

第七节: Dubbo服务调用源码解析

课程内容

笔记更新地址:

服务消费端执行逻辑

服务提供端执行逻辑

Dubbo的异常处理

课程内容

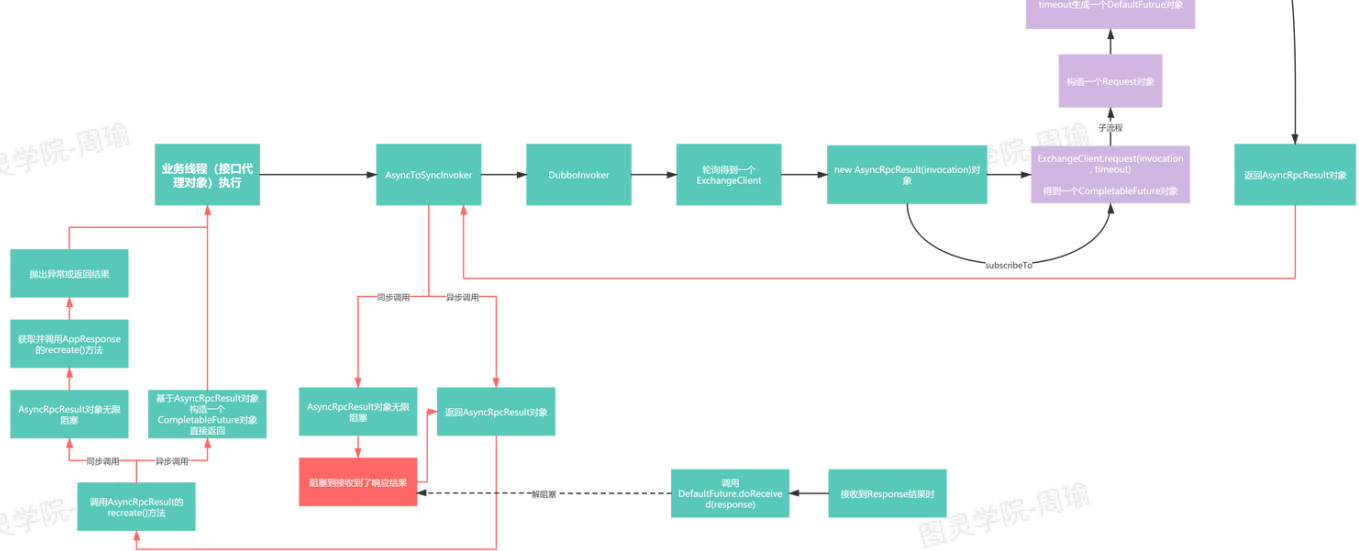
1. 服务提供者处理请求流程解析
2. 服务消费发送请求流程解析
3. 服务提供者端线程模型解析
4. 服务消费者端线程模型解析

笔记更新地址:

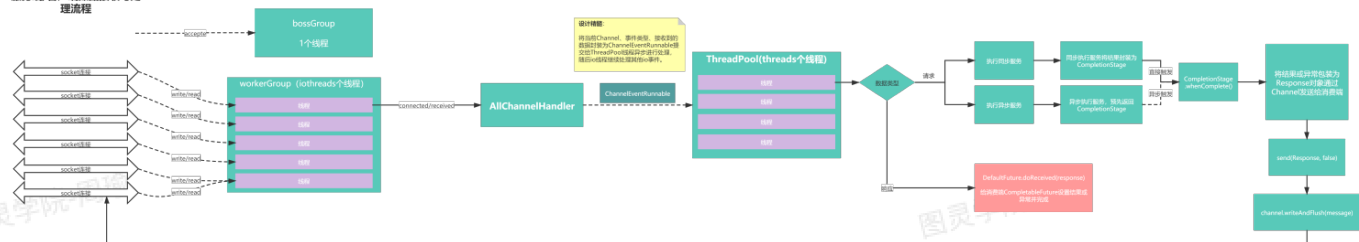
<https://www.yuque.com/books/share/f2394ae6-381b-4f44-819e-c231b39c1497> (密码: kyys)

《Dubbo笔记》

消费端发送请求线程模型



服务端/客户端数据接收与处理流程



processon链接: <https://www.processon.com/view/link/60110b847d9c08426cf10e49>

服务消费端执行逻辑

1. `MockClusterInvoker.invoke(new RpcInvocation(method, args))`: Mock逻辑
2. `AbstractClusterInvoker.invoke(invocation)`: 把`RpcContext`中设置的`Attachments`添加到`invocation`对象上, 调用路由链从服务目录上筛选出适合的服务Invoker, 获得服务均衡策略`loadbalance`
3. `FailoverClusterInvoker.doInvoke(invocation, invokers, loadbalance)`: 根据负载均衡策略选出一个Invoker, 然后执行
4. `InvokerWrapper.invoke(invocation)`: 没做什么事情
5. `CallbackRegistrationInvoker.invoke(invocation)`: 开始执行Filter链, 执行完得到结果后, 会获取`ListenableFilter`中的`listener`, 执行`listener`的`onResponse`方法
6. `ConsumerContextFilter.invoke(invocation)`: 设置`RpcContext`中`LocalAddress`、`RemoteAddress`、`RemoteApplicationName`参数
7. `FutureFilter.invoke(invocation)`:
8. `MonitorFilter.invoke(invocation)`: 方法的执行次数+1
9. `ListenerInvokerWrapper.invoke(invocation)`: 没做什么事情
10. `AsyncToSyncInvoker.invoke(invocation)`: 异步转同步, 会先用下层Invoker去异步执行, 然后阻塞`Integer.MAX_VALUE`时间, 直到拿到了结果
11. `AbstractInvoker.invoke(invocation)`: 主要调用`DubboInvoker`的`doInvoke`方法, 如果`doInvoker`方法出现了异常, 会进行包装, 包装成`AsyncRpcResult`
12. `DubboInvoker.doInvoke(invocation)`: 从`clients`轮询出一个client进行数据发送, 如果配置了不关心结果, 则调用`ReferenceCountExchangeClient`的`send`方法, 否则调用`ReferenceCountExchangeClient`的`request`方法
13. `ReferenceCountExchangeClient.request(Object request, int timeout)`: 没做什么事情
14. `HeaderExchangeClient.request(Object request, int timeout)`: 没做什么事情
15. `HeaderExchangeChannel.request(Object request, int timeout)`: 构造一个`Request`对象, 并且会构造一个`DefaultFuture`对象来阻塞`timeout`的时间来等待结果, 在构造`DefaultFuture`对象时, 会把`DefaultFuture`对象和`req`的`id`存入`FUTURES`中, `FUTURES`是一个`Map`, 当`HeaderExchangeHandler`接收到结果时, 会从这个`Map`中根据`id`获取到`DefaultFuture`对象, 然后返回`Response`。
16. `AbstractPeer.send(Object message)`: 从`url`中获取`send`参数, 默认为`false`
17. `AbstractClient.send(Object message, boolean sent)`: 没做什么
18. `NettyChannel.send(Object message, boolean sent)`: 调用`NioSocketChannel`的`writeAndFlush`发送数据, 然后判断`send`如果是`true`, 那么则阻塞`url`中指定的`timeout`时间, 因为如果`send`是`false`, 在`HeaderExchangeChannel`中会阻塞`timeout`时间
19. `NioSocketChannel.writeAndFlush(Object msg)`: 最底层的`Netty`非阻塞式的发送数据

总结一下上面调用流程:

1. 最外层是Mock逻辑, 调用前, 调用后进行Mock
2. 从服务目录中, 根据当前调用的方法和路由链, 筛选出部分服务Invoker (`DubboInvoker`)
3. 对服务Invoker进行负载均衡, 选出一个服务Invoker
4. 执行Filter链

5. AsyncToSyncInvoker完成异步转同步，因为DubboInvoker的执行是异步非阻塞的，所以如果是同步调用，则会在此处阻塞，知道拿到响应结果
6. DubboInvoker开始异步非阻塞的调用
7. HeaderExchangeChannel中会阻塞timeout的时间来等待结果，该timeout就是用户在消费端所配置的timeout

服务提供端执行逻辑

1. NettyServerHandler：接收数据
2. MultiMessageHandler：判断接收到的数据是否是MultiMessage，如果是则获取MultiMessage中的单个Message，传递给HeartbeatHandler进行处理
3. HeartbeatHandler：判断是不是心跳消息，如果是则把Message传递给AllChannelHandler
4. AllChannelHandler：把接收到的Message封装为一个ChannelEventRunnable对象，扔给线程池进行处理
5. ChannelEventRunnable：在ChannelEventRunnable的run方法中会调用DecodeHandler处理Message
6. DecodeHandler：按Dubbo协议的数据格式，解析当前请求的path，version，方法，方法参数等等，然后把解析好了的请求交给HeaderExchangeHandler
7. HeaderExchangeHandler：处理Request数据，首先构造一个Response对象，然后调用ExchangeHandlerAdapter得到一个CompletionStage future，然后给future通过whenComplete绑定一个回调函数，当future执行完了之后，就可以从回调函数中得到ExchangeHandlerAdapter的执行结果，并把执行结果设置给Response对象，通过channel发送出去。
8. ExchangeHandlerAdapter：从本机已经导出的Exporter中根据当前Request所对应的服务key，去寻找Exporter对象，从Exporter中得到Invoker，然后执行invoke方法，此Invoker为ProtocolFilterWrapper\$CallbackRegistrationInvoker
9. ProtocolFilterWrapper\$CallbackRegistrationInvoker：负责执行过滤器链，并且在执行完了之后回调每个过滤器的onResponse或onError方法
10. EchoFilter：判断当前请求是不是一个回升测试，如果是，则不继续执行过滤器链了（服务实现者Invoker也不会调用了）
11. ClassLoaderFilter：设置当前线程的classloader为当前要执行的服务接口所对应的classloader
12. GenericFilter：把泛化调用发送过来的信息包装为RpcInvocation对象
13. ContextFilter：设置RpcContext.getContext()的参数
14. TraceFilter：先执行下一个invoker的invoke方法，调用成功后记录调用信息
15. TimeoutFilter：调用时没有特别处理，只是记录了一下当前时间，当整个filter链都执行完了之后回调TimeoutFilter的onResponse方法时，会判断本次调用是否超过了timeout
16. MonitorFilter：记录当前服务的执行次数
17. ExceptionFilter：调用时没有特别处理，在回调onResponse方法时，对不同的异常进行处理，详解Dubbo的异常处理
18. DelegateProviderMetaDataInvoker：过滤器链结束，调用下一个Invoker

19. AbstractProxyInvoker：在服务导出时，根据服务接口，服务实现类对象生成的，它的invoke方法就会执行服务实现类对象的方法，得到结果

Dubbo的异常处理

当服务消费者在调用一个服务时，服务提供者在执行服务逻辑时可能会出现异常，对于Dubbo来说，服务消费者需要在消费端抛出这个异常，那么这个功能是怎么做到的呢？

服务提供者在执行服务时，如果出现了异常，那么框架会把异常捕获，捕获异常的逻辑在AbstractProxyInvoker中，捕获到异常后，会把异常信息包装为正常的AppResponse对象，只是AppResponse的value属性没有值，exception属性有值。

此后，服务提供者会把这个AppResponse对象发送给服务消费端，服务消费端是在InvokerInvocationHandler中调用AppResponse的recreate方法重新得到一个结果，在recreate方法中会去失败AppResponse对象是否正常，也就是是否存在exception信息，如果存在，则直接throw这个exception，从而做到服务执行时出现的异常，在服务消费端抛出。

那么这里存在一个问题，如果服务提供者抛出的异常类，在服务消费者这边不存在，那么服务消费者也就抛不出这个异常了，那么dubbo是怎么处理的呢？

这里就涉及到了ExceptionHandler，它是服务提供者端的一个过滤器，它主要是在服务提供者执行完服务后会去识别异常：

1. 如果是需要开发人员捕获的异常，那么忽略，直接把这个异常返回给消费者
2. 如果在当前所执行的方法签名上有声明，那么忽略，直接把这个异常返回给消费者
3. 如果抛出的异常不需要开发人员捕获，或者方法上没有申明，那么服务端或记录一个error日志
4. 异常类和接口类在同一jar包里，那么忽略，直接把这个异常返回给消费者
5. 如果异常类是JDK自带的异常，那么忽略，直接把这个异常返回给消费者
6. 如果异常类是Dubbo自带的异常，那么忽略，直接把这个异常返回给消费者
7. 否则，把异常信息包装成RuntimeException，并覆盖AppResponse对象中的exception属性