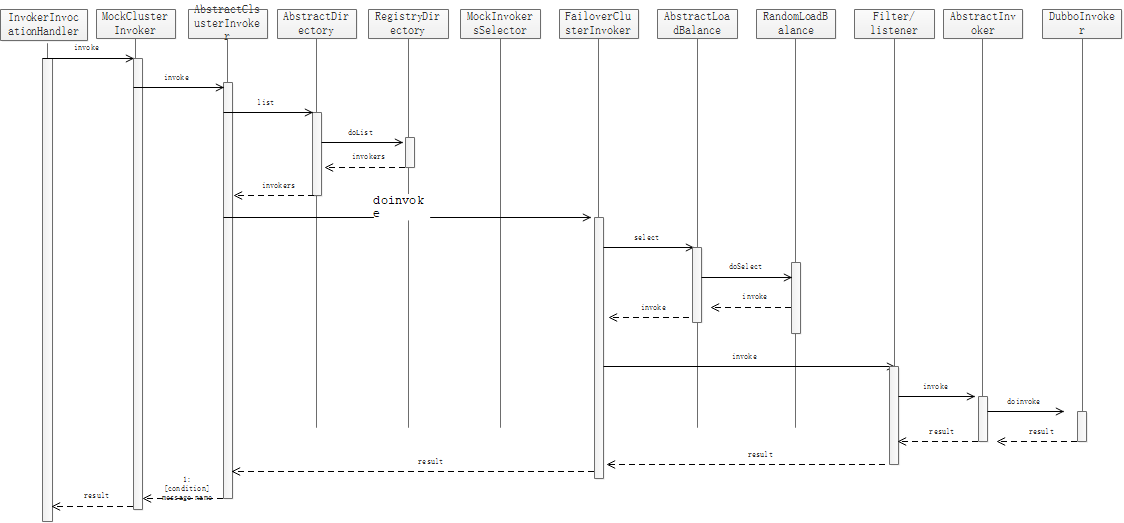


# 咕泡学院VIP课：Dubbo核心源码分析Dubbo(2.5.4)

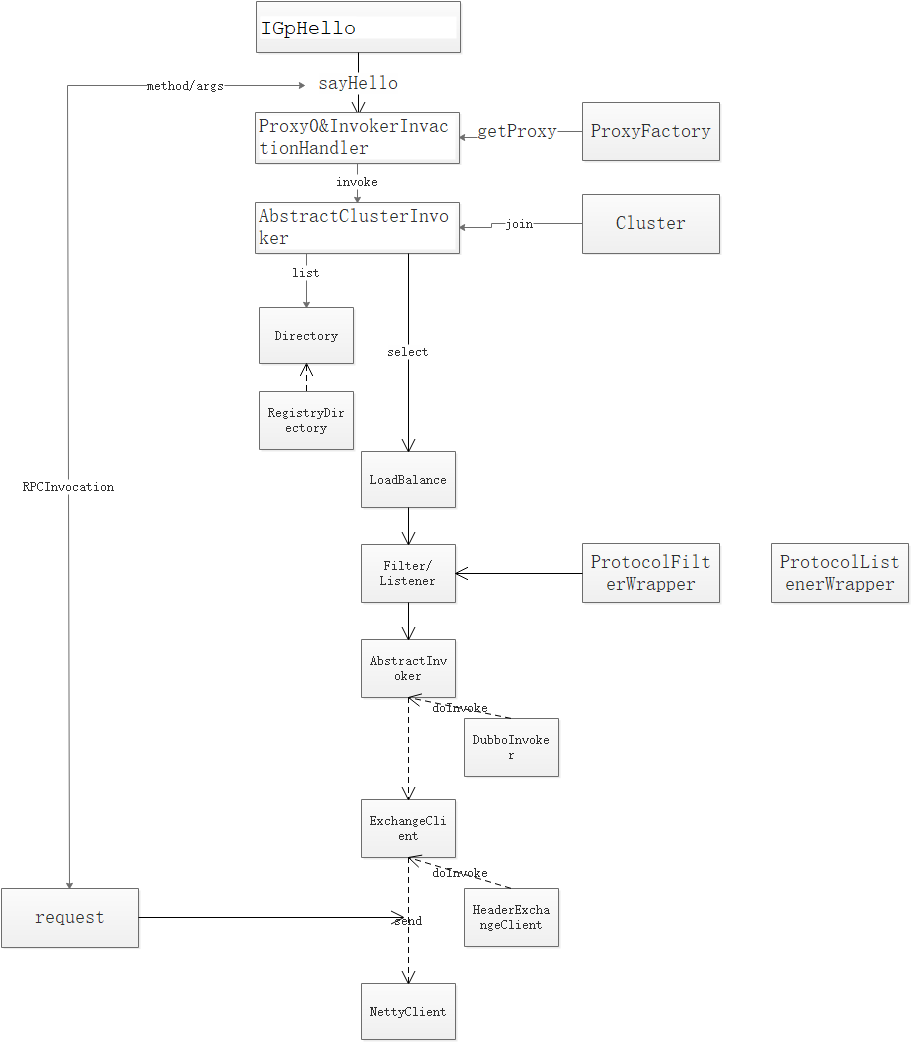
课程目标

1. 消费端发送消息流程
2. 服务消费端消息流程
3. Directory目录服务
4. LoadBalance负载均衡
5. Dubbo课程总结

# 消费端调用过程流程图



# 消费端的调用过程



# **服务端接收消息处理过程**

## NettyHandler**.** messageReceived

接收消息的时候，通过NettyHandler.messageReceived作为入口。

|  |
| --- |
| @Override **public void** messageReceived(ChannelHandlerContext ctx, MessageEvent e) **throws** Exception {  NettyChannel channel = NettyChannel.*getOrAddChannel*(ctx.getChannel(), **url**, **handler**);  **try** {  **handler**.received(channel, e.getMessage());  } **finally** {  NettyChannel.*removeChannelIfDisconnected*(ctx.getChannel());  } } |

## **handler.received**

这个handler是什么呢？还记得在服务发布的时候，组装了一系列的handler吗？代码如下

HeaderExchanger.bind

|  |
| --- |
| **public** ExchangeServer bind(URL url, ExchangeHandler handler) **throws** RemotingException {  **return new** HeaderExchangeServer(Transporters.*bind*(url, **new** DecodeHandler(**new** HeaderExchangeHandler(handler)))); } |

NettyServer

接着又在Nettyserver中，wrap了多个handler

|  |
| --- |
| **public** NettyServer(URL url, ChannelHandler handler) **throws** RemotingException {  **super**(url, ChannelHandlers.*wrap*(handler, ExecutorUtil.*setThreadName*(url, ***SERVER\_THREAD\_POOL\_NAME***))); }  **protected** ChannelHandler wrapInternal(ChannelHandler handler, URL url) {  **return new** MultiMessageHandler(**new** HeartbeatHandler(ExtensionLoader.*getExtensionLoader*(Dispatcher.**class**)  .getAdaptiveExtension().dispatch(handler, url))); } |

所以服务端的handler处理链为

MultiMessageHandler(HeartbeatHandler(AllChannelHandler(DecodeHandler)))

MultiMessageHandler: 复合消息处理

HeartbeatHandler：心跳消息处理，接收心跳并发送心跳响应

AllChannelHandler：业务线程转化处理器，把接收到的消息封装成ChannelEventRunnable可执行任务给线程池处理

DecodeHandler:业务解码处理器

## **HeaderExchangeHandler.received**

交互层请求响应处理，有三种处理方式

1. handlerRequest，双向请求
2. handler.received 单向请求
3. handleResponse 响应消息

|  |
| --- |
| **public void** received(Channel channel, Object message) **throws** RemotingException {  channel.setAttribute(*KEY\_READ\_TIMESTAMP*, System.*currentTimeMillis*());  ExchangeChannel exchangeChannel = HeaderExchangeChannel.*getOrAddChannel*(channel);  **try** {  **if** (message **instanceof** Request) {  *// handle request.* Request request = (Request) message;  **if** (request.isEvent()) {  handlerEvent(channel, request);  } **else** {  **if** (request.isTwoWay()) {  Response response = handleRequest(exchangeChannel, request);  channel.send(response);  } **else** {  **handler**.received(exchangeChannel, request.getData());  }  }  } **else if** (message **instanceof** Response) {  *handleResponse*(channel, (Response) message);  } **else if** (message **instanceof** String) {  **if** (*isClientSide*(channel)) {  Exception e = **new** Exception(**"Dubbo client can not supported string message: "** + message + **" in channel: "** + channel + **", url: "** + channel.getUrl());  ***logger***.error(e.getMessage(), e);  } **else** {  String echo = **handler**.telnet(channel, (String) message);  **if** (echo != **null** && echo.length() > 0) {  channel.send(echo);  }  }  } **else** {  **handler**.received(exchangeChannel, message);  }  } **finally** {  HeaderExchangeChannel.*removeChannelIfDisconnected*(channel);  } } |

## **handleRequest**

处理请求并返回response

|  |
| --- |
| Response handleRequest(ExchangeChannel channel, Request req) **throws** RemotingException {  Response res = **new** Response(req.getId(), req.getVersion());  **if** (req.isBroken()) {  Object data = req.getData();  String msg;  **if** (data == **null**) msg = **null**;  **else if** (data **instanceof** Throwable) msg = StringUtils.*toString*((Throwable) data);  **else** msg = data.toString();  res.setErrorMessage(**"Fail to decode request due to: "** + msg);  res.setStatus(Response.***BAD\_REQUEST***);   **return** res;  }  *// find handler by message class.* Object msg = req.getData();  **try** {  *// handle data.* Object result = **handler**.reply(channel, msg);  res.setStatus(Response.***OK***);  res.setResult(result);  } **catch** (Throwable e) {  res.setStatus(Response.***SERVICE\_ERROR***);  res.setErrorMessage(StringUtils.*toString*(e));  }  **return** res; } |

## **ExchangeHandlerAdaptive.replay(DubboProtocol)**

调用DubboProtocol中定义的ExchangeHandlerAdapter. reply方法处理消息

|  |
| --- |
| **private** ExchangeHandler **requestHandler** = **new** ExchangeHandlerAdapter() {    **public** Object reply(ExchangeChannel channel, Object message) **throws** RemotingException {  invoker.invoke(inv);  } |

那接下来invoker.invoke会调用哪个类中的方法呢？还记得在RegistryDirectory中发布本地方法的时候，对invoker做的包装吗？通过InvokerDelegete对原本的invoker做了一层包装，而原本的invoker是什么呢？是一个JavassistProxyFactory生成的动态代理吧。所以此处的invoker应该是

Filter(Listener(InvokerDelegete(AbstractProxyInvoker (Wrapper.invokeMethod)))

RegistryDirectory生成invoker的代码如下

|  |
| --- |
| private <T> ExporterChangeableWrapper<T> doLocalExport(final Invoker<T> originInvoker){  String key = getCacheKey(originInvoker);  ExporterChangeableWrapper<T> exporter = (ExporterChangeableWrapper<T>) bounds.get(key);  if (exporter == null) {  synchronized (bounds) {  exporter = (ExporterChangeableWrapper<T>) bounds.get(key);  if (exporter == null) {  final Invoker<?> invokerDelegete = new InvokerDelegete<T>(originInvoker, getProviderUrl(originInvoker));  exporter = new ExporterChangeableWrapper<T>((Exporter<T>)protocol.export(invokerDelegete), originInvoker);  bounds.put(key, exporter);  }  }  }  return (ExporterChangeableWrapper<T>) exporter; } |

# Directory

集群目录服务Directory， 代表多个Invoker, 可以看成List<Invoker>,它的值可能是动态变化的比如注册中心推送变更。集群选择调用服务时通过目录服务找到所有服务

StaticDirectory: 静态目录服务， 它的所有Invoker通过构造函数传入， 服务消费方引用服务的时候， 服务对多注册中心的引用，将Invokers集合直接传入 StaticDirectory构造器，再由Cluster伪装成一个Invoker；StaticDirectory的list方法直接返回所有invoker集合；

RegistryDirectory: 注册目录服务， 它的Invoker集合是从注册中心获取的， 它实现了NotifyListener接口实现了回调接口notify(List<Url>)

## **Directory目录服务的更新过程**

RegistryProtocol.doRefer方法，也就是消费端在初始化的时候，这里涉及到了RegistryDirectory这个类。然后执行cluster.join(directory)方法。这些代码在上节课有分析过。

cluster.join其实就是将Directory中的多个Invoker伪装成一个Invoker, 对上层透明，包含集群的容错机制

|  |
| --- |
| **private** <T> Invoker<T> doRefer(Cluster cluster, Registry registry, Class<T> type, URL url) {  RegistryDirectory<T> directory = **new** RegistryDirectory<T>(type, url);*//对多个invoker进行组装* directory.setRegistry(registry); *//ZookeeperRegistry* directory.setProtocol(**protocol**); *//protocol=Protocol$Adaptive  //url=consumer://192.168.111....* URL subscribeUrl = **new** URL(Constants.***CONSUMER\_PROTOCOL***, NetUtils.*getLocalHost*(), 0, type.getName(), directory.getUrl().getParameters());  *//会把consumer://192... 注册到注册中心* **if** (! Constants.***ANY\_VALUE***.equals(url.getServiceInterface())  && url.getParameter(Constants.***REGISTER\_KEY***, **true**)) {  *//zkClient.create()* registry.register(subscribeUrl.addParameters(Constants.***CATEGORY\_KEY***, Constants.***CONSUMERS\_CATEGORY***,  Constants.***CHECK\_KEY***, String.*valueOf*(**false**)));  }  directory.subscribe(subscribeUrl.addParameter(Constants.***CATEGORY\_KEY***,   Constants.***PROVIDERS\_CATEGORY*** + **","** + Constants.***CONFIGURATORS\_CATEGORY*** + **","** + Constants.***ROUTERS\_CATEGORY***));  *//Cluster$Adaptive* **return** cluster.join(directory); } |

## **directory.subscribe**

订阅节点的变化，

1. 当zookeeper上指定节点发生变化以后，会通知到RegistryDirectory的notify方法
2. 将url转化为invoker对象

## **调用过程中invokers的使用**

再调用过程中，AbstractClusterInvoker.invoke方法中，

|  |
| --- |
| **public** Result invoke(**final** Invocation invocation) **throws** RpcException {   checkWhetherDestroyed();   LoadBalance loadbalance;    List<Invoker<T>> invokers = list(invocation);   **if** (invokers != **null** && invokers.size() > 0) {  loadbalance = ExtensionLoader.*getExtensionLoader*(LoadBalance.**class**).getExtension(invokers.get(0).getUrl()  .getMethodParameter(invocation.getMethodName(),Constants.***LOADBALANCE\_KEY***, Constants.***DEFAULT\_LOADBALANCE***));  } **else** {  loadbalance = ExtensionLoader.*getExtensionLoader*(LoadBalance.**class**).getExtension(Constants.***DEFAULT\_LOADBALANCE***);  }  RpcUtils.*attachInvocationIdIfAsync*(getUrl(), invocation);  **return** doInvoke(invocation, invokers, loadbalance); } |

## **list方法**

从directory中获得invokers

|  |
| --- |
| **protected** List<Invoker<T>> list(Invocation invocation) **throws** RpcException {  List<Invoker<T>> invokers = **directory**.list(invocation);  **return** invokers; } |

# 负载均衡LoadBalance

LoadBalance负载均衡， 负责从多个 Invokers中选出具体的一个Invoker用于本次调用，调用过程中包含了负载均衡的算法。

## 负载均衡代码访问入口

在AbstractClusterInvoker.invoke中代码如下，通过名称获得指定的扩展点。RandomLoadBalance

|  |
| --- |
| **public** Result invoke(**final** Invocation invocation) **throws** RpcException {   checkWhetherDestroyed();   LoadBalance loadbalance;    List<Invoker<T>> invokers = list(invocation);  **if** (invokers != **null** && invokers.size() > 0) {  loadbalance = ExtensionLoader.*getExtensionLoader*(LoadBalance.**class**).getExtension(invokers.get(0).getUrl()  .getMethodParameter(invocation.getMethodName(),Constants.***LOADBALANCE\_KEY***, Constants.***DEFAULT\_LOADBALANCE***));  } **else** {  loadbalance = ExtensionLoader.*getExtensionLoader*(LoadBalance.**class**).getExtension(Constants.***DEFAULT\_LOADBALANCE***);  }  RpcUtils.*attachInvocationIdIfAsync*(getUrl(), invocation);  **return** doInvoke(invocation, invokers, loadbalance); } |

## AbstractClusterInvoker.doselect

调用LoadBalance.select方法，讲invokers按照指定算法进行负载

|  |
| --- |
| **private** Invoker<T> doselect(LoadBalance loadbalance, Invocation invocation, List<Invoker<T>> invokers, List<Invoker<T>> selected) **throws** RpcException {  **if** (invokers == **null** || invokers.size() == 0)  **return null**;  **if** (invokers.size() == 1)  **return** invokers.get(0);  *// 如果只有两个invoker，退化成轮循* **if** (invokers.size() == 2 && selected != **null** && selected.size() > 0) {  **return** selected.get(0) == invokers.get(0) ? invokers.get(1) : invokers.get(0);  }  Invoker<T> invoker = loadbalance.select(invokers, getUrl(), invocation);    *//如果 selected中包含（优先判断） 或者 不可用&&availablecheck=true 则重试.* **if**( (selected != **null** && selected.contains(invoker))  ||(!invoker.isAvailable() && getUrl()!=**null** && **availablecheck**)){  **try**{  Invoker<T> rinvoker = reselect(loadbalance, invocation, invokers, selected, **availablecheck**);  **if**(rinvoker != **null**){  invoker = rinvoker;  }**else**{  *//看下第一次选的位置，如果不是最后，选+1位置.* **int** index = invokers.indexOf(invoker);  **try**{  *//最后在避免碰撞* invoker = index <invokers.size()-1?invokers.get(index+1) :invoker;  }**catch** (Exception e) {  ***logger***.warn(e.getMessage()+**" may because invokers list dynamic change, ignore."**,e);  }  }  }**catch** (Throwable t){  ***logger***.error(**"clustor relselect fail reason is :"**+t.getMessage() +**" if can not slove ,you can set cluster.availablecheck=false in url"**,t);  }  }  **return** invoker; } |

默认情况下，LoadBalance使用的是Random算法，但是这个随机和我们理解上的随机还是不一样的,因为他还有个概念叫weight(权重)

## RandomLoadBalance

假设有四个集群节点A,B,C,D,对应的权重分别是1,2,3,4,那么请求到A节点的概率就为1/(1+2+3+4) = 10%.B,C,D节点依次类推为20%,30%,40%.

|  |
| --- |
| **protected** <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {  **int** length = invokers.size(); *// 总个数* **int** totalWeight = 0; *// 总权重* **boolean** sameWeight = **true**; *// 权重是否都一样* **for** (**int** i = 0; i < length; i++) {  **int** weight = getWeight(invokers.get(i), invocation);  totalWeight += weight; *// 累计总权重* **if** (sameWeight && i > 0  && weight != getWeight(invokers.get(i - 1), invocation)) {  sameWeight = **false**; *// 计算所有权重是否一样* }  }  **if** (totalWeight > 0 && ! sameWeight) {  *// 如果权重不相同且权重大于0则按总权重数随机* **int** offset = **random**.nextInt(totalWeight);  *// 并确定随机值落在哪个片断上* **for** (**int** i = 0; i < length; i++) {  offset -= getWeight(invokers.get(i), invocation);  **if** (offset < 0) {  **return** invokers.get(i);  }  }  }  *// 如果权重相同或权重为0则均等随机* **return** invokers.get(**random**.nextInt(length)); } |

# 总结

1. dubbo的发展和使用
2. 基于Dubbo的RPC调用
3. zookeeper+dubbo的实现原理
4. Dubbo的常用配置（多协议、多注册中心、启动时检查、重试、LoadBalance）
5. Dubbo的容器支持
6. 主机绑定
7. 集群容错
8. Dubbo的服务降级
9. SPI和Dubbo ExtensionLoader
10. ExtensionLoade的源码分析
11. Dubbo的服务发布
12. Dubbo的消费端初始化
13. 什么是Invoker
14. 消费端的消费调用链
15. 服务端的处理消息的调用链
16. 什么是Directory
17. 什么是LoadBalance
18. Cluster