crazyhzm 265

Dubbo源码解析（八）远程通信——开篇

java dubbo 阅读约 21 分钟

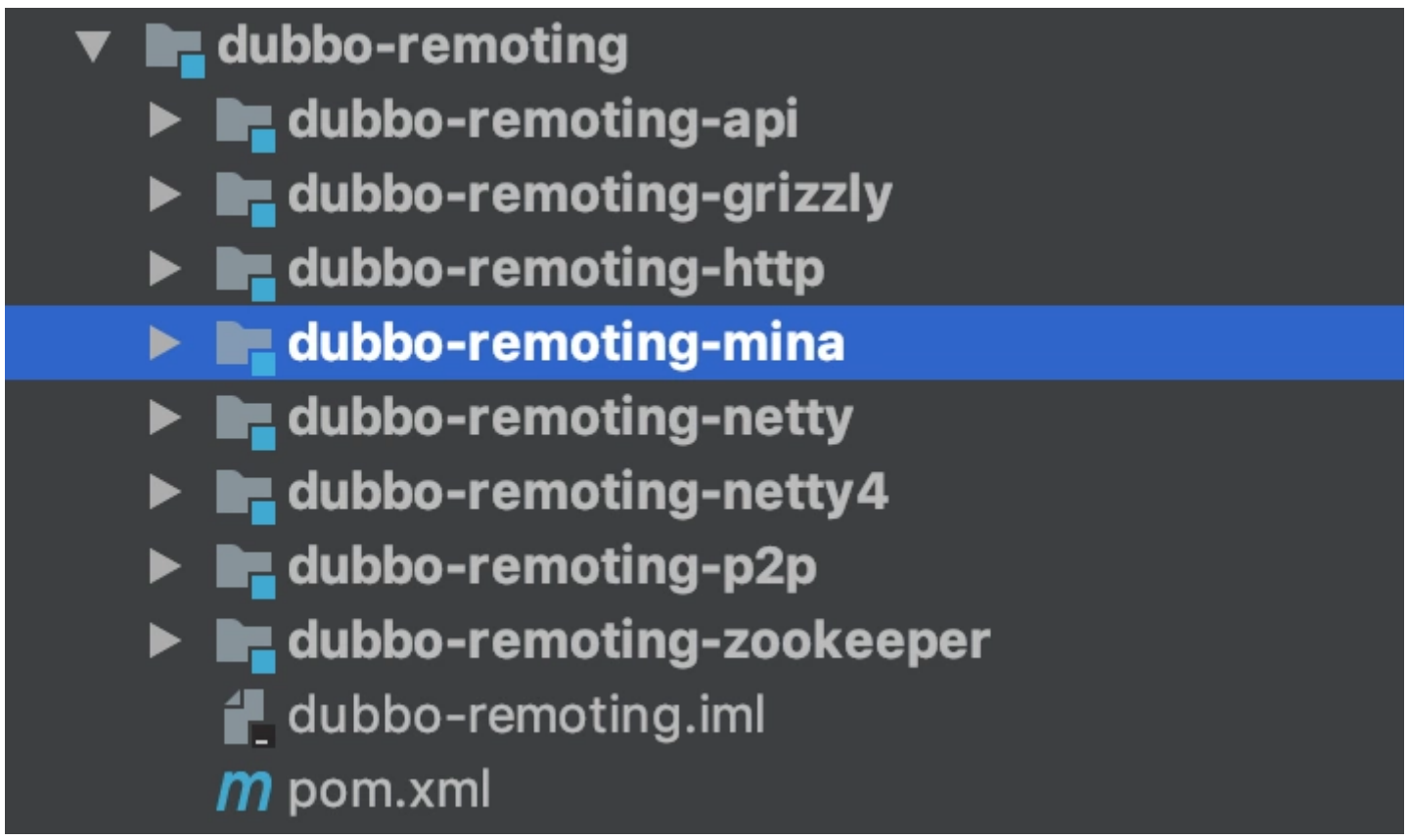
远程通讯——开篇

> 目标：介绍之后解读远程通讯模块的内容如何编排、介绍dubbo-remoting-api中的包结构设计以及最外层的的源码解析。

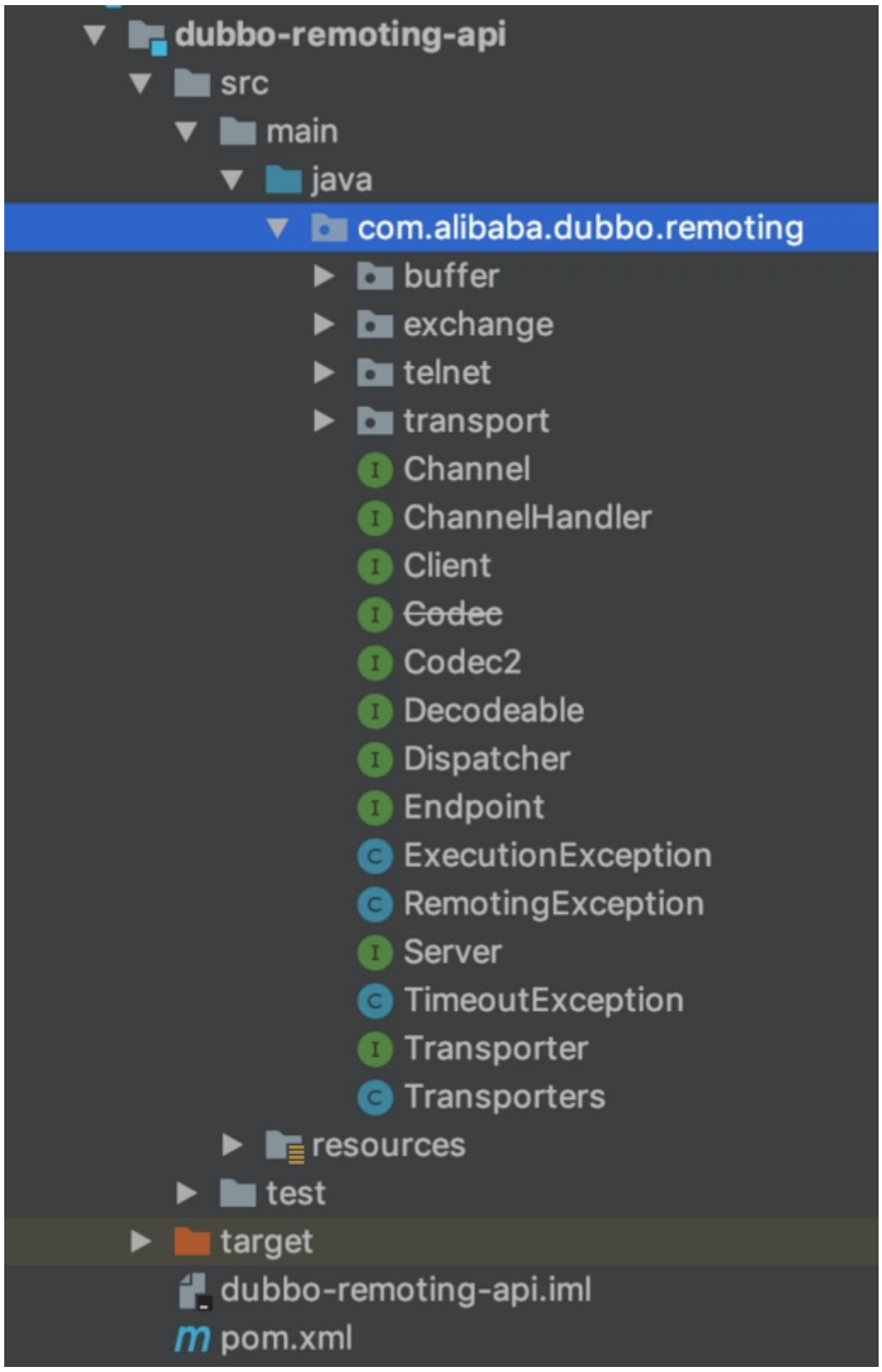
前言

服务治理框架中可以大致分为服务通信和服务管理两个部分，前面我先讲到有关注册中心的内容，也就是服务管理，当然dubbo的服务管理还包括监控中心、 telnet 命令，它们起到的是人工的服务管理作用，这个后续再介绍。接下来我要讲解的就是跟服务通信有关的部分，也就是远程通讯模块。我在《dubbo源码解析（一）Hello,Dubbo》的"（六）dubbo-remoting——远程通信模块"中提到过一些内容。该模块中提供了多种客户端和服务端通信的功能，而在对NIO框架选型上，dubbo交由用户选择，它集成了mina、netty、grizzly等各类NIO框架来搭建NIO服务器和客户端，并且利用dubbo的SPI扩展机制可以让用户自定义选择。如果对SPI不太了解的朋友可以查看《dubbo源码解析（二）Dubbo扩展机制SPI》。

接下来我们先来看看dubbo-remoting的包结构：



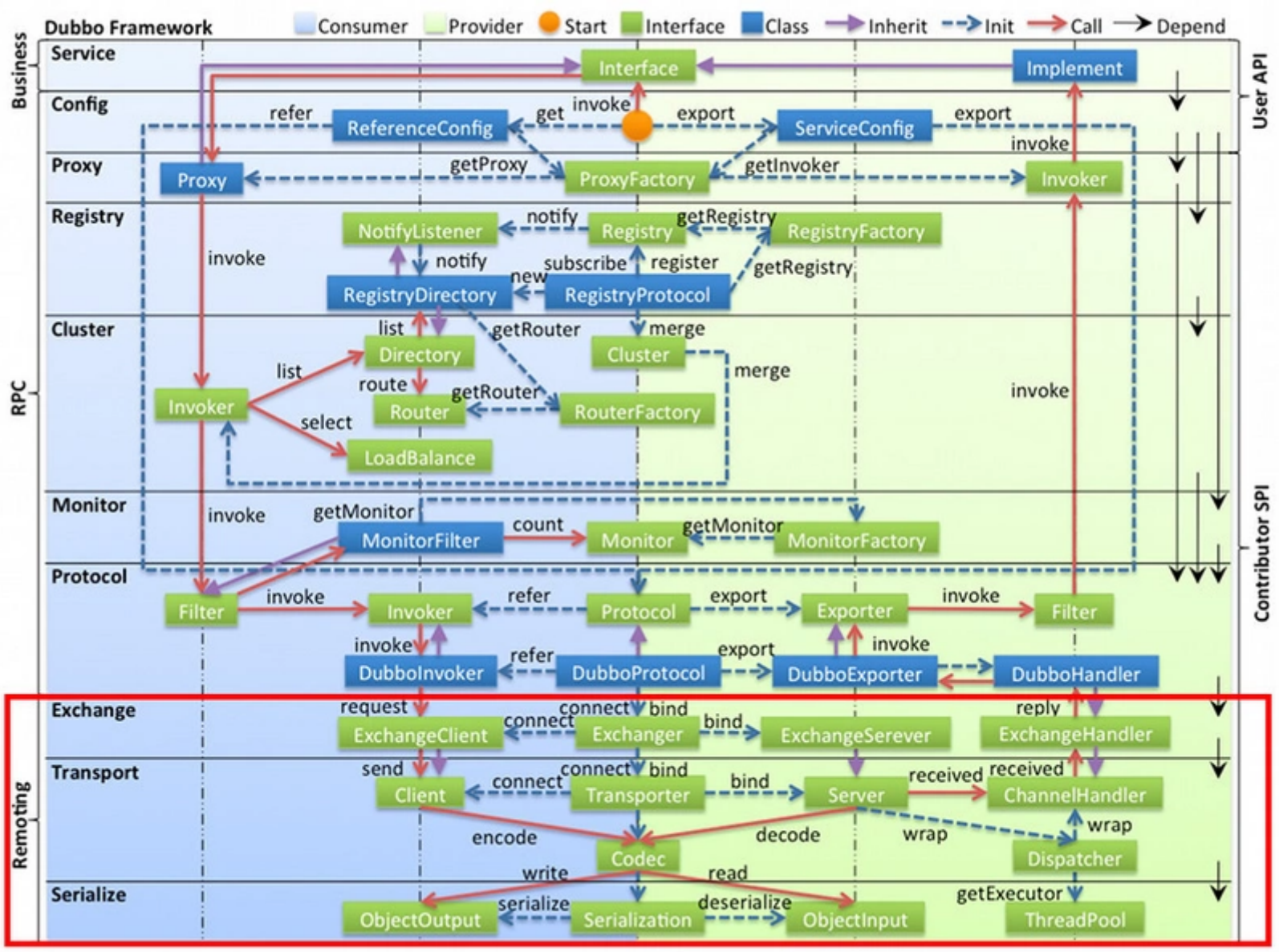
我接下来解读远程通讯模块的内容并不是按照一个包一篇文章的编排，先来看看dubbo-remoting-api的包结构：



可以看到，大篇幅的逻辑在dubbo-remoting-api中，所以我对于dubbo-remoting-api的解读会分为下面五个部分来说明，其中第五点会在本文介绍，其他四点会分别用四篇文章来介绍：

1. buffer包：缓冲在NIO框架中是很重要的存在，各个NIO框架都实现了自己相应的缓存操作。这个buffer包下包括了缓冲区的接口以及抽象
2. exchange包：信息交换层，其中封装了请求响应模式，在传输层之上重新封装了 Request-Response 语义，为了满足RPC的需求。这层可以认为专注在Request和Response携带的信息上。该层是RPC调用的通讯基础之一。
3. telnet包：dubbo支持通过telnet命令来进行服务治理，该包下就封装了这些通用指令的逻辑实现。
4. transport包：网络传输层，它只负责单向消息传输，是对 Mina, Netty, Grizzly 的抽象，它也可以扩展 UDP 传输。该层是RPC调用的通讯基础之一。
5. 最外层的源码：该部分我会在下面之间给出介绍。

为什么我要把一个api分成这么多文章来讲解，我们先来看看下面的图：



我们可以看到红框内的是远程通讯的框架，序列化我会在后面的主题中介绍，而Exchange层和Transport层在框架设计中起到了很重要的作用，也是支撑Remoting的核心，所以我要分开来介绍。

除了上述的五点外，根据惯例，我还是会分别介绍dubbo支持的实现客户端和服务端通信的七种方案，也就是说该远程通讯模块我会用12篇文章详细的讲解。

最外层源码解析

（一）接口Endpoint

dubbo抽象出一个端的概念，也就是Endpoint接口，这个端就是一个点，而点对点之间是可以双向传输。在端的基础上在衍生出通道、客户端以及服务端的概念，也就是下面要介绍的Channel、Client、Server三个接口。在传输层，其实Client和Server的区别只是在语义上区别，并不区分请求和应答职责，在交换层客户端和服务端也是一个点，但是已经是有方向的点，所以区分了明确的请求和应答职责。两者都具备发送的能力，只是客户端和服务端所关注的事情不一样，这个在后面会分开介绍，而Endpoint接口抽象的方法就是它们共同拥有的方法。这也就是它们都能被抽象成端的原因。

来看一下它的源码：

```
    // 获得该端的通道处理器
    ChannelHandler getChannelHandler();

    // 获得该端的本地地址
    InetSocketAddress getLocalAddress();

    // 发送消息
    void send(Object message) throws RemotingException;

    // 发送消息，sent是是否已经发送的标记
    void send(Object message, boolean sent) throws RemotingException;

    // 关闭
    void close();

    // 优雅的关闭，也就是加入了等待时间
    void close(int timeout);

    // 开始关闭
    void startClose();

    // 判断是否已经关闭
    boolean isClosed();

}
```

1. 前三个方法是获得该端本身的一些属性，
2. 两个send方法是发送消息，其中第二个方法多了一个sent的参数，为了区分是否是第一次发送消息。
3. 后面几个方法是提供了关闭通道的操作以及判断通道是否关闭的操作。

（二）接口Channel

该接口是通道接口，通道是通讯的载体。还是用自动贩卖机的例子，自动贩卖机就好比是一个通道，消息发送端会往通道输入消息，而接收端会从通道读消息。并且接收端发现通道没有消息，就去做其他事情了，不会造成阻塞。所以channel可以读也可以写，并且可以异步读写。channel是client和server的传输桥梁。channel和client是一一对应的，也就是一个client对应一个channel，但是channel和server是多对一对关系，也就是一个server可以对应多个channel。

```
public interface Channel extends Endpoint {

    // 获得远程地址
    InetSocketAddress getRemoteAddress();

    // 判断通道是否连接
    boolean isConnected();

    // 判断是否有该key的值
    boolean hasAttribute(String key);

    // 获得该key对应的值
    Object getAttribute(String key);

    // 添加属性
    void setAttribute(String key, Object value);

    // 移除属性
    void removeAttribute(String key);

}
```

可以看到Channel继承了Endpoint，也就是端抽象出来的方法也同样是channel所需要的。上面的几个方法很好理解，我就不多介绍了。

（三）接口ChannelHandler

```
@SPI
public interface ChannelHandler {

    // 连接该通道
    void connected(Channel channel) throws RemotingException;

    // 断开该通道
    void disconnected(Channel channel) throws RemotingException;

    // 发送给这个通道消息
    void sent(Channel channel, Object message) throws RemotingException;

    // 从这个通道内接收消息
    void received(Channel channel, Object message) throws RemotingException;

    // 从这个通道内捕获异常
    void caught(Channel channel, Throwable exception) throws RemotingException;

}
```

该接口是负责channel中的逻辑处理，并且可以看到这个接口有注解@SPI，是个可扩展接口，到时候都会在下面介绍各类NIO框架的时候会具体讲到它的实现类。

（四）接口Client

```
public interface Client extends Endpoint, Channel, Resetable {

    // 重连
    void reconnect() throws RemotingException;

    // 重置，不推荐使用
    @Deprecated
    void reset(com.alibaba.dubbo.common.Parameters parameters);

}
```

客户端接口，可以看到它继承了Endpoint、Channel和Resetable接口，继承Endpoint的原因上面我已经提到过了，客户端和服务端其实只是语义上的不同，客户端就是一个点。继承Channel是因为客户端跟通道是一一对应的，所以做了这样的设计，还继承了Resetable接口是为了实现reset方法，该方法，不过已经打上@Deprecated注解，不推荐使用。除了这些客户端就只需要关注一个重连的操作。

这里插播一个公共模块下的接口Resetable：

```
public interface Resetable {

    // 用于根据新传入的 url 属性，重置自己内部的一些属性
    void reset(URL url);

}
```

该方法就是根据新的url来重置内部的属性。

（五）接口Server

```
public interface Server extends Endpoint, Resetable {

    // 判断是否绑定到本地端口，也就是该服务器是否启动成功，能够连接、接收消息，提供服务。
    boolean isBound();

    // 获得连接该服务器的通道
    Collection<Channel> getChannels();

    // 通过远程地址获得该地址对应的通道
    Channel getChannel(InetSocketAddress remoteAddress);

    @Deprecated
    void reset(com.alibaba.dubbo.common.Parameters parameters);

}
```

该接口是服务端接口，继承了Endpoint和Resetable，继承Endpoint是因为服务端也是一个点，继承Resetable接口是为了继承reset方法。除了这些以外，服务端独有的是检测是否启动成功，还有事获得连接该服务器上所有通道，这里获得所有通道其实就意味着获得了所有连接该服务器的客户端，因为客户端和通道是一一对应的。

（六）接口Codec && Codec2

这两个都是编解码器，那么什么叫做编解码器，在网络中只是讲数据看成是原始的字节序列，但是我们的应用程序会把这些字节组织成有意义的信息，那么网络字节流和数据间的转化就是很常见的任务。而编码器是讲应用程序的数据转化为网络格式，解码器则是讲网络格式转化为应用程序，同时具备这两种功能的单一组件就叫编解码器。在dubbo中Codec是老的编解码器接口，而Codec2是新的编解码器接口，并且dubbo已经用CodecAdapter把Codec适配成Codec2了。所以在这里我就介绍Codec2接口，毕竟人总要往前看。

```
@SPI
public interface Codec2 {
      //编码
    @Adaptive({Constants.CODEC_KEY})
    void encode(Channel channel, ChannelBuffer buffer, Object message) throws IOException;
    //解码
    @Adaptive({Constants.CODEC_KEY})
    Object decode(Channel channel, ChannelBuffer buffer) throws IOException;

    enum DecodeResult {
        // 需要更多输入和忽略一些输入
        NEED_MORE_INPUT, SKIP_SOME_INPUT
    }

}
```

因为是编解码器，所以有两个方法分别是编码和解码，上述有以下几个关注的：

1. Codec2是一个可扩展的接口，因为有@SPI注解。
2. 用到了Adaptive机制，首先去url中寻找key为codec的value，来加载url携带的配置中指定的codec的实现。
3. 该接口中有个枚举类型DecodeResult，因为解码过程中，需要解决 TCP 拆包、粘包的场景，所以增加了这两种解码结果，关于 TCP 拆包、粘包的场景我就不多解释，不懂得朋友可以google一下。

（七）接口Decodeable

```
public interface Decodeable {

    //解码
    public void decode() throws Exception;

}
```

该接口是可解码的接口，该接口有两个作用，第一个是在调用真正的decode方法实现的时候会有一些校验，判断是否可以解码，并且对解码失败会有一些消息设置；第二个是被用来message核对用的。后面看具体的实现会更了解该接口的作用。

（八）接口Dispatcher

```
@SPI(AllDispatcher.NAME)
public interface Dispatcher {

    // 调度
    @Adaptive({Constants.DISPATCHER_KEY, "dispather", "channel.handler"})
    // The last two parameters are reserved for compatibility with the old configuration
    ChannelHandler dispatch(ChannelHandler handler, URL url);

}
```

该接口是调度器接口，dispatch是线程池的调度方法，这边有几个注意点：

1. 该接口是一个可扩展接口，并且默认实现AllDispatcher，也就是所有消息都派发到线程池，包括请求，响应，连接事件，断开事件，心跳等。
2. 用了Adaptive注解，也就是按照URL中配置来加载实现类，后面两个参数是为了兼容老版本，如果这是三个key对应的值都为空，就选择AllDispatcher来实现。

（九）接口Transporter

```
@SPI("netty")
public interface Transporter {

    // 绑定一个服务器
    @Adaptive({Constants.SERVER_KEY, Constants.TRANSPORTER_KEY})
    Server bind(URL url, ChannelHandler handler) throws RemotingException;

    // 连接一个服务器，即创建一个客户端
    @Adaptive({Constants.CLIENT_KEY, Constants.TRANSPORTER_KEY})
    Client connect(URL url, ChannelHandler handler) throws RemotingException;

}
```

该接口是网络传输接口，有以下几个注意点：

1. 该接口是一个可扩展的接口，并且默认实现NettyTransporter。
2. 用了dubbo SPI扩展机制中的Adaptive注解，加载对应的bind方法，使用url携带的server或者transporter属性值，加载对应的connect方法，使用url携带的client或者transporter属性值，不了解SPI扩展机制的可以查看《dubbo源码解析（二）Dubbo扩展机制SPI》。

（十）Transporters

```
public class Transporters {

    static {
        // check duplicate jar package
        // 检查重复的 jar 包
        Version.checkDuplicate(Transporters.class);
        Version.checkDuplicate(RemotingException.class);
    }

    private Transporters() {
    }

    public static Server bind(String url, ChannelHandler... handler) throws RemotingException {
        return bind(URL.valueOf(url), handler);
    }

    public static Server bind(URL url, ChannelHandler... handlers) throws RemotingException {
        if (url == null) {
            throw new IllegalArgumentException("url == null");
        }
        if (handlers == null || handlers.length == 0) {
            throw new IllegalArgumentException("handlers == null");
        }
        ChannelHandler handler;
        // 创建handler
        if (handlers.length == 1) {
```

1. 该类用到了设计模式的外观模式，通过该类的包装，我们就不会看到内部具体的实现细节，这样降低了程序的复杂度，也提高了程序的可维护性。比如这个类，包装了调用各种实现Transporter接口的方法，通过getTransporter来获得Transporter的实现对象，具体实现哪个实现类，取决于url中携带的配置信息，如果url中没有相应的配置，则默认选择@SPI中的默认值netty。
2. bind和connect方法分别有两个重载方法，其中的操作只是把把字符串的url转化为URL对象。
3. 静态代码块中检测了一下jar包是否有重复。

（十一）RemotingException && ExecutionException && TimeoutException

这三个类是远程通信的异常类：

1. RemotingException继承了Exception类，是远程通信的基础异常。
2. ExecutionException继承了RemotingException类，ExecutionException是远程通信的执行异常。
3. TimeoutException继承了RemotingException类，TimeoutException是超时异常。

为了不影响篇幅，这三个类源码我就不介绍了，因为比较简单。

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dubbo-remoting-api中的包结构设计以及最外层的的源码解析，其中关键的是理解端的概念，明白在哪一层才区分了发送和接收的职责，后续文章会按照我上面的编排去写。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。