Dubbo RPC 之 Protocol协议层（二）

本文将聚焦于Dubbo协议下的 `Protocol` 实现 `DubboProtocol` ，以及利用装饰模式实现的 `ProtocolFilterWrapper` 和 `ListenerExporterWrapper` 。

> **NOTE**
>
> 本文所涉及剖析源码不少依赖于Dubbo的SPI机制，在翻阅之前先请仔细阅读《Dubbo之SPI扩展点加载》一文。

`DubboProtocol` → `AbstractProtocol`

在本文开篇为了搞清楚Protocol协议层的作用，耗费不少笔墨，上来就剖析其实现，难免会感觉吃力，因此之后便着重挨个分析相关的`Result`、 `Invocation` 、 `Invoker` ，扫清障碍后，下述我们结合代码为本文画上一个圆满的句号。

`AbstractProtocol` 基类

同样，该类是为所有协议实现提供最基础的实现，主要着墨点在：

1. 整个协议实例的资源销毁处理。
2. 对所有的 `Invoker` 实现提供一个 `AsyncToSyncInvoker` 封装类，其目的是如果其实例所表示的微服务接口没有在签名或者配置总线声明自己使用异步机制调度时，便将其转换为同步调用。

异步转同步操作 `AsyncToSyncInvoker`

`AsyncToSyncInvoker` 的实现原理很简单，是一个装饰类，实际完成工作的是被装饰的另一个 `Invoker` 实例，因此对应接口的几乎所有行为会直接委托给后者，只根据其自身特点将实现 `invoke()` 方法加以特别处理。

过程便是先调用被封装 `invoker` 对象的同名 `invoke()` 方法，获得一个超类为 `CompleteableFuture<Result>` 的 `Result` 类的对象R，如果从入参 `Invocation` 中检验到不需要使用异步调用模式，则调用后者的 `get()` 方法，它会等待直到响应回来，除非超时。除此之外没什么差别，依然返回的是 `invoker.invoke(invocation)` 返回的结果R。

```java
public class AsyncToSyncInvoker<T> implements Invoker<T> {

    private Invoker<T> invoker;

    public Result invoke(Invocation invocation) throws RpcException {
        Result asyncResult = invoker.invoke(invocation);

        try {
            if (InvokeMode.SYNC == ((RpcInvocation) invocation).getInvokeMode()) {
                asyncResult.get(Integer.MAX_VALUE, TimeUnit.MILLISECONDS);
            }
        } catch (InterruptedException | ExecutionException | Throwable e) {
            ...
        }
        return asyncResult;
    }

    ...
}
```

在应用程序作为客户端使用 `refer()` 方法引用微服务的时候， `AbstractProtocol` 基类会利用该 `AsyncToSyncInvoker` 对其引用的任何微服务实例进行一层包装适配。因此另外提炼了一个需子类实现的抽象方法，如下：

```java
/**
* 客户端使用接口的类型和配置总线URL获得一个彼端微服务的引用实例
*/
@Override
public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {
    return new AsyncToSyncInvoker<>(protocolBindingRefer(type, url));
}

protected abstract <T> Invoker<T> protocolBindingRefer
    (Class<T> type, URL url) throws RpcException;
```

资源销毁处理

一个 `Invoker` 对象可以被认为是一个微服务实例，分成两大类，服务端微服务实例和客户端微服务引用实例，仔细阅读过上文应该不难明白他们的差异性，前者是真正提供服务的，其Invoker类无需具体协议实现，而后者是客户端这边的一个概念，使得应用可以像引用本地服务一样去对前者发起请求。

为了对微服务实例方便进行统一管理， `AbstractProtocol` 申明了如下两个变量， `exporterMap` 用于缓存所有服务端和客户端导出的所有微服务实例，而 `invokers` 则仅仅用于缓存所有做RPC调用的微服务引用实例。 细究起来，其实 `exporterMap` 和 `invokers` 的作用也基本限定在做销毁处理而已，尽管前者将实例封装在一个 `Exporter` 类型的对象中。

JAVA

```
protected final Map<String, Exporter<?>> exporterMap =
    new ConcurrentHashMap<String, Exporter<?>>();

protected final Set<Invoker<?>> invokers =
    new ConcurrentHashSet<Invoker<?>>();
```

资源的销毁过程比较简单，逐个遍历容器中的元素，先从容器中移除其有元素应用，再调用元素自身的 `destroy()` 或 `unexport()` 方法，实际上方法 `unexport()` 背后执行的依然是一个Invoker实例的 `destroy()` 。

JAVA

```
@Override
public void destroy() {
    for (Invoker<?> invoker : invokers) {
        if (invoker != null) {
            invokers.remove(invoker);
            try {
                if (logger.isInfoEnabled()) {
                    logger.info("Destroy reference: " + invoker.getUrl());
                }
                invoker.destroy();
            } catch (Throwable t) {
                logger.warn(t.getMessage(), t);
            }
        }
    }
    for (String key : new ArrayList<String>(exporterMap.keySet())) {
        Exporter<?> exporter = exporterMap.remove(key);
        if (exporter != null) {
            try {
                if (logger.isInfoEnabled()) {
                    logger.info("Unexport service: " + exporter.getInvoker().getUrl());
                }
                exporter.unexport();
            } catch (Throwable t) {
                logger.warn(t.getMessage(), t);
            }
        }
    }
}
```

Dubbo协议实现 DubboProtocol

`DubboProtocol` 是基于Dubbo协议对 `Protocol` 的实现，前面这半句话说感觉说得颇有问题，协议实际上如文章开头所言，只有在通讯双方发生会话时才有意义，其中包含了信息的编解码处理、流程控制等一序列复杂的约定，而 `Protocol` 的实现类仅仅是相关实现细节的最后总组装而已。

整个 `DubboProtocol` 的实现相当复杂，我们下面将按照客户端和服务端分成两个大的章节，逐步深入。

服务引用实现

如下述源码所示，服务引用的过程，实际上是产生微服务Invoker实例的过程，初步看起来流程相当精简。

```java
public <T> Invoker<T> protocolBindingRefer(Class<T> serviceType, URL url) throws
RpcException {
    optimizeSerialization(url);

    // create rpc invoker.
    DubboInvoker<T> invoker = new DubboInvoker<T>(serviceType, url, getClients(url),
invokers);
    invokers.add(invoker);

    return invoker;
}
```

如前文已经提及Protocol需要负责给客户端引用微服务实例提供用于远端通讯的ExchangeClient，也就是 `getClients(url)` 所表示的那一截代码，然而沿着它扩散开来，有如从公园外墙推开一扇门，大有洞天，难以尽收眼底。眼花缭乱之际也是眩晕之时，要一探究竟，咱还是先把绕道其看看和其密切相关的两个 `ExchangeClient` 接口实现类——`LazyConnectExchangeClient` & `ReferenceCountExchangeClient` 。

ReferenceCountExchangeClient

微服务开发过程中，一个应用引用多个微服务实例是很常见的事情，一个服务端微服务作为一个应用占用了一个JVM虚拟机，自然就拥有了其所在主机的唯一端口号，而连接它的客户端为了效率上的考量会利用连接池供多个线程并发地访问它，Dubbo的实现中，从单一客户端连接同一个微服务的微服务引用实例是可以存在多份的。这种情况下，一个 `ExchangeClient` 对象就可以被这多份实例共享，这时就不能随随便便被 `close` 掉，只有不再有微服务引用实例使用它时才可 `close` 。基于这种需求，Dubbo专门为 `ExchangeClient` 提供了一个使用引用计数的封装类 `ReferenceCountExchangeClient` 。

具体实现上很简单，和一般的装饰器类一样，实现 `ExchangeClient` 接口，其无关当前特定业务的接口方法全部委托给其实现同一接口的引用属性 `client` 完成，在特定方法进行业务逻辑改写处理。它声明了一个表示引用计数的基于CAS实现的原子变量 `AtomicInteger referenceCount` ，被实例化时，执行+1操作，后面一旦被另外其它一个同微服务的服务引用 `Invoker` 对象所使用，便再次执行+1操作，而 `close` 操作时会首先对其执行-1操作，然后检查该属性是否为0，为0可以调用被封装的 `client` 对象的 `close()` 方法安全关闭，否则直接忽略掉。

在当前Protocol协议层Dubbo对 `ExchangeClient` 的正常close操作做了更进一步处理，会使用 `LazyConnectExchangeClient` 封装将已经关闭的对象，如果当前 `ReferenceCountExchangeClient` 实例被再一次调用，该实例被会神奇般的复活。当然为了不导致其多层嵌套引用一个 `ReferenceCountExchangeClient` 类型的 `ExchangeClient` 实例对象，`LazyConnectExchangeClient` 不再直接封装一个 `ExchangeClient` 实例，而是基于后者获取其 `ExchangeHandler` 引用实现 `ExchangeClient` 接口。

JAVA

```java
final class ReferenceCountExchangeClient implements ExchangeClient {

    private final URL url;
    private final AtomicInteger referenceCount = new AtomicInteger(0);

    private ExchangeClient client;

    public ReferenceCountExchangeClient(ExchangeClient client) {
        this.client = client;
        referenceCount.incrementAndGet();
        this.url = client.getUrl();
    }
    ...
    /**
     * close() is not idempotent any longer
     */
    @Override
    public void close() {
        close(0);
    }

    @Override
    public void close(int timeout) {
        if (referenceCount.decrementAndGet() <= 0) {
            if (timeout == 0) {
                client.close();

            } else {
                client.close(timeout);
            }

            replaceWithLazyClient();
        }
    }
     /**
      * when closing the client, the client needs to be set to
LazyConnectExchangeClient, and if a new call is made,
      * the client will "resurrect".
      *
      * @return
      */
     private void replaceWithLazyClient() {
         // this is a defensive operation to avoid client is closed by accident, the
initial state of the client is false
         URL lazyUrl = URLBuilder.from(url)
                 .addParameter(LAZY_CONNECT_INITIAL_STATE_KEY, Boolean.FALSE)
                 .addParameter(RECONNECT_KEY, Boolean.FALSE)
                 .addParameter(SEND_RECONNECT_KEY, Boolean.TRUE.toString())
                 .addParameter("warning", Boolean.TRUE.toString())
                 .addParameter(LazyConnectExchangeClient.REQUEST_WITH_WARNING_KEY,
true)
                 .addParameter("_client_memo",
"referencecounthandler.replacewithlazyclient")
                 .build();

         /**
```

```
             * the order of judgment in the if statement cannot be changed.
             */
            if (!(client instanceof LazyConnectExchangeClient) || client.isClosed()) {
                client = new LazyConnectExchangeClient(lazyUrl,
client.getExchangeHandler());
            }
        }
        /**
         * The reference count of current ExchangeClient, connection will be closed if all
invokers destroyed.
         */
        public void incrementAndGetCount() {
            referenceCount.incrementAndGet();
        }

        ...
}
```

LazyConnectExchangeClient

本质上就如同 `ReferenceCountExchangeClient` 一样， `LazyConnectExchangeClient` 也是用于对某个目标 `ExchangeClient` 实例进行封装，但实现上功能和目的完全不同，后者的主要目的是在已知“配置总线URL”和“网络事件监听器ExchangeHandler”这两个输入的情况下延迟创建 `ExchangeClient` 实例，将这一时刻推迟到业务出站请求之时。

在《【六】Dubbo远程通讯 之 信息交换层》一文中曾提到，由于Dubbo的分层模型，网络I/O事件的回调是自下往上、逐层执行的，上层是对下一层的封装和增强，因此如果某一层一种组件对象的创建是依赖比其更低一级的，那么只能在其网络I/O事件的回调中反向完成其创建操作，还得使用一些排重手段，保证只会实例化一次，另外还需搭配一些前验代码。

类似对象创建方式在惰性实例化时也很常见，比如一个类提供了n个方法，其中有几个方法会涉及到实例化。 `LazyConnectExchangeClient` 中的 `client` 属性采取的便是这种方式。

如下述代码所示，分为初始化实现和调用两部分：第一部分使用volatile可见性修饰符、ReentrantLock可重入锁、锁的双检这几重机制确保在多线程并发情况下依然只会安全的实例化 `ExchangeClient` 对象一次；第二部分则是初始化调用，在每一个出站事件回调方法中均调用第一部分提供的 `initClient()` 方法，确保只有一个实例。

JAVA

```java
final class LazyConnectExchangeClient implements ExchangeClient {

    ...
    private final URL url;
    private final ExchangeHandler requestHandler;


    private volatile ExchangeClient client;
    private final Lock connectLock = new ReentrantLock();

    private void initClient() throws RemotingException {
        if (client != null) {
            return;
        }
        if (logger.isInfoEnabled()) {
            logger.info("Lazy connect to " + url);
        }
        connectLock.lock();
        try {
            if (client != null) {
                return;
            }
            this.client = Exchangers.connect(url, requestHandler);
        } finally {
            connectLock.unlock();
        }
    }
//==============================
//调用initClient初始化ExchangeClient实例
//==============================
    @Override
    public void send(Object message) throws RemotingException {
        initClient();
        client.send(message);
    }

    @Override
    public void send(Object message, boolean sent) throws RemotingException {
        initClient();
        client.send(message, sent);
    }

    @Override
    public CompletableFuture<Object> request(Object request, int timeout)
            throws RemotingException {
        warning();
        initClient();
        return client.request(request, timeout);
    }

    @Override
    public CompletableFuture<Object> request(Object request)
            throws RemotingException {
        warning();
        initClient();
        return client.request(request);
```

```
    }
    ...
}
```

于一些不涉及数据出站处理的方法， LazyConnectExchangeClient 专门为其提供了如下的前验检查代码，它们分别是“removeAttribute、setAttribute、 reconnect、reset、getChannelHandler”，检查通过则直接使用被封装的 ExchangeClient 实例完成功能，这类方法的特点是调用方需要感知到行为的发生。

JAVA

```java
private void checkClient() {
    if (client == null) {
        throw new IllegalStateException(
                "LazyConnectExchangeClient state error. the client has not be init
.url:" + url);
    }
}
```

对于 close 类操作则处理相对很简单，不满足条件时，直接进行忽略处理，如下所示：

JAVA

```java
@Override
public void startClose() {
    if (client != null) {
        client.startClose();
    }
}
```

有两个在构造方法中出现的配置总线参数，这里有必要提及下，分别是 url["send.reconnect"] 和 url["lazyclient_request_with_warning"] 。如下述源码所示，前者在当前对象创建时加入到配置总线中，用于确保该内嵌对象在向彼端发送请求之时，所使用的通道Channel是连接着的（在使用 Client 发送数据时，若连接断开，则自动连接）；而后者则是用于确认是否需要提示警告信息，需要的话，则每5000次的方法调用会提醒一次，对于一个频繁使用的微服务，其所使用 ExchangeClient 不应采用惰性模式。。

JAVA

```
final class LazyConnectExchangeClient implements ExchangeClient {
    ...
    /**
     * when this warning rises from invocation, program probably have bug.
     */
    protected static final String REQUEST_WITH_WARNING_KEY =
"lazyclient_request_with_warning";
    protected final boolean requestWithWarning;
    private final int warning_period = 5000;
    private AtomicLong warningcount = new AtomicLong(0);

    public LazyConnectExchangeClient(URL url, ExchangeHandler requestHandler) {
        // lazy connect, need set send.reconnect = true, to avoid channel bad status.
        this.url = url.addParameter(SEND_RECONNECT_KEY, Boolean.TRUE.toString());
        this.requestHandler = requestHandler;

        //DEFAULT_LAZY_CONNECT_INITIAL_STATE的默认值为true
        this.initialState = url.getParameter(LAZY_CONNECT_INITIAL_STATE_KEY,
DEFAULT_LAZY_CONNECT_INITIAL_STATE);
        this.requestWithWarning = url.getParameter(REQUEST_WITH_WARNING_KEY, false);
    }


    /**
     * If {@link #REQUEST_WITH_WARNING_KEY} is configured, then warn once every 5000
invocations.
     */
    private void warning() {
        if (requestWithWarning) {
            if (warningcount.get() % warning_period == 0) {
                logger.warn(new IllegalStateException("safe guard client , should not
be called ,must have a bug."));
            }
            warningcount.incrementAndGet();
        }
    }
}
```

ExchangeClient 候选集准备

经过上述的两个小章节的铺垫后，这时再回过头来，便可以比较轻松地理解Dubbo中是如何准备 ExchangeClient 的候选集的。在关于 ReferenceCountExchangeClient 的实现探究过程中，我们清楚，对于分别占用一个JVM的一对“客户端 ↔ 服务端”来说，他们之间存在通讯连接通道 Channel 和 ExchangeClient 一一绑定可以存 在多份，同时客户端也可以具备多份服务引用 Invoker 实例，实现上“ Invoker 服务引用实例”和“ ExchangeClient 客户端通讯处理实例”的关系可以是一对一或一对多的独占模式，也可以使多对一或者多对多的共享模式。

默认情况下，也即没有设置 url["connections"] 参数，采用的是共享模式，这时可以通过设置 url["shareconnections"] 或者系统参数 env["shareconnections"] ，防止默认只有一个共享的通讯连接通道而引发的瓶颈问题。此外,显示配置的情况下使用的是独占模式。

JAVA

```
private ExchangeClient[] getClients(URL url) {
    // whether to share connection

    boolean useShareConnect = false;

    int connections = url.getParameter(CONNECTIONS_KEY, 0);
    List<ReferenceCountExchangeClient> shareClients = null;
    // if not configured, connection is shared, otherwise, one connection for one
service
    if (connections == 0) {
        useShareConnect = true;

        /**
         * The xml configuration should have a higher priority than properties.
         */
        String shareConnectionsStr = url.getParameter(
            SHARE_CONNECTIONS_KEY, (String) null);
        connections = Integer.parseInt(StringUtils.isBlank(shareConnectionsStr) ?
            ConfigUtils.getProperty(SHARE_CONNECTIONS_KEY, DEFAULT_SHARE_CONNECTIONS) :
                shareConnectionsStr);
        shareClients = getSharedClient(url, connections);
    }

    ExchangeClient[] clients = new ExchangeClient[connections];
    for (int i = 0; i < clients.length; i++) {
        if (useShareConnect) {
            clients[i] = shareClients.get(i);

        } else {
            clients[i] = initClient(url);
        }
    }

    return clients;
}
```

独占模式下的单个 ExchangeClient 的初始化操作

独占模式下的 ExchangeClient 的初始化相对来说比较简单：

1. 首先，由于性能问题， DubboProtocol 协议实现在网络传输层不会选用低效的BIO模式，确保能找到 (url["server"] | url["client"] | "netty") 所指定的 Transporter 扩展点实现；
2. 然后再配置总线中增设解码 url["codec"] = "dubbo" 和心跳参数 url["heartbeat"] = "60000" ，确保：1）在信息交换层采用了Dubbo协议的编解码；2）使用心跳机制维持客户端到服务端的长连接，默认心跳时长为一分钟；
3. 随后就是使用上述增设了参数的配置总线url参数实例化 ExchangeClient 对象或者 LazyConnectExchangeClient 对象，后者需总线中已指定 url["lazy"] = "true" ；

JAVA

```java
/**
 * Create new connection
 *
 * @param url
 */
private ExchangeClient initClient(URL url) {

    // client type setting.
    String str = url.getParameter(CLIENT_KEY,
        url.getParameter(SERVER_KEY, DEFAULT_REMOTING_CLIENT));

    url = url.addParameter(CODEC_KEY, DubboCodec.NAME);
    // enable heartbeat by default
    url = url.addParameterIfAbsent(HEARTBEAT_KEY, String.valueOf(DEFAULT_HEARTBEAT));

    // BIO is not allowed since it has severe performance issue.
    if (str != null && str.length() > 0 &&
            !ExtensionLoader.getExtensionLoader(Transporter.class).hasExtension(str)) {
        throw new RpcException("Unsupported client type: " + str + "," +
            " supported client type is " + StringUtils.join(
            ExtensionLoader.getExtensionLoader(Transporter.class).
                getSupportedExtensions(), " "));
    }

    ExchangeClient client;
    try {
        // connection should be lazy
        if (url.getParameter(LAZY_CONNECT_KEY, false)) {
            client = new LazyConnectExchangeClient(url, requestHandler);

        } else {
            client = Exchangers.connect(url, requestHandler);
        }

    } catch (RemotingException e) {
        throw new RpcException("Fail to create remoting client for service(" + url +
"): " + e.getMessage(), e);
    }

    return client;
}
```

我们知道，在微服务开发中，一个服务可以定义多个接口，也即对应着Java中的 `interface` 和其中定义的若干方法，一个服务端服务通常占用了一个JVM虚拟机，处于其中的服务接口可能被访问的频度差异非常巨大，也有可能分布是比较均匀的。

共享模式下的单个 `ExchangeClient` 的初始化操作

实际上共享模式只是独占模式的一种特例，因此其 `ExchangeClient` 的实例化直接调用了 `initClient()` ，这也意味着 `ReferenceCountExchangeClient` 可以用于包装 `LazyConnectExchangeClient` ，如下述源码所示：

JAVA

```java
/**
 * Bulk build client
 *
 * @param url
 * @param connectNum
 * @return
 */
private List<ReferenceCountExchangeClient>
        buildReferenceCountExchangeClientList(URL url, int connectNum) {
    List<ReferenceCountExchangeClient> clients = new ArrayList<>();

    for (int i = 0; i < connectNum; i++) {
        clients.add(buildReferenceCountExchangeClient(url));
    }

    return clients;
}

/**
 * Build a single client
 *
 * @param url
 * @return
 */
private ReferenceCountExchangeClient buildReferenceCountExchangeClient(URL url) {
    ExchangeClient exchangeClient = initClient(url);

    return new ReferenceCountExchangeClient(exchangeClient);
}
```

共享模式下的初始化之所以复杂，原因是对于同一个微服务的客户端服务引用 `Invoker` ，每次其新创建实例的时候，均需要执行对 `ReferenceCountExchangeClient` 实例的计数器的 `+1` 操作，它可能是一个已经创建了并缓存在 `Map<String, List<ReferenceCountExchangeClient>>` 缓存Map中键为微服务的address地址。 由于该Map是共享的，并发模式下需要确保其安全，业务上需要确保能够稳定地提供相应数量的共享ReferenceCountExchangeClient对象，因此还需要在 线程安全的前提下实现对已经失效或 `close` 掉的实例做替换处理。

JAVA

```java
/**
 * <host:port,Exchanger>
 */
private final Map<String, List<ReferenceCountExchangeClient>> referenceClientMap = new
ConcurrentHashMap<>();
private final ConcurrentMap<String, Object> locks = new ConcurrentHashMap<>();

/**
 * Get shared connection
 *
 * @param url
 * @param connectNum connectNum must be greater than or equal to 1
 */
private List<ReferenceCountExchangeClient> getSharedClient(URL url, int connectNum) {
    String key = url.getAddress();
    List<ReferenceCountExchangeClient> clients = referenceClientMap.get(key);

    if (checkClientCanUse(clients)) {
        batchClientRefIncr(clients);
        return clients;
    }

    locks.putIfAbsent(key, new Object());
    synchronized (locks.get(key)) {
        clients = referenceClientMap.get(key);
        // dubbo check
        if (checkClientCanUse(clients)) {
            batchClientRefIncr(clients);
            return clients;
        }

        // connectNum must be greater than or equal to 1
        connectNum = Math.max(connectNum, 1);

        // If the clients is empty, then the first initialization is
        if (CollectionUtils.isEmpty(clients)) {
            clients = buildReferenceCountExchangeClientList(url, connectNum);
            referenceClientMap.put(key, clients);

        } else {
            for (int i = 0; i < clients.size(); i++) {
                ReferenceCountExchangeClient referenceCountExchangeClient
                    = clients.get(i);
                // If there is a client in the list that is no longer available,
                //create a new one to replace him.
                if (referenceCountExchangeClient == null ||
                        referenceCountExchangeClient.isClosed()) {
                    clients.set(i, buildReferenceCountExchangeClient(url));
                    continue;
                }

                referenceCountExchangeClient.incrementAndGetCount();
            }
        }

        /**
```

```
             * I understand that the purpose of the remove operation
             * here is to avoid the expired url key
             * always occupying this memory space.
             */
            locks.remove(key);

            return clients;
        }
    }
```

上述代码中也使用了很常见的锁的双检机制，当传入给 `checkClientCanUse` 的 `ReferenceCountExchangeClient` 对象列表中的只要有一个对象处于无效或者close状态，便随后进入主体逻辑中，确保满足数目要求的基础上，列表中所有的对象均可用，也即 `ExchangeClient` 所代表的客户端和服务端保 持长连状态。否则只会简单的对属于目标服务的下的 `ReferenceCountExchangeClient` 进行 `+1` 操作。

JAVA

```
/**
 * Check if the client list is all available
 *
 * @param referenceCountExchangeClients
 * @return true-available, false-unavailable
 */
private boolean checkClientCanUse(List<ReferenceCountExchangeClient>
referenceCountExchangeClients) {
    if (CollectionUtils.isEmpty(referenceCountExchangeClients)) {
        return false;
    }

    for (ReferenceCountExchangeClient referenceCountExchangeClient :
referenceCountExchangeClients) {
        // As long as one client is not available, you need to replace the unavailable
client with the available one.
        if (referenceCountExchangeClient == null ||
referenceCountExchangeClient.isClosed()) {
            return false;
        }
    }

    return true;
}

/**
 * Increase the reference Count if we create new invoker shares same connection, the
connection will be closed without any reference.
 *
 * @param referenceCountExchangeClients
 */
private void batchClientRefIncr(List<ReferenceCountExchangeClient>
referenceCountExchangeClients) {
    if (CollectionUtils.isEmpty(referenceCountExchangeClients)) {
        return;
    }

    for (ReferenceCountExchangeClient referenceCountExchangeClient :
referenceCountExchangeClients) {
        if (referenceCountExchangeClient != null) {
            referenceCountExchangeClient.incrementAndGetCount();
        }
    }
}
```

服务导出实现

上述有关 `DubboProtocol` 源码的分析中，已经刻意地忽略掉了 `requestHandler` 创建的问题，在前述有关Protocol协议层的讨论中也没有提及客户端服务导出的相关逻辑。实际上服务导出并不限于服务端，它同时也存在于客户端。

在前面的Dubbo实现源码剖析中，我们已经知道，不管是客户端还是服务端，都可以直接使用其通道Channel向彼端主动发送消息数据，但是对于来自彼端的请求则于应用层来说是被动的，只能在网络I/O事件就绪后，由框架回调应用层的逻辑代码。因此Dubbo在协议层需要在回调方法 `received()` 中，将收到的代表原生请求 `message` 类型为 `Invocation` 的请求转给对应的微服务实例或微服务引用实例处理，处理完再返回结果。

因此：

- 在微服务的原生Java方法发起调用之前，服务端需要导出提供服务的 `Invoker` ，而客户端则需要导出引用服务的 `Invoker` ，便于发起RPC调用；
- 无论是客户端还是服务端，均需要根据某种规则获取到 `Invoker` 对象微服务或其客户端引用的实例；
- 对于服务端还需要创建该 `Invoker` 的 `ExchangeServer` 服务实例，服务连入客户端；
- 实现 `ExchangeHandler` 被装饰者业务逻辑，响应 `Invocation` 类型入站请求详见下述相关章节；

`Invoker` 实例服务实例&引用实例导出

在服务导出实现源码中， `Exporter` 接口及其实现类 `DubboExporter<T>` → `AbstractExporter<T>` ，存在的目的更多的是保持框架分层业务语义上的完整性， 用于封装一个 `Invoker` 实例，便于后续进行销毁处理。其实例化也即导出，调用 `unexport` 时便驱动执行 `Invoker` 实例的 `destroy()` 方法，为了确保只会销毁操作不会重复执行，声明了一个辅助变量——`volatile boolean unexported` 。

所有被导出 `Invoker` 实例先被装入一个 `DubboExporter` 实例，随后整体载入到 `Map<String, Exporter<?>> exporterMap` 缓存中，其中的键值的表示形式为“ `[group/]serviceName[:version]:port` ”[XXX]：XXX可选，其中包含的4个元素分别对应配置总线URL中的值：1） `url["group"]` ；2） `url.path` ；3） `url["version"]` ；4） `url.port` 。

服务导出是由Dubbo框架调用方法 `public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException` 完成的，其传入的Invoker要么是框架使用动态代理方式实现的，要么就是协议层中的由第三方提供的类似 `DubboInvoker` 实现。

JAVA

```java
public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
    URL url = invoker.getUrl();

    // export service.
    // 构建完Key之后，将invoker缓存起来
    String key = serviceKey(url);
    DubboExporter<T> exporter = new DubboExporter<T>(invoker, key, exporterMap);
    exporterMap.put(key, exporter);

    ...

    //配置总线告知是server端才需要创建服务实例，分下参见下文
    openServer(url);
    optimizeSerialization(url);

    return exporter;
}
```

ExchangeServer 创建

默认而言，如果服务配置总线中没有设置 `url["isserver"]` ，Dubbo会默认为当前的 `export()` 操作准备提供服务的 `ExchangeServer` 实例。上述提到，引用同一个服务端微服务实例的所有 `ReferenceCountExchangeClient` 对象会被装入到一个列表中，最后再以 `<host:port,Exchanger>` 的形式缓存起来，也即类型为 `ConcurrentHashMap<String, List<ReferenceCountExchangeClient>>` 的 `referenceClientMap` 变量。

同样，所有服务端提供服务的 `ExchangeServer` 也会以类似的形式缓存在 `ConcurrentHashMap<String, ExchangeServer>` 类型的serverMap中，在创建服务实例时，若发现已经存在对应的实例，则会使用配置总线对其参数做重设处理。

JAVA

```java
private final Map<String, ExchangeServer> serverMap = new ConcurrentHashMap<>();

private void openServer(URL url) {
    // find server.
    String key = url.getAddress();
    //client can export a service which's only for server to invoke
    boolean isServer = url.getParameter(IS_SERVER_KEY, true);
    if (isServer) {
        ExchangeServer server = serverMap.get(key);

        //使用双检模式创建服务实例
        if (server == null) {
            synchronized (this) {
                server = serverMap.get(key);
                if (server == null) {
                    serverMap.put(key, createServer(url));
                }
            }
        } else {
            // server supports reset, use together with override
            server.reset(url);
        }
    }
}
```

下述创建 ExchangeServer 实例的创建过程中， Exchangers.bind(url, requestHandler) 为最核心的一句，需要指定 url["channel.readonly.sent"] = (|true)、url["heartbeat"] = (|60000)、url["codec"] = "dubbo" ，同时需要确保当前应用中存在由 url["server"] 和 url["client"] 所配置的 Transporter 扩展点。

JAVA

```
private ExchangeServer createServer(URL url) {
    //配置总线相关参数设置
    url = URLBuilder.from(url)
            // send readonly event when server closes, it's enabled by default
            .addParameterIfAbsent(CHANNEL_READONLYEVENT_SENT_KEY,
Boolean.TRUE.toString())
            // enable heartbeat by default
            .addParameterIfAbsent(HEARTBEAT_KEY, String.valueOf(DEFAULT_HEARTBEAT))
            .addParameter(CODEC_KEY, DubboCodec.NAME)
            .build();
    String str = url.getParameter(SERVER_KEY, DEFAULT_REMOTING_SERVER);

    //校验是否配置服务类型已经存在相应的实现，由SPI指定
    if (str != null && str.length() > 0 && !ExtensionLoader.
            getExtensionLoader(Transporter.class).hasExtension(str)) {
        throw new RpcException("Unsupported server type: " + str + ", url: " + url);
    }

    ExchangeServer server;
    try {
        server = Exchangers.bind(url, requestHandler);
    } catch (RemotingException e) {
        throw new RpcException("Fail to start server(url: "
            + url + ") " + e.getMessage(), e);
    }

    str = url.getParameter(CLIENT_KEY);
    if (str != null && str.length() > 0) {
        Set<String> supportedTypes = ExtensionLoader.getExtensionLoader(
            Transporter.class).getSupportedExtensions();
        if (!supportedTypes.contains(str)) {
            throw new RpcException("Unsupported client type: " + str);
        }
    }

    return server;
}
```

响应RPC调用

前面已经阐述过入站的网络数据包括Request请求和Response响应，当他们的网络I/O事件就绪时，便会触发绑定在通道上的 `HeaderExchangeHandler` 事件监听器 A 的 `received()` 方法，处于更加底层的信息交换层会将其中的 `Request` 请求（需要返回响应）转发给 `ExchangeHandler` 对象 B 定义的 `reply()` 方法，由其构建 `CompletableFuture<Object>` 类型的响应结果。①

注：

1）**ExchangeHandler**：`public CompletableFuture<Object> reply(ExchangeChannel channel, Object message) throws RemotingException`；

2）**ChannelHandler**：`public void received(Channel channel, Object message) throws RemotingException`

IMPORTANT

B的类型是一个扩充版的 `ChannelHandler` ，而A的类型 `HeaderExchangeHandler` 则是前者装饰者实现，也即A封装了B，A的I/O回调最终都会委托给B。

在Dubbo协议层中，对类型为 `Invocation` 非 `[Request、Response、String]` 外的入站请求做了同样的处理，也就是 `ExchangeHandler#reply()` 方法会进一步调用 `Invoker` 实例的 `invoke()` 方法，以完成对应Java原生方法的调用，或者由Java原生方法转换后的跨机网络请求。②

需要注意的是，上述讨论的两种被调用的场景①和②， `reply()` 均属于同一个 `ExchangeHandler` 对象，因此要求第一种场景中，其Request对象封装的 `mData` 也是 `Invocation` 类型的，否则会抛出异常。

大体实现如下述源码所示，会首先从 `exporterMap` 缓存中取得相对应的 `Invoker` 实例，使用它回调表征原生Java方法的 `Invocation` 对象：

JAVA

```
private ExchangeHandler requestHandler = new ExchangeHandlerAdapter() {
    ...
    @Override
    public CompletableFuture<Object> reply(ExchangeChannel channel, Object message)
throws RemotingException {

        //当前ExchangeHandlerAdapter只响应消息为Invocation类型的请求
        if (!(message instanceof Invocation)) {
            throw new RemotingException(channel, "Unsupported request: "
                    + (message == null ? null : (message.getClass().getName() + ": " +
message))
                    + ", channel: consumer: " + channel.getRemoteAddress()
                    + " --> provider: " + channel.getLocalAddress());
        }

        Invocation inv = (Invocation) message;
        Invoker<?> invoker = getInvoker(channel, inv);
        ...

        RpcContext.getContext().setRemoteAddress(channel.getRemoteAddress());

        //间接完成对应Java原生方法的调用或者由Java原生方法转换后的跨机网络请求
        Result result = invoker.invoke(inv);

        //先调用completionFuture()将CompletionStage<Result>
        //  转换成CompletableFuture<Result>
        //再调用thenApply(Function.identity())装换成CompletableFuture<Object>
        //利用了泛型出参自带类型转换特性，也即：
        // <U> CompletableFuture<U> thenApply(Function<? super T,? extends U> fn)
        return result.completionFuture().thenApply(Function.identity());
    }

    @Override
    public void received(Channel channel, Object message) throws RemotingException {
        if (message instanceof Invocation) {
            reply((ExchangeChannel) channel, message);

        } else {
            super.received(channel, message);
        }
    }
    ...


//根据当前通道内含信息及Invocation对象中的本地参数容器构建serviceKey，
//由其从exporterMap键值对中最终获取到Invoker对象
Invoker<?> getInvoker(Channel channel, Invocation inv) throws RemotingException {
    int port = channel.getLocalAddress().getPort();
    String path = inv.getAttachments().get(PATH_KEY);

    ...
    String serviceKey = serviceKey(port, path,
        inv.getAttachments().get(VERSION_KEY), inv.getAttachments().get(GROUP_KEY));

    DubboExporter<?> exporter = (DubboExporter<?>) exporterMap.get(serviceKey);
```

```
    if (exporter == null) {
        throw new RemotingException(channel, "Not found exported service: "
            + serviceKey + " in " + exporterMap.keySet() + ", may be version or group
mismatch "
            + ", channel: consumer: " + channel.getRemoteAddress() + " --> provider: "
            + channel.getLocalAddress() + ", message:" + inv);
    }

    return exporter.getInvoker();
}
```

另外，Dubbo允许为微服务实例或者引用实例配置 `url["ondisconnect"]` 和 `url["onconnect"]`，监听链入或者断链时的监听，如下述示例配置的 `ondisconnect`：

XML

```
<beans>
    <bean id="demoService" class="org.apache.dubbo.samples.impl.DemoServiceImpl"/>
    <dubbo:service async="true" interface="org.apache.dubbo.samples.api.DemoService"
        version="1.2.3" group="dubbo-simple" ref="demoService"
ondisconnect="disCallback"
        executes="4500" retries="7" owner="vict" timeout="5300"/>
</beans>
```

实际也就是对5种典型的网络I/O事件的 `connected` 和 `disconnected` 做适配处理，为其创建相应的 `RpcInvocation` 类的 `Invocation` 实例，以该实例和被回调方法接受的通道 `channel` 入参为参数，调用当前被装饰 `ExchangeHandler` 对象的 `received()` 方法。当然，只有在配置了链入或者断链的监听方法，对应事件回调中才会运行实际的 `received(channel, invocation)` 业务代码。

```java
private ExchangeHandler requestHandler = new ExchangeHandlerAdapter() {
    ...

    @Override
    public void connected(Channel channel) throws RemotingException {
        invoke(channel, ON_CONNECT_KEY);
    }

    @Override
    public void disconnected(Channel channel) throws RemotingException {
        if (logger.isDebugEnabled()) {
            logger.debug("disconnected from " + channel.getRemoteAddress()
                + ",url:" + channel.getUrl());
        }
        invoke(channel, ON_DISCONNECT_KEY);
    }

    private void invoke(Channel channel, String methodKey) {
        Invocation invocation = createInvocation(channel, channel.getUrl(), methodKey);
        if (invocation != null) {
            try {
                received(channel, invocation);
            } catch (Throwable t) {
                logger.warn("Failed to invoke event method " +
                    invocation.getMethodName() + "(), cause: " + t.getMessage(), t);
            }
        }
    }

    private Invocation createInvocation(Channel channel, URL url, String methodKey) {
        String method = url.getParameter(methodKey);
        if (method == null || method.length() == 0) {
            return null;
        }

        RpcInvocation invocation = new RpcInvocation(method, new Class<?>[0], new
Object[0]);
        invocation.setAttachment(PATH_KEY, url.getPath());
        invocation.setAttachment(GROUP_KEY, url.getParameter(GROUP_KEY));
        invocation.setAttachment(INTERFACE_KEY, url.getParameter(INTERFACE_KEY));
        invocation.setAttachment(VERSION_KEY, url.getParameter(VERSION_KEY));
        if (url.getParameter(STUB_EVENT_KEY, false)) {
            invocation.setAttachment(STUB_EVENT_KEY, Boolean.TRUE.toString());
        }

        return invocation;
    }
};
```

上述 `createInvocation(…)` 方法的实现表明，微服务实例或者引用实例的链入或者断链监听是通过本地存根机制实现的，这里暂时忽略处理，后续在分析PRC中的代理实现时会对类似AOP的存根机制做深入剖析，具体参考《Dubbo服务代理》一文。

其它

有关DubboProtocol的源码实现基本已剖析完，但前文提及的 `requestHandler` 类似于幽灵般的存在，服务实例和服务引用实例都有使用到，它的实现始终是从 `exporterMap` 缓存中获取的，但 `refer()` 产生的 `Invoker` 服务引用实例机会就是漂浮着的存在，并没有被存入 `exporterMap` ，仅从当前框架层几乎没法一览全貌，尚待后续。

ProtocolFilterWrapper >> Protocol

在微服务开发涉及RPC请求的场景中，常常有些和特定业务无关的需求，比如某个接口访问量的数据采集、记录访问日志、设置访问令牌等。类似的场景，在一般类似Spring的开发框架会采用拦截器来实现，同样Dubbo中也提供了类似的机制，拦截服务提供方和服务消费方的RPC调用，Dubbo中类似TPS限额的不少内置特性也是基于这一机制实现。

Dubbo内部给的实现方案是，采用装饰者模式，对 `Protocol` 实现做一层装饰，在其导出微服务实例，或者引出微服务引用实例时，加入一个拦截链，供框架或者应用层纳入更多的特性，大致源码如下：

JAVA

```
public class ProtocolFilterWrapper implements Protocol {

    private final Protocol protocol;

    public ProtocolFilterWrapper(Protocol protocol) {
        if (protocol == null) {
            throw new IllegalArgumentException("protocol == null");
        }
        this.protocol = protocol;
    }

    @Override
    public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
        if (REGISTRY_PROTOCOL.equals(invoker.getUrl().getProtocol())) {
            return protocol.export(invoker);
        }
        return protocol.export(buildInvokerChain(invoker, SERVICE_FILTER_KEY,
CommonConstants.PROVIDER));
    }

    @Override
    public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {
        if (REGISTRY_PROTOCOL.equals(url.getProtocol())) {
            return protocol.refer(type, url);
        }
        return buildInvokerChain(protocol.refer(type, url), REFERENCE_FILTER_KEY,
CommonConstants.CONSUMER);
    }
    ...
}
```

Filter 接口定义

Dubbo的内部RPC调用过程是异步的，出站请求和相对应的入站响应是两个界限明显的分段过程，前者不会因为后者还未执行完没有获得结果而阻塞，因此表征原生方法调用的 `Result invoke(Invocation invocation)` 出参 `Result` 会被设计成扩展 `CompletionStage<Result>` 的接口，RPC处理结果是基于响应式回调机制设置给 `Result` 的。同样，其拦截器也应该相应被设计成两阶段式的，如下述接口定义，其中用于响应阶段的 `Listener` 是可选的，如果实现了，需要接口实现类扩展自 `ListenableFilter` 抽象类。

JAVA

```
@SPI
public interface Filter {
    /**
     * Does not need to override/implement this method.
     */
    Result invoke(Invoker<?> invoker, Invocation invocation)
        throws RpcException;


    interface Listener {

        void onResponse(Result appResponse, Invoker<?> invoker, Invocation invocation);

        void onError(Throwable t, Invoker<?> invoker, Invocation invocation);
    }
}

public abstract class ListenableFilter implements Filter {

    protected Listener listener = null;

    public Listener listener() {
        return listener;
    }
}
```

拦截器执行原理

Dubbo中的拦截器的调度实现设计得非常巧妙，尽管其执行也是按顺序挨个执行的，但并没有直接呆板地使用列表遍历的形式，而是采用类似单向链表的形式，上一个拦截器运行完，会接着驱动下一个拦截器来接棒执行。

具体实现上，Dubbo会为每一个 `Filter` 创建并实例化一个 `Invoker` 的匿名内部类，在其 `invoke()` 方法体中执行当前 `Filter` 对象的 `Result invoke(Invoker<?> invoker, Invocation invocation)` 方法， `Filter` 对象所对应的实现类要确保在该方法体内以其两个入参执行类似如下的一个代码片段：

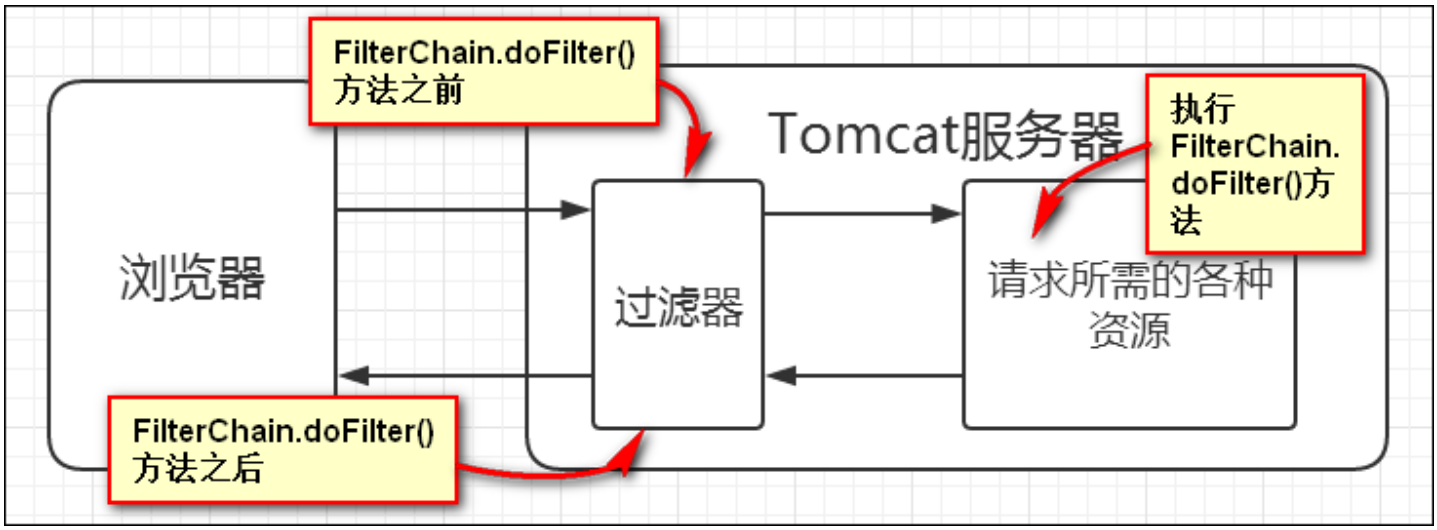
```
...//前验处理或前置特性业务逻辑实现
Result result = invoker.invoke(invocation);
...//后验处理或后置特性业务逻辑实现
return result;
```

不难看出，在当前 Invoker 对象上执行其 invoke(Invocation) 方法，其执行结果取决于其在 Filter 对象上调用 invoke(Invoker<?>, Invocation) 方法传入的首个 Invoker 类型入参。

这里可以认为这个入参是被当前对象所属的匿名内部类给装饰了，如果它也是类似被装饰的 Invoker 类型对象， 那么最后代码执行轨迹就会递归地一直往下调用直到碰到异常或者首次能返回 Result 值的为止，随后便由下往上逐层返回这个结果。

可以看出每个 Invoker 对象可以根据当前特性需要决定是先执行自身业务逻辑还是先调用它所装饰的另一个 Invoker 对象的 invoke(Invocation) 方法，也就是说个体上而言，装饰者和被装饰者的逻辑执行顺序是不确定的，但总体而言，经过层层装饰之后形成的递归关系，理解起来感觉比较混乱。然而这种类似AOP的环绕场景，从RPC调用的视觉来看瞬觉廓然开朗，就是每一个 Filter 装饰者间接等价可以根据自身需要决定逻辑代码的执行时刻：1）在发出出站请求之前； 2）在收到入站响应之后；3）上述两个时刻。

熟悉Servlet过滤器实现原理的童鞋此时定会心领神会，笑意舒展，直观如下图所示：



概括下：**在顺序上越排在前面的 Filter，其前置逻辑越先执行，而后置逻辑则越后执行。**

拦截器调度源码

弄懂机制后，读源码就比较轻松了，总体实现代码如下：

```java
private static <T> Invoker<T> buildInvokerChain(final Invoker<T> invoker, String key,
String group) {
    Invoker<T> last = invoker;
    List<Filter> filters = ExtensionLoader.getExtensionLoader(Filter.class)
        .getActivateExtension(invoker.getUrl(), key, group);

    if (!filters.isEmpty()) {
        for (int i = filters.size() - 1; i >= 0; i--) {
            final Filter filter = filters.get(i);
            final Invoker<T> next = last;
            last = new Invoker<T>() {

                @Override
                public Result invoke(Invocation invocation) throws RpcException {
                    Result asyncResult;
                    try {
                        asyncResult = filter.invoke(next, invocation);
                    } catch (Exception e) {
                        // onError callback
                        if (filter instanceof ListenableFilter) {
                            Filter.Listener listener = ((ListenableFilter)
filter).listener();
                            if (listener != null) {
                                listener.onError(e, invoker, invocation);
                            }
                        }
                        throw e;
                    }
                    return asyncResult;
                }
                ...
            };
        }
    }
    return new CallbackRegistrationInvoker<>(last, filters);
}
```

上述这段代码中，有几个需要注意的地方：

1. 入参 `invoker` 对象是作为首个被装饰者出现的，也即它是实际处理RPC调用的微服务实例或者微服务引用实例，使用 `getActivateExtension` 获取到的所有 `filters` 已经按照优先级排好序，越高的越靠近底层的RPC调用。
2. `Invocation` 类型入参基本不会发生变化，是伴随整个拦截链的，隐含的意思是， `Fitler` 实现可以根据需要在其本地参数容器存入相应的键值对，让其在链中传播，供其他 `Fitler` 实现联动逻辑。这种特性也适用于可携带本地参数容器的 `Result` 。
3. `catch 代码块表示，拦截链中包括当前节点在内的其它前驱中某个 Filter 两个阶段都有可能会发生处理出现了异常，并且显示地Throw出来了，若这些 Filter 属监听型，则回调其 onError() ，通知异常发生。另外异常也可以由 Result``携带返回，这后面一种类型的异常处理，整个拦截链依然可以无感知地继续work。

另外上述有关 Invoker 内部类实现逻辑中省略了如下的代码段，结合上述代码，不难理解最初被装饰的那个 invoker 对象始终是业务逻辑运行的主战场，其它环绕它执行的 Invoker 是独立于业务逻辑之外的增强和补充，因而链上的所有Invoker节点都使用 invoker 获取相关状态。

JAVA

```
new Invoker<T>() {

    @Override
    public Class<T> getInterface() {
        return invoker.getInterface();
    }

    @Override
    public URL getUrl() {
        return invoker.getUrl();
    }

    @Override
    public boolean isAvailable() {
        return invoker.isAvailable();
    }
    @Override
    public void destroy() {
        invoker.destroy();
    }

    @Override
    public String toString() {
        return invoker.toString();
    }
    ...
}
```

Filter 共享反馈结果

拦截器实现中，并非所有内部 Invoker 装饰者等价于Fiter实现会回调 onError() ，结果正常时也不会被回调 onResponse() ，如果异常结果携带在出参 Result 中，这些回调就根本不会发生。而特性上要求所有加入到链中的Filter，只要有要求均能在有结果包括Exception获得通知。因此在上一章节中代码片段中出现了 return new CallbackRegistrationInvoker<>(last, filters) ，它表示*last*这个 Invoker 对象最后又被装饰了一次，目的是让链所有的 ListenableFilter 能通过回调感知到处理结果，如下述源码所示：

JAVA

```java
static class CallbackRegistrationInvoker<T> implements Invoker<T> {

    private final Invoker<T> filterInvoker;
    private final List<Filter> filters;

    public CallbackRegistrationInvoker(Invoker<T> filterInvoker, List<Filter> filters)
{
        this.filterInvoker = filterInvoker;
        this.filters = filters;
    }

    @Override
    public Result invoke(Invocation invocation) throws RpcException {
        Result asyncResult = filterInvoker.invoke(invocation);

        asyncResult = asyncResult.whenCompleteWithContext((r, t) -> {
            for (int i = filters.size() - 1; i >= 0; i--) {
                Filter filter = filters.get(i);
                // onResponse callback
                if (filter instanceof ListenableFilter) {
                    Filter.Listener listener = ((ListenableFilter) filter).listener();
                    if (listener != null) {
                        if (t == null) {
                            listener.onResponse(r, filterInvoker, invocation);
                        } else {
                            listener.onError(t, filterInvoker, invocation);
                        }
                    }
                } else {
                    filter.onResponse(r, filterInvoker, invocation);
                }
            }
        });
        return asyncResult;
    }
    ...//类似上一章节最后呈现的那段代码
}
```

细究两处异常处理实现，如果前者先发生，后者是没有机会执行的，可以认为前者是短路型异常处理。原因是 `CallbackRegistrationInvoker` 是最后一个被执行的 `Invoker` 装饰者对象。不论是哪种方案，`onError`的回调时机都处于拦截链回退的途中。

完结