课程内容

1. HTTP1的劣势与HTTP2的优势
2. Http2之数据帧与Stream介绍
3. Triple协议服务调用底层原理
4. Triple协议服务响应底层原理
5. Triple流式调用底层原理分析

有道云链接：https://note.youdao.com/s/EA6gNBwT

在Dubbo2.7中，默认的是Dubbo协议，因为Dubbo协议相比较于Http1.1而言，Dubbo协议性能上是要更好的。

但是Dubbo协议自己的缺点就是不通用，假如现在通过Dubbo协议提供了一个服务，那如果想要调用该服务就必须要求服务消费者也要支持Dubbo协议，比如想通过浏览器直接调用Dubbo服务是不行的，想通过Nginx调Dubbo服务也是不行得。

而随着企业的发展，往往可能会出现公司内部使用多种技术栈，可能这个部门使用Dubbo，另外一个部门使用Spring Cloud，另外一个部门使用gRPC，那此时部门之间要想相互调用服务就比较复杂了，所以需要一个通用的、性能也好的协议，这就是Triple协议。

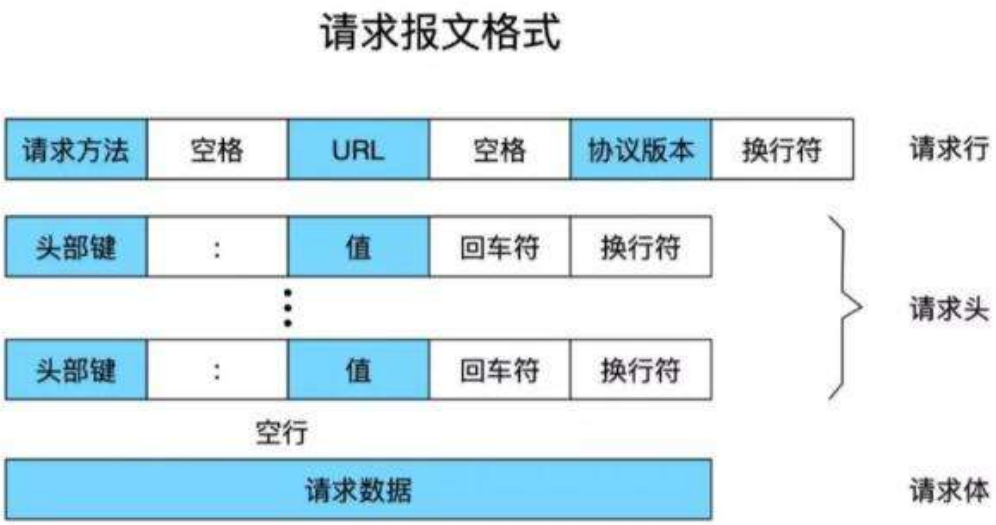
Triple协议是基于Http2协议的，也就是在使用Triple协议发送数据时，会按HTTP2协议的格式来发送数据，而HTTP2协议相比较于HTTP1协议而言，HTTP2是HTTP1的升级版，完全兼容HTTP1，而且HTTP2协议从设计层面就解决了HTTP1性能低的问题，具体看https://www.cnblogs.com/mrliuzf/p/14596005.html

另外，Google公司开发的gRPC，也基于的HTTP2，目前gRPC是云原生事实上协议标准，包括k8s/etcd等都支持gRPC协议。

所以Dubbo3.0为了能够更方便的和k8s进行通信，在实现Triple的时候也兼容了gRPC，也就是可以用gPRC的客户端调用Dubbo3.0所提供的triple服务，也可以用triple服务调用gRPC的服务，这些前面有演示。

HTTP2简单介绍

因为Triple协议是基于HTTP2协议的，所以我们得先大概了解一下HTTP2，我们比较熟悉的是HTTP1，比如一个HTTP1的请求数据格式如下：



- 样例：

```
GET /department/87423/users HTTP/1.1
host: www.xxx.com
accept: application/json
accept-encoding: gzip, deflate, br
accept-language: zh-CN,zh;q=0.9
user-agent: AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.109

name=flyhero
```

表示在使用HTTP1协议时，需要把要发送的数据组织成上面这个格式，比如，我现在想要发送"hello"这个字符串，你可以这样发：

```
GET /服务器地址 HTTP/1.1 换行符
mycontent: hello 回车符换行符
context-length: 0 回车符换行符
回车符换行符
```

也可以这样发：

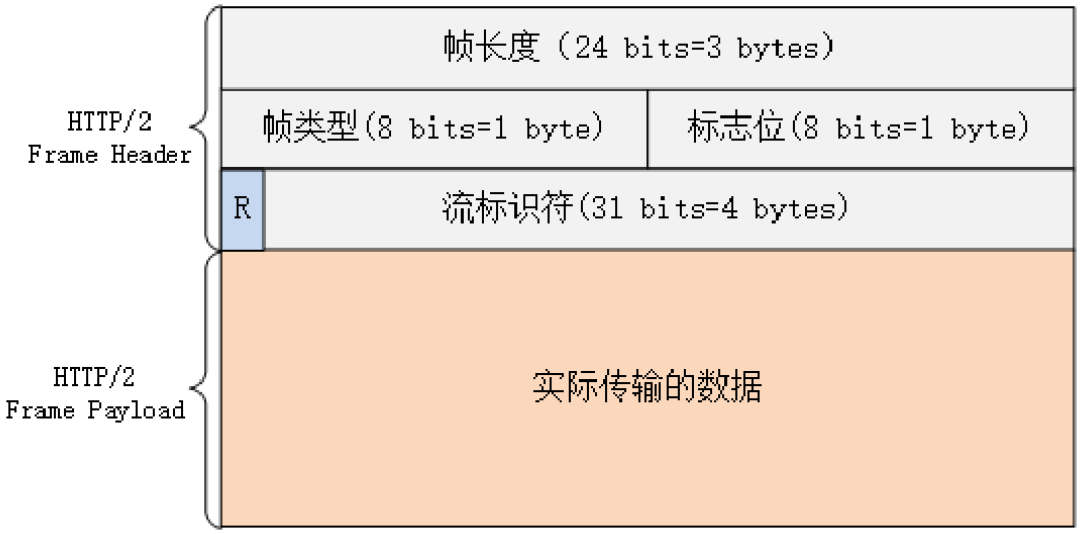
```
GET /服务器地址 HTTP/1.1 换行符
context-length: 5 回车符换行符
回车符换行符
hello
```

不管怎么发，我们发现，我们只是想把“hello”这个字符串发送给服务器，但是实际上我得额外发送很多其他的字符，最终把请求行与请求头的字符采用ascii编码成字节，可以通过Content-Type来指定请求体的编码，这样服务器接收到这个请求后，就能解析出该请求的具体内容了。

不过HTTP1协议的这种格式，缺点也是很明显的：

1. 额外占用了很多字节，比如众多的回车符、换行符，它们都是字符，都需要一个字节
2. 大头儿子，通常一个HTTP1的请求，都会携带各种请求头，我们可以通过请求头来指定**请求体的压缩方式**，但是我们没有地方可以指定请求头的压缩方式，这样就导致了大头儿子的存在

为了解决这两个严重影响性能的问题，HTTP2出来了，你不就是要发送请求行、请求头、请求体吗，那HTTP2这么来设计，HTTP2设计了帧的概念，比如分为：



1. 帧长度，用三个字节来存一个数字，这个数字表示当前帧的实际传输的数据的大小，3个字节表示的最大数字是2的24次方（16M），所以一个帧最大为9字节+16M。
2. 帧类型，占一个字节，可以分为数据帧和控制帧
    a. 数据帧又分为：HEADERS 帧和 DATA 帧，用来传输请求头、请求体的
    b. 控制帧又分为：SETTINGS、PING、PRIORITY，用来进行管理的
1. 标志位，占一个字节，可以用来表示当前帧是整个请求里的最后一帧，方便服务端解析
2. 流标识符，占4个字节，在Java中也就是一个int，不过最高位保留不用，表示Stream ID，这也是HTTP2的一个重要设计
3. 实际传输的数据Payload，如果帧类型是HEADERS，那么这里存的就是请求头，如果帧类型是DATA ，那么这里存的就是请求体

通过这种设计，我们可以发现，我们就可以来压缩请求头了，比如如果帧的类型是HEADERS ，那就进行压缩，当然压缩算法是固定的HPACK算法，不能更换。

如果仅仅是新增了能压缩请求头，那还不至于说HTTP2性能高，HTTP2最优秀的地方在于支持Stream。

上面可以看到每个帧里有一个流标识符，表示Stream ID，这是HTTP2的新特性，表示一个“虚拟流”，达到的效果是，我们可以在一个TCP连接中，同时维护多个Stream，每一个帧都是属于某一个Stream。

也就是说客户端在一个TCP连接上，可以并发的给服务端同时发送多个帧，比如同时发送多个HEADERS帧（请求头），多个DATA 帧（请求体），这是HTTP1不支持，对于HTTP1而言，如果你先发了一个HTTP1请求，还没有收到响应结果的情况下，你又发了一个HTTP1请求，然后你收到了一个响应结果，此时你怎么知道这个响应结果对应的到底是哪一个HTTP1请求呢？你可能会想到在请求头和响应头中做一个标记，但这是属于HTTP1的扩展，HTTP1原生是不支持的，所以HTTP2就做了这个扩展，也就是Stream ID。

比如，现在有一个客户端，想向服务端发送三个请求，如果你只建了一个Stream，那么你仍然只能：发送请求1-->接收响应1-->发送请求2-->接收响应2-->发送请求2-->接收响应2。

但是如果你建了三个Stream，那么你就可以开三个线程，同时把这三个请求分别发送到不同的Stream中去，这样服务端那么也会分别从三个Stream中获取到不同的请求进行处理，然后把响应结果也发送到对应的Stream中被客户端接收到，这样就极大的提高了并发。

所以我们在利用HTTP2发送一个请求时，首先：

1. 新建一个TCP连接（三次握手）
2. 新建一个Stream，生成一个新的StreamID，生成一个控制帧，帧里记录了前面生成出来的StreamID，通过TCP连接发送出去
3. 生成一个要发送的请求对应的HEADERS 帧，用来发送请求头，也是key:value的格式，先利用ascii进行编码，然后利用HPACK算法进行压缩，最终把压缩之后的字节存在帧中的Payload区域，记录好StreamID，最后通过TCP连接把这个HEADERS 帧发送出去
4. 最后把要发送的请求体数据按指定的压缩算法(请求中所指定的压缩算法，比如gzip)进行压缩，把压缩之后的字节生成DATA 帧，记录好StreamID，通过TCP连接把DATA 帧发送出去。

对于服务端而言：

1. 会不断的从TCP连接接收到某些帧
2. 当接收到一个控制帧时，表示客户端要和服务端新建一个Stream，在服务端记录一下StreamID，比如在Dubbo3.0的源码中会生成一个ServerStreamObserver的对象
3. 当接收到一个HEADERS 帧，取出StreamID，找到对应的ServerStreamObserver对象，并解压得到请求头，把请求头信息保存在ServerStreamObserver对象中
4. 当接收到一个DATA 帧时，取出StreamID，找到对应的ServerStreamObserver对象，根据请求头的信息看如何解压请求体，解压之后就得到了原生了请求体数据，然后按业务逻辑处理请求体
5. 处理完了之后，就把结果也生成HEADERS 帧和DATA 帧时发送客户端，客户端此时就变成了服务

端，来处理响应结果。

6. 客户端接收到响应结果的HEADERS 帧，是也先解压得到响应头，记录响应体的解压方式
7. 然后继续接收到响应结果的DATA 帧，解压响应体，得到原生的响应体，处理响应体

对于Triple协议而言，我们主要理解HTTP2中的Stream、HEADERS 帧、DATA 帧就可以了。

其他关于HTTP2的就不在本节课来继续介绍了。

Triple的底层原理分析

就是因为HTTP2中的数据帧机制，Triple协议才能支持UNARY、SERVER_STREAM、BI_STREAM三种模式。

1. UNARY：就是最普通的，服务端只有在接收到完请求包括的所有的HEADERS帧和DATA帧之后(通过调用onCompleted()发送最后一个DATA帧)，才会处理数据，客户端也只有接收完响应包括的所有的HEADERS帧和DATA帧之后，才会处理响应结果。
2. SERVER_STREAM：服务端流，特殊的地方在于，服务端在接收完请求包括的所有的DATA帧之后，才会处理数据，不过在处理数据的过程中，可以多次发送响应DATA帧（第一个DATA帧发送之前会发送一个HEADERS帧），客户端每接收到一个响应DATA帧就可以直接处理该响应DATA 帧，这个模式下，客户端只能发一次数据，但**能多次处理响应DATA帧。**（目前有Bug，gRPC的效果是正确的，Dubbo3.0需要异步进行发送）
3. BI_STREAM：双端流，或者客户端流，特殊的地方在于，客户端**可以控制发送多个请求DATA帧**（第一个DATA帧发送之前会发送一个HEADERS帧），服务端会不断的接收到请求DATA帧并进行处理，并且及时的把处理结果作为响应DATA帧发送给客户端（第一个DATA帧发送之前会发送一个HEADERS帧），而客户端每接收到一个响应结果DATA帧也会直接处理，这种模式下，**客户端和服务端都在不断的接收和发送DATA帧并进行处理，注意请求HEADER帧和响应HEADERS帧都只发了一个。**

Triple请求调用和响应处理

创建一个Stream的前提是先得有一个Socket连接，所以我们得先知道Socket连接是在哪创建的。

在服务提供者进行服务导出时，会按照协议以及对应的端口启动Server，比如Triple协议就会启动Netty并绑定指定的端口，等待Socket连接，在进行服务消费者进行服务引入的过程中，会生成TripleInvoker对象，在构造TripleInvoker对象的构造方法中，会利用ConnectionManager创建一个Connection对象，而Connection对象中包含了一个Bootstrap对象（Netty中用来建立Socket连接的），不过以上都只是创建对象，并不会真正和服务去建立Socket连接，所以在生成TripleInvoker对象过程中不会真正去创建Socket连接，那什么时候创建的呢？

当我们在服务消费端执行以下代码时：

```
demoService.sayHello("zhouyu")
```

demoService是一个代理对象，在执行方法的过程中，最终会调用TripleInvoker的doInvoke()方法，在doInvoke()方法中，会利用Connection对象来判断Socket连接是否可用，如果不可用并且没有初始化，那就会创建Socket连接。

一个Connection对象就表示一个Socket连接，在TripleInvoker对象中也只有一个Connection对象，也就是一个TripleInvoker对象只对应一个Socket连接，这个和DubboInvoker不太一样，一个DubboInvoker中可以有多个ExchangeClient，每个ExchangeClient都会与服务端创建一个Socket连接，所以一个DubboInvoker可以对应多个Socket连接，当然多个Socket连接的目的就是提高并发，不过在TripleInvoker对象中就不需要这么来设计了，因为可以Stream机制来提高并发。

以上，我们知道了，当我们利用**服务接口的代理对象**执行方法时就会创建一个Socket连接，就算这个代理对象再次执行方法时也不会再次创建Socket连接了，值得注意的是，有可能两个服务接口对应的是一个Socket连接，举个例子。

比如服务提供者应用A，提供了DemoService和HelloService两个服务，服务消费者应用B引入了这两个服务，那么在服务消费者这端，这个两个接口对应的代理对象对应的TripleInvoker是不同的两个，但是这两个TripleInvoker会公用一个Socket连接，因为ConnectionManager在创建Connection对象时会根据服务URL的address进行缓存，后续这两个代理对象在执行方法时使用的就是同一个Socket连接，但是是不同的Stream。

Socket连接创建好之后，就需要发送Invocation对象给服务提供者了，因为是基于的HTTP2，所以要先创建一个Stream，然后再通过Stream来发送数据。

TripleInvoker中用的是Netty，所以最终会利用Netty来创建Stream，对应的对象为**Http2StreamChannel**，消费端的TripleInvoker最终会利用Http2StreamChannel来发送和接收数据帧，数据帧对应的对象为**Http2Frame**，它又分为Http2DataFrame、Http2HeadersFrame等具体类型。

正常情况下，会每生成一个数据帧就会通过**Http2StreamChannel**发送出去，但是在Triple中有一个小小的优化，会有一个批量发送的思想，当要发送一个数据帧时，会先把数据帧放入一个WriteQueue中，然后会从线程池中拿到一个线程调用WriteQueue的flush方法，该方法的实现为：

```
private void flush() {
        try {
                QueuedCommand cmd;
```

```
                int i = 0;
                boolean flushedOnce = false;

                // 只要队列中有元素就取出来，没有则退出while
                while ((cmd = queue.poll()) != null) {
                        // 把数据帧添加到Http2StreamChannel中，添加并不会立马发送，
                        // 调用了channel.flush()才发送
                        cmd.run(channel);
                        i++;

                        // DEQUE_CHUNK_SIZE=128
                        // 连续从队列中取到了128个数据帧就flush一次
                        if (i == DEQUE_CHUNK_SIZE) {
                                i = 0;
                                channel.flush();
                                flushedOnce = true;
                        }
                }

                // i != 0 表示从队列中取到了数据但是没满128个
                // 如果i=0，flushedOnce=false也flush一次
                if (i != 0 || !flushedOnce) {
                        channel.flush();
                }
        } finally {
                // cas的标记
                scheduled.set(false);

                // 如果队列中又有数据了，则继续继续从线程池获取一个线程调用当前flush方法
                if (!queue.isEmpty()) {
                        scheduleFlush();
                }
        }
}
```

总体思想是，只要向WriteQueue中添加一个数据帧之后，那就会尝试开启一个线程，要不要开启线程要看CAS，比如现在有10个线程同时向WriteQueue中添加了一个数据帧，那么这10个线程中的某一个会CAS成功，其他会CAS失败，那么此时CAS成功的线程会负责从线程池中获取另外一个线程执行上面的flush方法，从而获取WriteQueue中的数据帧然后发送出去。

有了底层这套设计之后，对于TripleInvoker而言，它只需要把要发送的数据封装为数据帧，然后添加到WriteQueue中就可以了。

在TripleInvoker的doInvoke()源码中，在创建完成Socket连接后，就会：

1. 基于Socket连接先构造一个ClientCall对象
2. 根据当前调用的方法信息构造一个RequestMetadata对象，这个对象表示，当前调用的是哪个接口的哪个方法，并且记录了所配置的序列化方式，压缩方式，超时时间等
3. 紧接着构造一个ClientCall.Listener，这个Listener是用来处理响应结果的，针对不同的流式调用类型，会构造出不同的ClientCall.Listener：
    a. UNARY：会构造出一个UnaryClientCallListener，内部包含了一个DeadlineFuture，DeadlineFuture是用来控制timeout的
    b. SERVER_STREAM：会构造出一个ObserverToClientCallListenerAdapter，内部包含了调用方法时传入进来的StreamObserver对象，最终就是由这个StreamObserver对象来处理响应结果的
    c. BI_STREAM：和SERVER_STREAM一样，也会构造出来一个ObserverToClientCallListenerAdapter
1. 紧着着，就会调用ClientCall对象的start方法创建一个Stream，并且返回一个StreamObserver对象
2. 得到了StreamObserver对象后，会根据不同的流式调用类型来使用这个StreamObserver对象
    a. UNARY：直接调用StreamObserver对象的onNext()方法来发送方法参数，然后调用onCompleted方法，然后返回一个

new AsyncRpcResult(future, invocation)，future就是DeadlineFuture，后续会通过DeadlineFuture同步等待响应结果的到来，并最终把获取到的响应结果返回给业务方法。

    b. SERVER_STREAM：直接调用StreamObserver对象的onNext()方法来发送方法参数，然后调用onCompleted方法，然后返回一个

new AsyncRpcResult(CompletableFuture.completedFuture(new AppResponse()), invocation)，后续不会同步了，并且返回null给业务方法。

    c. BI_STREAM：直接返回

new AsyncRpcResult(CompletableFuture.completedFuture(new AppResponse(requestObserver)), invocation)，也不同同步等待响应结果了，而是直接把requestObserver对象返回给了业务方法。

所以我们可以发现，不管是哪种流式调用类型，都会先创建一个Stream，得到对应的一个StreamObserver对象，然后调用StreamObserver对象的onNext方法来发送数据，比如发送服务接口方法的入参值，比如一个User对象：

1. 在发送User对象之前，会**先发送请求头**，请求头中包含了当前调用的是**哪个接口、哪个方法、版本号、序列化方式、压缩方式**等信息，注意请求头中会包含一些gRPC相关的key，主要就是为了兼容gRPC
2. 然后就是发送请求体
3. 然后再对User对象进行序列化，得到字节数组

4. 然后再压缩字节数组
5. 然后把压缩之后的字节数组以及是否压缩标记生成一个DataQueueCommand对象，并且把这个对象添加到writeQueue中去，然后执行scheduleFlush()，该方法就会开启一个线程从writeQueue中获取数据进行发送，发送时就会触发DataQueueCommand对象的doSend方法进行发送，该方法中会构造一个DefaultHttp2DataFrame对象，该对象中由两个属性endStream，表示是不是Stream中的最后一帧，另外一个属性为content，表示帧携带的核心数据，该数据格式为：
    a. 第一个字节记录请求体是否被压缩
    b. 紧着的四个字节记录字节数组的长度
    c. 后面就真正的字节数据

以上是TripleInvoker发送数据的流程，接下来就是TripleInvoker接收响应数据的流程，ClientCall.Listener就是用来监听是否接收到的响应数据的，不同的流式调用方式会对应不同的ClientCall.Listener：

a. UNARY：UnaryClientCallListener，内部包含了一个DeadlineFuture，DeadlineFuture是用来控制timeout的
b. SERVER_STREAM：ObserverToClientCallListenerAdapter，内部包含了调用方法时传入进来的StreamObserver对象，最终就是由这个StreamObserver对象来处理响应结果的
c. BI_STREAM：和SERVER_STREAM一样，也会构造出来一个ObserverToClientCallListenerAdapter

那现在要了解的就是，如何知道某个Stream中有响应数据，然后触发调用ClientCall.Listener对象的相应的方法。

要监听某个Stream中是否有响应数据，这个肯定时Netty来做的，实际上，在之前创建Stream时，会向Http2StreamChannel绑定一个TripleHttp2ClientResponseHandler，很明显这个Handler就是用来处理接收到的响应数据的。

在TripleHttp2ClientResponseHandler的channelRead0方法中，每接收一个响应数据就会判断是Http2HeadersFrame还是Http2DataFrame，然后调用ClientTransportListener中对应的onHeader方法和onData方法：

1. onHeader方法通过处理响应头，会生成一个TriDecoder，它是用来解压并处理响应体的
2. onData方法会利用TriDecoder的deframe()方法来处理响应体

另外如果服务提供者那边调用了onCompleted方法，会向客户端响应一个请求头，endStream为true，表示响应结束，也会触发执行onHeader方法，从而会调用TriDecoder的close()方法.

TriDecoder的deframe()方法在处理响应体数据时，会分为两个步骤：

1. 先解析前5个字节，先解析第1个字节确定该响应体是否压缩了，再解析后续4个字节确定响应体内容的字节长度
2. 然后再取出该长度的字节作为响应体数据，如果压缩了，那就进行解压，然后把解压之后的字节数组传递给

ClientStreamListenerImpl的onMessage()方法，该方法就会按对应的序列化方式进行反序列化，得到最终的对象，然后再调用到最终的UnaryClientCallListener或者ObserverToClientCallListenerAdapter的onMessage()方法。

TriDecoder的close()方法最终也会调用到UnaryClientCallListener或者ObserverToClientCallListenerAdapter的close()方法。

UnaryClientCallListener，构造它时传递了一个DeadlineFuture对象：

1. onMessage()接收到响应结果对象后，会把结果对象赋值给appResponse属性
2. onClose()会取出appResponse属性记录的结果对象构造出来一个AppResponse对象，然后调用DeadlineFuture的received方法，从而将方法调用线程接阻塞，并得到响应结果对象。

ObserverToClientCallListenerAdapter，构造它时传递了一个StreamObserver对象：

1. onMessage()接收到响应结果对象后，会调用StreamObserver对象的onNext()，并把结果对象传给onNext()方法，从触发了程序员的onNext()方法逻辑。
2. onClose()就会调用StreamObserver对象的onCompleted()，或者调用onError()方法

Triple请求处理和响应结果发送

其实这部分内容和发送请求和处理响应是非常类似的，无非就是把视角从消费端切换到服务端，前面分析的是消费端发送和接收数据，现在要分析的是服务端接收和发送数据。

消费端在创建一个Stream后，会生成一个对应的StreamObserver对象用来发送数据和一个ClientCall.Listener用来接收响应数据，对于服务端其实也一样，在接收到消费端创建Stream的命令后，也需要生成一个对应的StreamObserver对象用来响应数据以及一个ServerCall.Listener用来接收请求数据。

在服务导出时，TripleProtocol的export方法中会开启一个ServerBootstrap，并绑定指定的端口，并且最重要的是，Netty会负责接收创建Stream的信息，一旦就收到这个信号，就会生成一个ChannelPipeline，并给ChannelPipeline绑定一个TripleHttp2FrameServerHandler，而这个TripleHttp2FrameServerHandler就可以用来处理Http2HeadersFrame和Http2DataFrame。

比如在接收到请求头后，会构造一个ServerStream对象，该对象有一个ServerTransportObserver对象，ServerTransportObserver对象就会真正来处理请求头和请求体：

1. onHeader()方法，用来处理请求头
    a. 比如从请求头中得到当前请求调用的是哪个服务接口，哪个方法

b. 构造一个TriDecoder对象，TriDecoder对象用来处理请求体
c. 构造一个ReflectionServerCall对象并调用它的doStartCall()方法，从而生成不同的ServerCall.Listener
    i. UNARY：UnaryServerCallListener
    ii. SERVER_STREAM：ServerStreamServerCallListener
    iii. BI_STREAM：BiStreamServerCallListener
    iv. 并且在构造这些ServerCall.Listener时会把ReflectionServerCall对象传入进去，ReflectionServerCall对象可以用来向客户端发送数据

1. onData()方法，用来处理请求体，调用TriDecoder对象的deframe方法来处理请求体，如果是endStream，那还会调用TriDecoder对象的close方法

TriDecoder：

1. deframe()：这个方法的作用和客户端时一样的，都是先解析请求体的前5个字节，然后解压请全体，然后反序列化得到请求参数对象，然后调用不同的ServerCall.Listener中的onMessage()
2. close()：当客户端调用onCompleted方法时，就表示发送数据完毕，此时会发送一个DefaultHttp2DataFrame并且endStream为true，从而会触发服务端TriDecoder对象的close()方法，从而调用不同的ServerCall.Listener中的onComplete()

UnaryServerCallListener：

1. 在接收到请求头时，会构造UnaryServerCallListener对象，没什么特殊的
2. 然后接收到请求体时，请求体中的数据就是调用接口方法的入参值，比如User对象，那么就会调用UnaryServerCallListener的**onMessage()**方法，在这个方法中会把User对象设置到invocation对象中
3. 当消费端调用onCompleted()方法，表示请求体数据发送完毕，从而触发UnaryServerCallListener**的onComplete()**方法，在该方法中会调用**invoke()**方法，从而执行服务方法，并得到结果，得到结果后，会调用UnaryServerCallListener的**onReturn()**方法，把结果通过responseObserver发送给消费端，并调用responseObserver的onCompleted()方法，表示响应数据发送完毕，responseObserver是ReflectionServerCall对象的一个StreamObserver适配对象（ServerCallToObserverAdapter）。

再来看ServerStreamServerCallListener：

1. 在接收到请求头时，会构造ServerStreamServerCallListener对象，没什么特殊的
2. 然后接收到请求体时，请求体中的数据就是调用接口方法的入参值，比如User对象，那么就会调用ServerStreamServerCallListener的**onMessage()**方法，在这个方法中会把**User对象以及responseObserver对象**设置到invocation对象中，这是和UnaryServerCallListener不同的地方，UnaryServerCallListener只会把User对象设置给invocation，而ServerStreamServerCallListener还会把**responseObserver对象**设置进去，因为服务端流需要这个responseObserver对象，服务方法拿到这个对象后就可以自己来控制如何发送响应体，并什么时候调用onCompleted()方法来表示响应体发送完毕。
3. 当消费端调用onCompleted()方法，表示请求体数据发送完毕，从而触发ServerStreamServerCallListener**的**

onComplete()方法，在该方法中会调用**invoke()**方法，从而执行服务方法，从而会通过responseObserver对象来发送数据

4. 方法执行完后，仍然会调用ServerStreamServerCallListener的**onReturn()**方法，但是个空方法

再来看最后一个BiStreamServerCallListener：

1. 在接收到请求头时，会构造BiStreamServerCallListener对象，这里比较特殊，会把responseObserver设置给invocation并执行invoke()方法，从而执行服务方法，并执行onReturn()方法，onReturn()方法中会把服务方法的执行结果，也是一个StreamObserver对象，赋值给BiStreamServerCallListener对象的**requestObserver属性。**
2. 这样，在接收到请求头时，服务方法就会执行了，并且得到了一个**requestObserver**，**它是程序员定义的，是用来处理请求体的**，另外的responseObserver是用来发送响应体的。
3. 紧接着就会收到请求体，从而触发onMessage()方法，该方法中会调用requestObserver的onNext()方法，这样就可以做到，服务端能实时的接收到消费端每次所发送过来的数据，并且进行处理，处理过程中，如果需要响应就可以利用responseObserver进行响应
4. 一旦消费端那边调用了onCompleted()方法，那么就会触发BiStreamServerCallListener的onComplete方法，该方法中也就是调用requestObserver的onCompleted()，主要就触发程序员自己写的StreamObserver对象中的onCompleted()，并没有针对底层的Stream做什么事情。

总结

不管是Unary，还是ServerStream，还是BiStream，底层客户端和服务端之前都只有一个Stream，它们三者的区别在于：

1. Unary：通过流，将方法入参值作为请求体发送出去，而且只发送一次，服务端这边接收到请求体之后，会执行服务方法，得到结果，把结果返回给客户端，也只响应一次。
2. ServerStream：通过流，将方法入参值作为请求体发送出去，而且只发送一次，服务端这边接收到请求体之后，会执行服务方法，并且会把当前流对应的StreamObserver对象也传给服务方法，由服务方法自己控制如何响应，响应几次，响应什么数据，什么时候响应结束，都由服务方法自己控制。
3. BiStream，通过流，客户端和服务端，都可以发送和响应多次。