。

5.reconnect

```
@Override
public void reconnect() throws RemotingException {
    disconnect();
    connect();
}
```

单独放该方法是因为这是该类关注的重点。实现了客户端的重连逻辑。

6.其他

connect、disconnect、close等方法都是调用了对应的抽象方法，而具体的逻辑需要看具体的子类如何去实现相关的抽象方法，这几个方法逻辑比较简单，我不在这里贴出源码，有兴趣可以看我的GitHub，地址文章末尾会给出。

（四）AbstractChannel

该类是通道的抽象类，该类里面做的逻辑很简单，具体的发送消息逻辑在它 的子类中实现。

```
@Override
public void send(Object message, boolean sent) throws RemotingException {
    // 检测通道是否关闭
    if (isClosed()) {
        throw new RemotingException(this, "Failed to send message "
                + (message == null ? "" : message.getClass().getName()) + ":" + message
                + ", cause: Channel closed. channel: " + getLocalAddress() + " -> " + getRemoteAddress());
    }
}
```

可以看到send方法，其中只做了检测通道是否关闭的状态检测，没有实现具体的发送消息的逻辑。

（五）ChannelHandlerDelegate

该类继承了ChannelHandler，从它的名字可以看出是ChannelHandler的代表，它就是作为装饰模式中的Component角色，后面讲到的AbstractChannelHandlerDelegate作为装饰模式中的Decorator角色。

```
public interface ChannelHandlerDelegate extends ChannelHandler {
    /**
     * 获得通道
     * @return
     */
    ChannelHandler getHandler();
}
```

（六）AbstractChannelHandlerDelegate

属性：

```
protected ChannelHandler handler
```

该类实现了ChannelHandlerDelegate接口，并且有一个属性是ChannelHandler，上述已经说到这是装饰模式中的装饰角色，其中的所有实现方法都直接调用被装饰的handler属性的方法。

（七）DecodeHandler

该类为解码处理器，继承了AbstractChannelHandlerDelegate，对接收到的消息进行解码，在父类处理接收消息的功能上叠加了解码功能。

我们来看看received方法：

```
@Override
public void received(Channel channel, Object message) throws RemotingException {
    // 如果是Decodeable类型的消息，则对整个消息解码
    if (message instanceof Decodeable) {
        decode(message);
    }

    // 如果是Request请求类型消息，则对请求中对请求数据解码
    if (message instanceof Request) {
        decode(((Request) message).getData());
    }

    // 如果是Response返回类型的消息，则对返回消息中对结果进行解码
    if (message instanceof Response) {
        decode(((Response) message).getResult());
    }

    // 继续将消息委托给handler，继续处理
    handler.received(channel, message);
}
```

可以看到做了三次判断，根据消息的不同会对消息的不同数据做解码。可以看到，这里用到装饰模式后，在处理消息的前面做了解码的处理，并且还能继续委托给handler来处理消息，通过组合做到了功能的叠加。

```
private void decode(Object message) {
    // 如果消息类型是Decodeable，进一步调用Decodeable的decode来解码
    if (message != null && message instanceof Decodeable) {
        try {
            ((Decodeable) message).decode();
            if (log.isDebugEnabled()) {
                log.debug("Decode decodeable message " + message.getClass().getName());
            }
        } catch (Throwable e) {
            if (log.isWarnEnabled()) {
                log.warn("Call Decodeable.decode failed: " + e.getMessage(), e);
            }
        } // ~ end of catch
    } // ~ end of if
} // ~ end of method decode
```

可以看到这是解析消息的逻辑，当消息是Decodeable类型，还会继续调用Decodeable的decode方法来进行解析。它的实现类后续会讲解到。

（八）MultiMessageHandler

该类是多消息处理器的抽象类。同样继承了AbstractChannelHandlerDelegate类，我们来看看它的received方法：

```
@SuppressWarnings("unchecked")
@Override
public void received(Channel channel, Object message) throws RemotingException {
    // 当消息为多消息时 循环交给handler处理接收到当消息
    if (message instanceof MultiMessage) {
        MultiMessage list = (MultiMessage) message;
        for (Object obj : list) {
            handler.received(channel, obj);
        }
    } else {
        // 如果是单消息，就直接交给handler处理器
        handler.received(channel, message);
    }
}
```

逻辑很简单，当消息是多消息类型时，也就是一次性接收到多条消息的情况，循环去处理消息，当消息是单消息时候，直接交给handler去处理。

（九）WrappedChannelHandler

该类跟AbstractChannelHandlerDelegate的作用类似，都是装饰模式中的装饰角色，其中的所有实现方法都直接调用被装饰的handler属性的方法，该类是为了添加线程池的功能，它的子类都是去关心哪些消息是需要分发到线程池的，哪些消息直接由I / O线程执行，现在版本有四种场景，也就是它的四个子类，下面我一一描述。

```
public WrappedChannelHandler(ChannelHandler handler, URL url) {
    this.handler = handler;
    this.url = url;
    // 创建线程池
    executor = (ExecutorService)
ExtensionLoader.getExtensionLoader(ThreadPool.class).getAdaptiveExtension().getExecutor(url);

    // 设置组件的key
    String componentKey = Constants.EXECUTOR_SERVICE_COMPONENT_KEY;
    if (Constants.CONSUMER_SIDE.equalsIgnoreCase(url.getParameter(Constants.SIDE_KEY))) {
        componentKey = Constants.CONSUMER_SIDE;
    }
    // 获得dataStore实例
    DataStore dataStore = ExtensionLoader.getExtensionLoader(DataStore.class).getDefaultExtension();
    // 把线程池放到dataStore中缓存
    dataStore.put(componentKey, Integer.toString(url.getPort()), executor);
}
```

可以看到构造方法除了属性的填充以外，线程池是基于dubbo 的SPI Adaptive机制创建的，在dataStore中把线程池加进去， 该线程池就是AbstractClient 或 AbstractServer 从 DataStore 获得的线程池。

```
public ExecutorService getExecutorService() {
    // 首先返回的不是共享线程池，是该类的线程池
    ExecutorService cexecutor = executor;
    // 如果该类的线程池关闭或者为空，则返回的是共享线程池
    if (cexecutor == null || cexecutor.isShutdown()) {
        cexecutor = SHARED_EXECUTOR;
    }
    return cexecutor;
}
```

该方法是获得线程池的实例，不过该类里面有两个线程池，还加入了一个共享线程池，共享线程池优先级较低。

（十）ExecutionChannelHandler

该类继承了WrappedChannelHandler，也是增强了功能，处理的是接收请求消息时，把请求消息分发到线程池，而除了请求消息以外，其他消息类型都直接通过I / O线程直接执行。

```
@Override
public void received(Channel channel, Object message) throws RemotingException {
    // 获得线程池实例
    ExecutorService cexecutor = getExecutorService();
    // 如果消息是request类型，才会分发到线程池，其他消息，如响应，连接，断开连接，心跳将由I / O线程直接执行。
    if (message instanceof Request) {
        try {
            // 把请求消息分发到线程池
            cexecutor.execute(new ChannelEventRunnable(channel, handler, ChannelState.RECEIVED, message));
        } catch (Throwable t) {
            // FIXME: when the thread pool is full, SERVER_THREADPOOL_EXHAUSTED_ERROR cannot return properly,
            // therefore the consumer side has to wait until gets timeout. This is a temporary solution to
prevent
            // this scenario from happening, but a better solution should be considered later.
            // 当线程池满了，SERVER_THREADPOOL_EXHAUSTED_ERROR错误无法正常返回
            // 因此消费者方必须等到超时。这是一种预防的临时解决方案，所以这里直接返回该错误
            if (t instanceof RejectedExecutionException) {
                Request request = (Request) message;
                if (request.isTwoWay()) {
                    String msg = "Server side(" + url.getIp() + "," + url.getPort()
                            + ") thread pool is exhausted, detail msg:" + t.getMessage();
                    Response response = new Response(request.getId(), request.getVersion());
                    response.setStatus(Response.SERVER_THREADPOOL_EXHAUSTED_ERROR);
                    response.setErrorMessage(msg);
                    channel.send(response);
                    return;
```

上述就可以都看到对于请求消息的处理，其中有个打补丁的方式是当线程池满了的时候，消费者只能等待请求超时，所以这里直接返回线程池满的错误。

（十一）AllChannelHandler

该类也继承了WrappedChannelHandler，也是为了增强功能，处理的是连接、断开连接、捕获异常以及接收到的所有消息都分发到线程池。

```
@Override
public void connected(Channel channel) throws RemotingException {
    ExecutorService cexecutor = getExecutorService();
    try {
        // 把连接操作分发到线程池处理
        cexecutor.execute(new ChannelEventRunnable(channel, handler, ChannelState.CONNECTED));
    } catch (Throwable t) {
        throw new ExecutionException("connect event", channel, getClass() + " error when process connected event
.", t);
    }
}

@Override
public void disconnected(Channel channel) throws RemotingException {
    ExecutorService cexecutor = getExecutorService();
    try {
        // 把断开连接操作分发到线程池处理
        cexecutor.execute(new ChannelEventRunnable(channel, handler, ChannelState.DISCONNECTED));
    } catch (Throwable t) {
        throw new ExecutionException("disconnect event", channel, getClass() + " error when process disconnected
event .", t);
    }
}

@Override
public void received(Channel channel, Object message) throws RemotingException {
```

可以看到，所有操作以及消息都分到到线程池中。并且注意操作不同，传入的状态也不同。

（十二）ConnectionOrderedChannelHandler

该类也是继承了WrappedChannelHandler，增强功能，该类是把连接、取消连接以及接收到的消息都分发到线程池，但是不同的是，该类自己创建了一个跟连接相关的线程池，把连接操作和断开连接操分发到该线程池，而接收到的消息则分发到WrappedChannelHandler的线程池中。来看看具体的实现。

```
/**
 * 连接线程池
 */
protected final ThreadPoolExecutor connectionExecutor;
/**
 * 连接队列大小限制
 */
private final int queuewarninglimit;

public ConnectionOrderedChannelHandler(ChannelHandler handler, URL url) {
    super(handler, url);
    // 获得线程名，默认是Dubbo
    String threadName = url.getParameter(Constants.THREAD_NAME_KEY, Constants.DEFAULT_THREAD_NAME);
    // 创建连接线程池
    connectionExecutor = new ThreadPoolExecutor(1, 1,
            0L, TimeUnit.MILLISECONDS,
            new LinkedBlockingQueue<Runnable>(url.getPositiveParameter(Constants.CONNECT_QUEUE_CAPACITY,
Integer.MAX_VALUE)),
            new NamedThreadFactory(threadName, true),
            new AbortPolicyWithReport(threadName, url)
    );  // FIXME There's no place to release connectionExecutor!
    // 设置工作队列限制，默认是1000
    queuewarninglimit = url.getParameter(Constants.CONNECT_QUEUE_WARNING_SIZE,
Constants.DEFAULT_CONNECT_QUEUE_WARNING_SIZE);
}
```

可以属性中有一个连接线程池，看到在构造函数里创建了该线程池，而queuewarninglimit是用来限制连接线程池的工作队列长度，比较简单。来看看连接和断开连接到逻辑。

```
@Override
public void connected(Channel channel) throws RemotingException {
    try {
        // 核对工作队列长度
        checkQueueLength();
        // 分发连接操作
        connectionExecutor.execute(new ChannelEventRunnable(channel, handler, ChannelState.CONNECTED));
    } catch (Throwable t) {
        throw new ExecutionException("connect event", channel, getClass() + " error when process connected event .",
t);
    }
}

@Override
public void disconnected(Channel channel) throws RemotingException {
    try {
        // 核对工作队列长度
        checkQueueLength();
        // 分发断开连接操作
        connectionExecutor.execute(new ChannelEventRunnable(channel, handler, ChannelState.DISCONNECTED));
    } catch (Throwable t) {
        throw new ExecutionException("disconnected event", channel, getClass() + " error when process disconnected
event .", t);
    }
}
```

可以看到，这两个操作都是分发到连接线程池connectionExecutor中，和AllChannelHandle类r中的分发的线程池不是同一个。而ConnectionOrderedChannelHandler的received方法跟AllChannelHandle一样，我就不贴出来。

（十三）MessageOnlyChannelHandler

该类也是继承了WrappedChannelHandler，是WrappedChannelHandler的最后一个子类，也是增强功能，不过该类只是处理了所有的消息分发到线程池。可以看到源码，比较简单：

```
@Override
public void received(Channel channel, Object message) throws RemotingException {
    // 获得线程池实例
    ExecutorService cexecutor = getExecutorService();
    try {
        // 把消息分发到线程池
        cexecutor.execute(new ChannelEventRunnable(channel, handler, ChannelState.RECEIVED, message));
    } catch (Throwable t) {
        throw new ExecutionException(message, channel, getClass() + " error when process received event .", t);
    }
}
```

下面我讲讲解五种线程池的调度策略，也就是我在[《dubbo源码解析（八）远程通信——开篇》]中提到的Dispatcher接口的五种实现，分别是AllDispatcher、DirectDispatcher、MessageOnlyDispatcher、ExecutionDispatcher、ConnectionOrderedDispatcher。

（十四）AllDispatcher

```
public class AllDispatcher implements Dispatcher {

    public static final String NAME = "all";

    @Override
    public ChannelHandler dispatch(ChannelHandler handler, URL url) {
        // 线程池调度方法：任何消息以及操作都分发到线程池中
        return new AllChannelHandler(handler, url);
    }

}
```

对照着上述讲到的AllChannelHandler，是不是很清晰这种线程池的调度方法。并且该调度方法是默认的调度方法。

（十五）ConnectionOrderedDispatcher

```
public class ConnectionOrderedDispatcher implements Dispatcher {

    public static final String NAME = "connection";

    @Override
    public ChannelHandler dispatch(ChannelHandler handler, URL url) {
        // 线程池调度方法：连接、断开连接分发到到线程池和其他消息分发到线程池不是同一个
        return new ConnectionOrderedChannelHandler(handler, url);
    }

}
```

对照上述讲到的ConnectionOrderedChannelHandler，也很清晰该线程池调度方法。

（十六）DirectDispatcher

```
public class DirectDispatcher implements Dispatcher {

    public static final String NAME = "direct";

    @Override
    public ChannelHandler dispatch(ChannelHandler handler, URL url) {
        // 直接处理消息，不分发到线程池
        return handler;
    }

}
```

该线程池调度方法是不调度线程池，直接执行。

（十七）ExecutionDispatcher

```
public class ExecutionDispatcher implements Dispatcher {

    public static final String NAME = "execution";

    @Override
    public ChannelHandler dispatch(ChannelHandler handler, URL url) {
        // 线程池调度方法：只有请求消息分发到线程池，其他都直接执行
        return new ExecutionChannelHandler(handler, url);
    }

}
```

对照着上述的ExecutionChannelHandler讲解，也可以很清晰的看出该线程池调度策略。

（十八）MessageOnlyDispatcher

```
public class MessageOnlyDispatcher implements Dispatcher {

    public static final String NAME = "message";

    @Override
    public ChannelHandler dispatch(ChannelHandler handler, URL url) {
        // 只要是接收到的消息，都分发到线程池
        return new MessageOnlyChannelHandler(handler, url);
    }

}
```

对照着上述讲到的MessageOnlyChannelHandler，可以很清晰该线程池调度策略。

（十九）ChannelHandlers

该类是通道处理器工厂，会对传入的handler进行一次包装，无论是Client还是Server都会做这样的处理，也就是做了一些功能上的增强，就像上述我说到的装饰模式中的那些功能。

我们来看看源码：

```
public static ChannelHandler wrap(ChannelHandler handler, URL url) {
    return ChannelHandlers.getInstance().wrapInternal(handler, url);
}

protected ChannelHandler wrapInternal(ChannelHandler handler, URL url) {
    // 调用了多消息处理器，对心跳消息进行了功能加强
    return new MultiMessageHandler(new HeartbeatHandler(ExtensionLoader.getExtensionLoader(Dispatcher.class)
            .getAdaptiveExtension().dispatch(handler, url)));
}
```

最关键的是这两个方法，看第二个方法，其实就是包装了MultiMessageHandler功能，增加了多消息处理的功能，以及对心跳消息做了功能增强。

（二十）AbstractCodec

实现 Codec**2** 接口，，其中实现了一些编解码的公共逻辑。

1.checkPayload

```
protected static void checkPayload(Channel channel, long size) throws IOException {
    // 默认长度
    int payload = Constants.DEFAULT_PAYLOAD;
    if (channel != null && channel.getUrl() != null) {
        // 优先从url中获得消息长度配置，如果没有则用默认长度
        payload = channel.getUrl().getParameter(Constants.PAYLOAD_KEY, Constants.DEFAULT_PAYLOAD);
    }
    // 如果消息长度过长，则报错
    if (payload > 0 && size > payload) {
        ExceedPayloadLimitException e = new ExceedPayloadLimitException("Data length too large: " + size + ", max
payload: " + payload + ", channel: " + channel);
        logger.error(e);
        throw e;
    }
}
```

该方法是检验消息长度。

2.getSerialization

```
protected Serialization getSerialization(Channel channel) {
    return CodecSupport.getSerialization(channel.getUrl());
}
```

该方法是获得序列化对象。

3.isClientSide

```
protected boolean isClientSide(Channel channel) {
    // 获得是side对应的value
    String side = (String) channel.getAttribute(Constants.SIDE_KEY);
    if ("client".equals(side)) {
        return true;
    } else if ("server".equals(side)) {
        return false;
    } else {
        InetSocketAddress address = channel.getRemoteAddress();
        URL url = channel.getUrl();
        // 判断url的主机地址是否和远程地址一样，如果是，则判断为client，如果不是，则判断为server
        boolean client = url.getPort() == address.getPort()
                && NetUtils.filterLocalHost(url.getIp()).equals(
                NetUtils.filterLocalHost(address.getAddress()
                        .getHostAddress()));
        // 把value设置进去
        channel.setAttribute(Constants.SIDE_KEY, client ? "client"
                : "server");
        return client;
    }
}
```

该方法是判断是否为客户端侧的通道。

4.isServerSide

```
protected boolean isServerSide(Channel channel) {
    return !isClientSide(channel);
}
```

该方法是判断是否为服务端侧的通道。

（二十一）TransportCodec

该类是传输编解码器，使用 Serialization 进行序列化/反序列化，直接编解码。关于序列化为会在后续文章中介绍。

```
@Override
public void encode(Channel channel, ChannelBuffer buffer, Object message) throws IOException {
    // 获得序列化的 ObjectOutput 对象
    OutputStream output = new ChannelBufferOutputStream(buffer);
    ObjectOutput objectOutput = getSerialization(channel).serialize(channel.getUrl(), output);
    // 写入 ObjectOutput
    encodeData(channel, objectOutput, message);
    objectOutput.flushBuffer();
    // 释放
    if (objectOutput instanceof Cleanable) {
        ((Cleanable) objectOutput).cleanup();
    }
}

@Override
public Object decode(Channel channel, ChannelBuffer buffer) throws IOException {
    // 获得反序列化的 ObjectInput 对象
    InputStream input = new ChannelBufferInputStream(buffer);
    ObjectInput objectInput = getSerialization(channel).deserialize(channel.getUrl(), input);
    // 读取 ObjectInput
    Object object = decodeData(channel, objectInput);
    // 释放
    if (objectInput instanceof Cleanable) {
        ((Cleanable) objectInput).cleanup();
    }
    return object;
```

该类关键方法就是编码和解码，比较好理解，直接进行了序列化和反序列化。

（二十二）CodecAdapter

该类是Codec 的适配器，用到了适配器模式，把Codec适配成Codec2。将Codec的编码和解码方法都适配成Codec2。比如很多时候都只能用Codec2的编解码器，但是有的时候需要用Codec，但是不能满足导致只能加入适配器来完成使用。

```
@Override
public void encode(Channel channel, ChannelBuffer buffer, Object message)
        throws IOException {
    UnsafeByteArrayOutputStream os = new UnsafeByteArrayOutputStream(1024);
    // 调用旧的编解码器的编码
    codec.encode(channel, os, message);
    buffer.writeBytes(os.toByteArray());
}

@Override
public Object decode(Channel channel, ChannelBuffer buffer) throws IOException {
    byte[] bytes = new byte[buffer.readableBytes()];
    int savedReaderIndex = buffer.readerIndex();
    buffer.readBytes(bytes);
    UnsafeByteArrayInputStream is = new UnsafeByteArrayInputStream(bytes);
    // 调用旧的编解码器的解码
    Object result = codec.decode(channel, is);
    buffer.readerIndex(savedReaderIndex + is.position());
    return result == Codec.NEED_MORE_INPUT ? DecodeResult.NEED_MORE_INPUT : result;
}
```

可以看到，在编码和解码的方法中都调用了codec的方法。

（二十三）ChannelDelegate、ServerDelegate、ClientDelegate

ChannelDelegate实现类Channel，ServerDelegate实现了Server，ClientDelegate实现了Client，都用到了装饰模式，都作为装饰模式中的装饰角色，所以类中的所有实现方法都调用了属性的方法。具体代码就不贴了，朋友们可以自行查看。

（二十四）ChannelHandlerAdapter

该类实现了ChannelHandler接口，是通道处理器适配类，该类中所有实现方法都是空的，所有想实现ChannelHandler接口的类可以直接继承该类，选择需要实现的方法进行实现，不需要实现ChannelHandler接口中所有方法。

（二十五）ChannelHandlerDispatcher

该类是通道处理器调度器，其中缓存了所有通道处理器，有一个通道处理器集合。并且每个操作都会去遍历该集合，执行相应的操作，例如：

```
@Override
public void connected(Channel channel) {
    // 遍历通道处理器集合
    for (ChannelHandler listener : channelHandlers) {
        try {
            // 连接
            listener.connected(channel);
        } catch (Throwable t) {
            logger.error(t.getMessage(), t);
        }
    }
}
```

(二十六) CodecSupport

该类是编解码工具类，提供查询 Serialization 的功能。

```
/**
 * 序列化对象集合 key为序列化类型编号
 */
private static Map<Byte, Serialization> ID_SERIALIZATION_MAP = new HashMap<Byte, Serialization>();
/**
 * 序列化扩展名集合 key为序列化类型编号 value为序列化扩展名
 */
private static Map<Byte, String> ID_SERIALIZATIONNAME_MAP = new HashMap<Byte, String>();

static {
    // 利用dubbo 的SPI机制获得序列化扩展名
    Set<String> supportedExtensions =
ExtensionLoader.getExtensionLoader(Serialization.class).getSupportedExtensions();
    for (String name : supportedExtensions) {
        // 获得相应扩展名的序列化实现
        Serialization serialization = ExtensionLoader.getExtensionLoader(Serialization.class).getExtension(name);
        byte idByte = serialization.getContentTypeId();
        if (ID_SERIALIZATION_MAP.containsKey(idByte)) {
            logger.error("Serialization extension " + serialization.getClass().getName()
                    + " has duplicate id to Serialization extension "
                    + ID_SERIALIZATION_MAP.get(idByte).getClass().getName()
                    + ", ignore this Serialization extension");
            continue;
        }
        // 缓存序列化实现
        ID_SERIALIZATION_MAP.put(idByte, serialization);
```

可以看到该类中缓存了所有的序列化对象和序列化扩展名。可以从中拿到Serialization。

（二十七）ExceedPayloadLimitException

该类是消息长度限制异常。

```
public class ExceedPayloadLimitException extends IOException {
    private static final long serialVersionUID = -1112322085391551410L;

    public ExceedPayloadLimitException(String message) {
        super(message);
    }
}
```

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Transport层的相关设计和逻辑、介绍dubbo-remoting-api中的transport包内的源码解，其中关键的是整个设计都在使用装饰模式，传输层中关键的编解码器以及客户端、服务的、通道的抽象，还有关键的就是线程池的调度方法，熟悉那五种调度方法，对消息的处理。整个传输层核心的消息，很多操作围绕着消息展开。下一篇我会讲解交换层exchange部分。如果我在哪一部分写的不够到位或者写错了，欢迎给我提意见，我的私人微信号码：HUA799695226。

阅读 1.3k • 更新于 11月8日

赞 3 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

2 条评论

得票 • 时间

撰写评论 ...

提交评论

_拈花_微笑 ： 楼主的文章很有深度，继续更新啊，我正好跟着把dubbo系统学习一下

_ • 回复 • 2月1日

crazyhzm： 我会持续更新，有不明白或者错误的地方望提出来

_ • 回复 • 2月1日

推荐阅读

微信开源mars源码分析1—上层samples分析

微信已经开源了mars，但是市面上相关的文章较少，即使有也是多在于使用xlog等这些，那么这次我希望能够从stn这个直接用于i...

机械面条 • 阅读 81

Dubbo 源码分析 - 自适应拓展原理

我在上一篇文章中分析了Dubbo的SPI机制，DubboSPI是Dubbo框架的核心。Dubbo中的很多拓展都是通过SPI机制进行加载的，比...

coolblog • 阅读 329

Dubbo分析之Exchange层

Dubbo分析Serialize层Dubbo分析之Transport层Dubbo分析之Exchange层紧接着上文Dubbo分析之Transport层，本文继续介绍Exch...

Coding狗 • 阅读 50