consumer端的数据经过处理后，最终进入发送的流程。接下来我们继续跟着数据的流向进行分析。 首先进入到了DubboInvoker，DubboInvoker中包含了多个ExchangeClient， 每个ExchangeClient都对应了一个物理连接，同一个DubboInvoker中的所有ExchangeClient都是连接的同一个ip/port。DubboInvoker循环的从ExchangeClient数组中获取一个，并利用该ExchangeClient发送数据，发送的模式有三种：

1、单项发送：发送完数据直接返回，不需要结果；

2、双向发送：发送完数据后等待数据返回（类似Future.get()）；

3、异步发送：发送完数据直接返回，同时往RpcContext中存入对应的Future，应用可以通过RpcContext.getContext().getFuture()获取到Future。通过Future可以发起多个异步调用，减少业务的执行时间。

[java] view plain copy

```
1. protected Result doInvoke(final Invocation invocation) throws
Throwable {
2.     RpcInvocation inv = (RpcInvocation) invocation;
3.     final String methodName =
RpcUtils.getMethodName(invocation);
4.     inv.setAttachment(Constants.PATH_KEY, getUrl().getPath());
5.     inv.setAttachment(Constants.VERSION_KEY, version);
6.
7.     // 如果有多个连接则轮流发
8.     ExchangeClient currentClient;
9.     if (clients.length == 1) {
10.         currentClient = clients[0];
11.     } else {
12.         currentClient = clients[index.getAndIncrement() %
clients.length];
13.     }
14.     try {
15.         boolean isAsync = RpcUtils.isAsync(getUrl(),
invocation);
16.         boolean isOneway = RpcUtils.isOneway(getUrl(),
invocation);
17.         int timeout = getUrl().getMethodParameter(methodName,
Constants.TIMEOUT_KEY,Constants.DEFAULT_TIMEOUT);
18.         if (isOneway) {
```

```
19.                 // 不需要返回则发送后不等待立即返回
20.                 boolean isSent =
getUrl().getMethodParameter(methodName, Constants.SENT_KEY,
false);
21.                 currentClient.send(inv, isSent);
22.                 RpcContext.getContext().setFuture(null);
23.                 return new RpcResult();
24.             } else if (isAsync) {
25.                 // 异步返回时将Future设置到RpcContext中供业务去获取，由
业务自行处理异步后的逻辑
26.                 ResponseFuture future =
currentClient.request(inv, timeout) ;
27.                 RpcContext.getContext().setFuture(new
FutureAdapter<Object>(future));
28.                 return new RpcResult();
29.             } else {
30.                 // 通过future.get()阻塞等待结果返回
31.                 RpcContext.getContext().setFuture(null);
32.                 return (Result) currentClient.request(inv,
timeout).get();
33.             }
34.         } catch (TimeoutException e) {
35.             throw new
RpcException(RpcException.TIMEOUT_EXCEPTION, "Invoke remote
method timeout. method: " + invocation.getMethodName() + ",
provider: " + getUrl() + ", cause: " + e.getMessage(), e);
36.         } catch (RemotingException e) {
37.             throw new
RpcException(RpcException.NETWORK_EXCEPTION, "Failed to invoke
remote method: " + invocation.getMethodName() + ", provider: " +
getUrl() + ", cause: " + e.getMessage(), e);
38.         }
39. }
```

除了异步转同步的功能外，HeaderExchangeClient还加入了心跳检测的功能：

[java] view plain copy

```
1. private void startHeatbeatTimer() {
2.     // 停止之前的心跳任务
3.     stopHeartbeatTimer();
```

```
4.     if ( heartbeat > 0 ) {
5.         // 创建定时任务，默认心跳间隔为60s
6.         heatbeatTimer = scheduled.scheduleWithFixedDelay(
7.                 new HeartBeatTask( new
HeartBeatTask.ChannelProvider() {
8.                     public Collection<Channel> getChannels() {
9.                         return Collections.
<Channel>singletonList( HeaderExchangeClient.this );
10.                     }
11.                 }, heartbeat, heartbeatTimeout),
12.                 heartbeat, heartbeat, TimeUnit.MILLISECONDS
);
13.     }
14. }
```

定时任务的逻辑比较简单： 获取连接最后一次读/写数据的时间，如果读或写的时间距当前时间超过心跳的时间，则主动发起一个心跳包，如果该心跳被收到并回复则对应的最后一次读数据时间也会更新，表示连接正常；

HeaderExchangeChannel：将发送的数据封装为Request对象， 产生一个Future对象（用户异步转同步）与Request关联，然后调用更底层的Channel发送Request。 需要注意的是每一个Request对象都对应了一个唯一id(id为int类型，因此当id达到最大后，又会变为最小值，这样重复利用id)。该id代表了当前连接，在provider有返回数据的时候，会根据这个id来查找对应的Channel。

底层的Channel根据配置不同而不同，默认情况下使用的是netty，consumer端对应实现为NettyClient。netty本身的实现比较高效也很复杂，这里不详讲，有兴趣的同学可以关注本博客内netty相关的文章。这里只关注序列化的部分，具体实现在ExchangeCodec中，以request的encode为例：

[java] view plain copy

```
1. protected void encodeRequest(Channel channel, ChannelBuffer
buffer, Request req) throws IOException {
2.     // 加载序列化实现
3.     Serialization serialization = getSerialization(channel);
4.     // header.
5.     byte[] header = new byte[HEADER_LENGTH];
```

```
6.      // set magic number.
7.      Bytes.short2bytes(MAGIC, header);
8.
9.      // set request and serialization flag.
10.     header[2] = (byte) (FLAG_REQUEST |
serialization.getContentTypeId());
11.
12.     if (req.isTwoWay()) header[2] |= FLAG_TWOWAY;
13.     if (req.isEvent()) header[2] |= FLAG_EVENT;
14.
15.     // set request id.
16.     Bytes.long2bytes(req.getId(), header, 4);
17.
18.     // encode request data.
19.     int savedWriteIndex = buffer.writerIndex();
20.     buffer.writerIndex(savedWriteIndex + HEADER_LENGTH);
21.     ChannelBufferOutputStream bos = new
ChannelBufferOutputStream(buffer);
22.     ObjectOutput out =
serialization.serialize(channel.getUrl(), bos);
23.     if (req.isEvent()) {
24.         encodeEventData(channel, out, req.getData());
25.     } else {
26.         encodeRequestData(channel, out, req.getData());
27.     }
28.     out.flushBuffer();
29.     bos.flush();
30.     bos.close();
31.     int len = bos.writtenBytes();
32.     checkPayload(channel, len);
33.     Bytes.int2bytes(len, header, 12);
34.
35.     // write
36.     buffer.writerIndex(savedWriteIndex);
37.     buffer.writeBytes(header); // write header.
38.     buffer.writerIndex(savedWriteIndex + HEADER_LENGTH +
len);
39. }
```

可以看到一个数据包含header和body， header固定16个字节，前2字节为magic number, 第3字节包括类型（类型+序列化实现的id），第4字节在response时为status,request时为0, 第5-12字节为请求id,13-16字节为body字节数。注意每种序列化都有对应的id，如果要新增序列化方式一定不能与现有id重复。上面的代码中包含一个小细节，一开始并不知道body

的字节数，只有序列化完成后才知道，因此header是最后写入的（writerIndex（savedWriteIndex）方法相当于将index移到起点，写入header后再将index移到最后）。每一个请求分配唯一id，到最大后循环使用， 循环利用时之前id对应的请求早就已经不再了，因此还是唯一的。

数据返回后的decode方法也是在ExchangeCodec中，具体代码这里不贴了。 response返回时id与request一致，这样可以从consumer缓存map中根据id取出Future并往里设置数据，数据设置完成后，之前在future.get()阻塞的地方恢复（见DubboInvoker的doInvoke方法），继续执行后续逻辑。最终层层返回到业务代码中。