

从前面一篇创建注册中心的流程当中，我们知道在从注册中心获取到provider的连接信息后，会通过连接创建Invoker。代码见com.alibaba.dubbo.registry.integration.RegistryDirectory的toInvokers方法：

[java] [view plain copy](#)

C

8

```
1. // protocol实现为com.alibaba.dubbo.rpc.Protocol$Adaptive,
2. // 之前已经讲过，这是dubbo在运行时动态创建的一个类；
3. // serviceType为服务类的class，如demo中的
   com.alibaba.dubbo.demo.DemoService;
4. // providerUrl为服务提供方注册的连接；
5. // url为providerUrl与消费方参数的合并
6. invoker = new InvokerDelegete<T>(protocol.refer(serviceType,
   url), url, providerUrl);
```

此处url的protocol为dubbo，因此protocol.refer最终会调用com.alibaba.dubbo.rpc.protocol.dubbo.DubboProtocol.refer，同时Protocol存在两个wrapper类，分别为：com.alibaba.dubbo.rpc.protocol.ProtocolListenerWrapper、com.alibaba.dubbo.rpc.protocol.ProtocolFilterWrapper。在dubbo中存在wrapper类的类会被wrapper实例包装后返回，因此在protocol.refer方法调用的时候，会先经过wrapper类。由于这里的复杂性，我们先不讲wrapper类里的refer实现，直接跳到DubboProtocol.refer。

url的demo如下：

[plain] [view plain copy](#)

C

8

```
1. dubbo://30.33.47.127:20880/com.alibaba.dubbo.demo.DemoService?
   anyhost=true&application=demo-consumer&check=false&....
```

DubboProtocol的refer代码如下：

[java] [view plain copy](#)

C

8

```
1. public <T> Invoker<T> refer(Class<T> serviceType, URL url)
   throws RpcException {
```

```

2. // 创建一个DubboInvoker
3. DubboInvoker<T> invoker = new DubboInvoker<T>(serviceType,
url, getClients(url), invokers);
4. // 将invoker加入到invokers这个Set中
5. invokers.add(invoker);
6. return invoker;
7. }
8.
9. // 创建连接Client, 该Client主要负责建立连接, 发送数据等
10. private ExchangeClient[] getClients(URL url){
11. //是否共享连接
12. boolean service_share_connect = false;
13. int connections =
url.getParameter(Constants.CONNECTIONS_KEY, 0);
14. // 如果connections不配置, 则共享连接, 否则每服务每连接,
15. // 共享连接的意思是对于同一个ip+port的所有服务只创建一个连接,
16. // 如果是非共享连接则每个服务+(ip+port)创建一个连接
17. if (connections == 0){
18. service_share_connect = true;
19. connections = 1;
20. }
21.
22. ExchangeClient[] clients = new
ExchangeClient[connections];
23. for (int i = 0; i < clients.length; i++) {
24. if (service_share_connect){
25. clients[i] = getSharedClient(url);
26. } else {
27. clients[i] = initClient(url);
28. }
29. }
30. return clients;
31. }
32.
33. /**
34. *获取共享连接
35. */
36. private ExchangeClient getSharedClient(URL url){
37. // 以address(ip:port)为key进行缓存
38. String key = url.getAddress();
39. ReferenceCountExchangeClient client =
referenceClientMap.get(key);
40. if ( client != null ){
41. // 如果连接存在了则引用数加1, 引用数表示有多少个服务使用了此
client,

```

```

42.         // 当某个client调用close()时, 引用数减一,
43.         // 如果引用数大于0, 表示还有服务在使用此连接, 不会真正关闭
client
44.         // 如果引用数为0, 表示没有服务在用此连接, 此时连接彻底关闭
45.         if ( !client.isClosed()){
46.             client.incrementAndGetCount();
47.             return client;
48.         } else {
49.             logger.warn(new IllegalStateException("client
is closed,but stay in clientmap .client :"+ client));
50.             referenceClientMap.remove(key);
51.         }
52.     }
53.     // 调用initClient来初始化Client
54.     ExchangeClient exchagneclient = initClient(url);
55.     // 使用ReferenceCountExchangeClient进行包装
56.     client = new ReferenceCountExchangeClient(exchagneclient,
ghostClientMap);
57.     referenceClientMap.put(key, client);
58.     ghostClientMap.remove(key);
59.     return client;
60. }
61.
62. /**
63.  * 创建新连接.
64.  */
65. private ExchangeClient initClient(URL url) {
66.     // 获取client参数的值, 为空则获取server参数的值, 默认为netty
67.     String str = url.getParameter(Constants.CLIENT_KEY,
url.getParameter(Constants.SERVER_KEY,
Constants.DEFAULT_REMOTING_CLIENT));
68.
69.     String version =
url.getParameter(Constants.DUBBO_VERSION_KEY);
70.     // 如果是1.0.x版本, 需要兼容
71.     boolean compatible = (version != null &&
version.startsWith("1.0."));
72.     // 加入codec参数, 默认为dubbo, 即DubboCodec
73.     url = url.addParameter(Constants.CODEC_KEY,
Version.isCompatibleVersion() && compatible ?
COMPATIBLE_CODEC_NAME : DubboCodec.NAME);
74.     //默认开启心跳, 默认每60s发送一次心跳包
75.     url = url.addParameterIfAbsent(Constants.HEARTBEAT_KEY,
String.valueOf(Constants.DEFAULT_HEARTBEAT));
76.

```

```

77.      // BIO存在严重性能问题, 暂时不允许使用
78.      if (str != null && str.length() > 0 && !
ExtensionLoader.getExtensionLoader(Transporter.class).hasExtension
n(str)) {
79.          throw new RpcException("Unsupported client type: " +
str + ", " +
80.              " supported client type is " +
StringUtils.join(ExtensionLoader.getExtensionLoader(Transporter.c
lass).getSupportedExtensions(), " "));
81.      }
82.
83.      ExchangeClient client ;
84.      try {
85.          //设置连接应该是lazy的
86.          if (url.getParameter(Constants.LAZY_CONNECT_KEY,
false)){
87.              client = new LazyConnectExchangeClient(url
,requestHandler);
88.          } else {
89.              client = Exchangers.connect(url ,requestHandler);
90.          }
91.      } catch (RemotingException e) {
92.          throw new RpcException("Fail to create remoting
client for service(" + url
93.              + "): " + e.getMessage(), e);
94.      }
95.      return client;
96. }

```

可以看到client创建由

com.alibaba.dubbo.remoting.exchange.Exchangers处理, 其代码如下:

[java] [view plain copy](#)



```

1. public static ExchangeClient connect(URL url, ExchangeHandler
handler) throws RemotingException {
2.     if (url == null) {
3.         throw new IllegalArgumentException("url == null");
4.     }
5.     if (handler == null) {
6.         throw new IllegalArgumentException("handler == null");
7.     }
8.     url = url.addParameterIfAbsent(Constants.CODEC_KEY,
"exchange");

```

```

9.      // 默认通过HeaderExchanger.connect创建
10.     return getExchanger(url).connect(url, handler);
11. }
12.
13. public static Exchanger getExchanger(URL url) {
14.     // 默认type为header,因此默认的Exchanger为
    com.alibaba.dubbo.remoting.exchange.support.header.HeaderExchange
    r
15.     String type = url.getParameter(Constants.EXCHANGER_KEY,
    Constants.DEFAULT_EXCHANGER);
16.     return getExchanger(type);
17. }
18.
19. public static Exchanger getExchanger(String type) {
20.     return
    ExtensionLoader.getExtensionLoader(Exchanger.class).getExtension(
    type);
21. }

```

HeaderExchanger的connect代码如下：

[java] [view plain copy](#)



```

1. public ExchangeClient connect(URL url, ExchangeHandler
    handler) throws RemotingException {
2.     return new HeaderExchangeClient(Transporters.connect(url,
    new DecodeHandler(new HeaderExchangeHandler(handler))));
3. }

```

这里简单介绍下这些类的作用：

HeaderExchangeHandler: ExchangeHandler的代理，HeaderExchangeHandler将数据封装后调用ExchangeHandler的连接/断开/发送请求/接收返回数据/捕获异常等方法；

DecodeHandler: 也是一个代理，在HeaderExchangeHandler的功能之上加入了解码功能；

Transporters.connect默认得到的是NettyTransporter：创建NettyClient, 该client是真正的发起通讯的类；

NettyClient在初始化的时候会做一些比较重要的事情，我们先看下：

[java] [view plain copy](#)





```

1. public NettyClient(final URL url, final ChannelHandler
handler) throws RemotingException {
2.     super(url, wrapChannelHandler(url, handler));
3. }
4.
5. protected static ChannelHandler wrapChannelHandler(URL url,
ChannelHandler handler){
6.     // 设置threadName, 设置默认的threadpool类型,
7.     //
8.     url = ExecutorUtil.setThreadName(url,
CLIENT_THREAD_POOL_NAME);
9.     url = url.addParameterIfAbsent(Constants.THREADPOOL_KEY,
Constants.DEFAULT_CLIENT_THREADPOOL);
10.    // 对handler再次进行包装
11.    return ChannelHandlers.wrap(handler, url);
12. }

```

我们知道前面得到的包装对象DecodeHandler，而

ChannelHandlers.wrap对该Handler再次进行包装：

[java] [view plain copy](#)



```

1. protected ChannelHandler wrapInternal(ChannelHandler handler,
URL url) {
2.     return new MultiMessageHandler(new
HeartbeatHandler(ExtensionLoader.getExtensionLoader(Dispatcher.cl
ass)
3.
.getAdaptiveExtension().dispatch(handler, url)));
4. }

```

这些包装类在之前handler的基础上加入的功能：

dispatch生成的对象AllChannelHandler：加入线程池，所有方法都异步的调用；

HeartbeatHandler: 心跳包的发送和接收到心跳包后的处理；

MultiMessageHandler：如果接收到的消息为MultiMessage，则将其拆分为单个Message给后面的Handler处理；

再看看NettyClient在构造方法中还做了哪些操作：

[java] [view plain copy](#)





1. // 调用了父类com.alibaba.dubbo.remoting.transport.AbstractClient的构造方法

```
2. public AbstractClient(URL url, ChannelHandler handler) throws
RemotingException {
3.     ...省略部分代码...
4.     try {
5.         //
6.         doOpen();
7.     } catch (Throwable t) {
8.         close();
9.         throw new RemotingException(url.toInetSocketAddress(),
null,
10.             "Failed to start " +
getClass().getSimpleName() + " " + NetUtils.getLocalAddress()
11.             + " connect to the server
" + getRemoteAddress() + ", cause: " + t.getMessage(), t);
12.     }
13.     try {
14.         // connect.
15.         connect();
16.         if (logger.isInfoEnabled()) {
17.             logger.info("Start " + getClass().getSimpleName()
+ " " + NetUtils.getLocalAddress() + " connect to the server " +
getRemoteAddress());
18.         }
19.     } catch (RemotingException t) {
20.         if (url.getParameter(Constants.CHECK_KEY, true)) {
21.             close();
22.             throw t;
23.         } else {
24.             // 如果check为false, 则连接失败时Invoker依然可以创建
25.             logger.warn("Failed to start " +
getClass().getSimpleName() + " " + NetUtils.getLocalAddress()
26.             + " connect to the server " +
getRemoteAddress() + " (check == false, ignore and retry later!),
cause: " + t.getMessage(), t);
27.         }
28.     } catch (Throwable t) {
29.         close();
30.         throw new
RemotingException(url.toInetSocketAddress(), null,
31.             "Failed to start " +
getClass().getSimpleName() + " " + NetUtils.getLocalAddress()
32.             + " connect to the server " +
```

```

getRemoteAddress() + ", cause: " + t.getMessage(), t);
33.     }
34.
35.     ...省略部分代码...
36. }

```

可以看到在构造方法处已经开始创建连接，netty如何创建连接此处不再详细介绍，可以看看之前的netty介绍。需要注意的是连接失败的时候，如果check参数为false则Invoker依然可以创建，否则在初始化阶段会报异常。

回过头来看看HeaderExchangeClient，改类创建了一个发送心跳包的定时任务：

[java] [view plain copy](#)



```

1. public HeaderExchangeClient(Client client){
2.     if (client == null) {
3.         throw new IllegalArgumentException("client == null");
4.     }
5.     this.client = client;
6.     this.channel = new HeaderExchangeChannel(client);
7.     String dubbo =
client.getUrl().getParameter(Constants.DUBBO_VERSION_KEY);
8.     // 默认为60秒发一次心跳包，如果连续3个心跳包无响应则表示连接断开
9.     this.heartbeat = client.getUrl().getParameter(
Constants.HEARTBEAT_KEY, dubbo != null &&
dubbo.startsWith("1.0.") ? Constants.DEFAULT_HEARTBEAT : 0 );
10.    this.heartbeatTimeout = client.getUrl().getParameter(
Constants.HEARTBEAT_TIMEOUT_KEY, heartbeat * 3 );
11.    if ( heartbeatTimeout < heartbeat * 2 ) {
12.        throw new IllegalStateException( "heartbeatTimeout <
heartbeatInterval * 2" );
13.    }
14.    startHeartbeatTimer();
15. }
16.
17. private void startHeartbeatTimer() {
18.     stopHeartbeatTimer();
19.     if ( heartbeat > 0 ) {
20.         heartbeatTimer = scheduled.scheduleWithFixedDelay(
21.             new HeartBeatTask( new
HeartBeatTask.ChannelProvider() {
22.                 public Collection<Channel> getChannels()

```



```

{
23.         return Collections.
<Channel>singletonList( HeaderExchangeClient.this );
24.     }
25.     }, heartbeat, heartbeatTimeout),
26.     heartbeat, heartbeat, TimeUnit.MILLISECONDS
);
27. }
28. }

```

我们知道，在socket通讯时，数据发送方和接收方必须建立连接，而建立的连接是否可用，为了探测连接是否可用，可以通过发送简单的通讯包并看是否收到回包的方式，这就是心跳。如果没有心跳包，则很有可能连接的一方已经断开或者中间线路故障，双方都不知道这种情况。因此心跳包很有必要引入。心跳包的实现比较简单，这里简单介绍下，不再贴具体代码：通过拦截（代理）所有的发送/接收数据的方法，记录下最后一次read(接收数据)、write(发送数据)的时间，如果都大于心跳的时间阈值（如上面的60s)则发送一条数据给对方，该数据的格式不重要，只要有心跳的标识（即对方可以解析出这是一个心跳包）即可，对方接收到数据以后也会返回一个应答的包，如果发送方接收到回包，则最后一次read时间将会被充值为当前时间，表示连接未断开。如果发送方一直未收到回包，则指定时间（如上面的60s)后再次发送心跳包。如果多次(如上面的3次)发送均未收到回包（心跳超时），则判断连接已经断开。此时根据应用的需求断开连接或者重新连接。在dubbo中，如果心跳超时则进行重连。

除了心跳以外，我们可以看到HeaderExchangeChannel对client再次进行了封装，它的作用是将要发送的实际数据封装成com.alibaba.dubbo.remoting.exchange.Request对象。

最终获得的HeaderExchangeChannel被封装到HeaderExchangeClient中，传入到DubboInvoker，最终DubboProtocol.refer返回了DubboInvoker。但流程还未结束，还记得我们一开头提起的wrapper类吧。下面来看看这两个类还做了哪些操作。

DubboProtocol.refer执行后，进入到ProtocolFilterWrapper，其refer代码如下：

[java] [view plain copy](#)



8

```
1. public <T> Invoker<T> refer(Class<T> type, URL url) throws
   RpcException {
2.     if (Constants.REGISTRY_PROTOCOL.equals(url.getProtocol()))
   {
3.         return protocol.refer(type, url);
4.     }
5.     // protocol为dubbo时执行到这里
6.     return buildInvokerChain(protocol.refer(type, url),
   Constants.REFERENCE_FILTER_KEY, Constants.CONSUMER);
7. }
8.
9. private static <T> Invoker<T> buildInvokerChain(final
   Invoker<T> invoker, String key, String group) {
10.    // 初始的last为刚刚创建的DubboInvoker
11.    Invoker<T> last = invoker;
12.    // 加载group为consumer的Filter, 加载到的Filter依次为:
13.    // com.alibaba.dubbo.rpc.filter.ConsumerContextFilter
14.    //
   com.alibaba.dubbo.rpc.protocol.dubbo.filter.FutureFilter
15.    // com.alibaba.dubbo.monitor.support.MonitorFilter
16.    List<Filter> filters =
   ExtensionLoader.getExtensionLoader(Filter.class).getActivateExten
   sion(invoker.getUrl(), key, group);
17.    if (filters.size() > 0) {
18.        // filter从最后一个开始依次封装, 最终形成一个链, 调用顺序为
   filters的顺序
19.        for (int i = filters.size() - 1; i >= 0; i --) {
20.            final Filter filter = filters.get(i);
21.            final Invoker<T> next = last;
22.            last = new Invoker<T>() {
23.
24.                public Class<T> getInterface() {
25.                    return invoker.getInterface();
26.                }
27.
28.                public URL getUrl() {
29.                    return invoker.getUrl();
30.                }
31.
32.                public boolean isAvailable() {
33.                    return invoker.isAvailable();
34.                }
35.
36.                public Result invoke(Invocation invocation)
```

```

throws RpcException {
37.         return filter.invoke(next, invocation);
38.     }
39.
40.     public void destroy() {
41.         invoker.destroy();
42.     }
43.
44.     @Override
45.     public String toString() {
46.         return invoker.toString();
47.     }
48. };
49. }
50. }
51. return last;
52. }

```

再看看ProtocolListenerWrapper:

[java] [view plain copy](#)



```

1. public <T> Invoker<T> refer(Class<T> type, URL url) throws
RpcException {
2.     if (Constants.REGISTRY_PROTOCOL.equals(url.getProtocol()))
{
3.         return protocol.refer(type, url);
4.     }
5.     return new ListenerInvokerWrapper<T>(protocol.refer(type,
url),
6.         Collections.unmodifiableList(
7.
ExtensionLoader.getExtensionLoader(InvokerListener.class)
8.         .getActivateExtension(url,
Constants.INVOKER_LISTENER_KEY)));
9. }
10.
11. // ListenerInvokerWrapper构造方法
12. public ListenerInvokerWrapper(Invoker<T> invoker,
List<InvokerListener> listeners){
13.     if (invoker == null) {
14.         throw new IllegalArgumentException("invoker ==
null");
15.     }
16.     this.invoker = invoker;

```

```

17.     this.listeners = listeners;
18.     if (listeners != null && listeners.size() > 0) {
19.         for (InvokerListener listener : listeners) {
20.             if (listener != null) {
21.                 try {
22.                     // 直接触发referred方法
23.                     listener.referred(invoker);
24.                 } catch (Throwable t) {
25.                     logger.error(t.getMessage(), t);
26.                 }
27.             }
28.         }
29.     }
30. }

```

listener在consumer初始化和destroy时生效，不影响正常的执行，默认情况下listeners为空。

到这里InvokerDelegete的生成基本上完成了，结合第一篇consumer的介绍，我们可以得到下图（后续我们再讲讲各个类的具体实现）：



