

Dubbo源码解析（四十六）消费端发送请求过程

 java  dubbo 阅读约 40 分钟

2.7大揭秘——消费端发送请求过程

目标：从源码的角度分析一个服务方法调用经历怎么样的磨难以后到达服务端。

前言

前一篇文章讲到的是引用服务的过程，引用服务无非就是创建出一个代理。供消费者调用服务的相关方法。本节将从调用方法开始讲解内部的整个调用链。我们就拿dubbo内部的例子讲。

```
ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("spring/dubbo-consumer.xml");
context.start();
DemoService demoService = context.getBean("demoService", DemoService.class);
String hello = demoService.sayHello("world");
System.out.println("result: " + hello);
```

这是dubbo-demo-xml-consumer内的实例代码。接下来我们就开始来看调用demoService.sayHello方法的时候，dubbo执行了哪些操作。

执行过程

(一) InvokerInvocationHandler的invoke

```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    // 获得方法名称
    String methodName = method.getName();
    // 获得方法参数类型
    Class<?>[] parameterTypes = method.getParameterTypes();
    // 如果该方法所在的类是Object类型，则直接调用invoke。
    if (method.getDeclaringClass() == Object.class) {
        return method.invoke(invoker, args);
    }
    // 如果这个方法是toString，则直接调用invoker.toString()
    if ("toString".equals(methodName) && parameterTypes.length == 0) {
        return invoker.toString();
    }
    // 如果这个方法是hashCode直接调用invoker.hashCode()
    if ("hashCode".equals(methodName) && parameterTypes.length == 0) {
        return invoker.hashCode();
    }
    // 如果这个方法是equals，直接调用invoker.equals(args[0])
    if ("equals".equals(methodName) && parameterTypes.length == 1) {
        return invoker.equals(args[0]);
    }

    // 调用invoke
    return invoker.invoke(new RpcInvocation(method, args)).recreate();
}
```

可以看到上面的源码，首先对Object的方法进行了处理，如果调用的方法不是这些方法，则先会创建RpcInvocation，然后再调用invoke。

RpcInvocation的构造方法

```
public RpcInvocation(Method method, Object[] arguments) {
    this(method.getName(), method.getParameterTypes(), arguments, null, null);
}
```

```
public RpcInvocation(String methodName, Class<?>[] parameterTypes, Object[] arguments, Map<String, String>
attachments, Invoker<?> invoker) {
    // 设置方法名
    this.methodName = methodName;
    // 设置参数类型
    this.parameterTypes = parameterTypes == null ? new Class<?>[0] : parameterTypes;
    // 设置参数
    this.arguments = arguments == null ? new Object[0] : arguments;
    // 设置附加值
    this.attachments = attachments == null ? new HashMap<String, String>() : attachments;
    // 设置invoker实体
    this.invoker = invoker;
}
```

创建完RpcInvocation后，就是调用invoke。先进入的是ListenerInvokerWrapper的invoke。

(二) MockClusterInvoker的invoke

可以参考[《dubbo源码解析（四十一）集群——Mock》](#)的(二)MockClusterInvoker，降级后的返回策略的实现，根据配置的不同来决定不用降级还是强制服务降级还是失败后再服务降级。

(三) AbstractClusterInvoker的invoke

可以参考[《dubbo源码解析（三十五）集群——cluster》](#)的(一)AbstractClusterInvoker，该类是一个抽象类，其中封装了一些公用的方法，AbstractClusterInvoker的invoke也只是做了一些公用操作。主要的逻辑在doInvoke中。

(四) FailoverClusterInvoker的doInvoke

可以参考[《dubbo源码解析（三十五）集群——cluster》](#)的(十二)FailoverClusterInvoker，该类实现了失败重试的容错策略。

(五) InvokerWrapper的invoke

可以参考[《dubbo源码解析（二十二）远程调用——Protocol》](#)的(五)InvokerWrapper。该类用了装饰模式，不过并没有实现实际的功能增强。

(六) ProtocolFilterWrapper的内部类CallbackRegistrationInvoker的invoke

```

public Result invoke(Invocation invocation) throws RpcException {
    // 调用拦截器链的invoke
    Result asyncResult = filterInvoker.invoke(invocation);

    // 把异步返回的结果加入到上下文中
    asyncResult.thenApplyWithContext(r -> {
        // 循环各个过滤器
        for (int i = filters.size() - 1; i >= 0; i--) {
            Filter filter = filters.get(i);
            // onResponse callback
            // 如果该过滤器是ListenableFilter类型的
            if (filter instanceof ListenableFilter) {
                // 强制类型转化
                Filter.Listener listener = ((ListenableFilter) filter).listener();
                if (listener != null) {
                    // 如果内部类Listener不为空，则调用回调方法onResponse
                    listener.onResponse(r, filterInvoker, invocation);
                }
            } else {
                // 否则，直接调用filter的onResponse，做兼容。
                filter.onResponse(r, filterInvoker, invocation);
            }
        }
        // 返回异步结果
        return r;
    });
}

```

这里看到先是调用拦截器链的invoke方法。下面的逻辑是把异步返回的结果放到上下文中，具体的ListenableFilter以及内部类的设计，还有thenApplyWithContext等方法我会在异步的实现中讲到。

(七) ProtocolFilterWrapper的buildInvokerChain方法中的invoker实例的invoke方法。

```

public Result invoke(Invocation invocation) throws RpcException {
    Result asyncResult;
    try {
        // 依次调用各个过滤器，获得最终的返回结果
        asyncResult = filter.invoke(next, invocation);
    } catch (Exception e) {
        // onError callback
        // 捕获异常，如果该过滤器是ListenableFilter类型的
        if (filter instanceof ListenableFilter) {
            // 获得内部类Listener
            Filter.Listener listener = ((ListenableFilter) filter).listener();
            if (listener != null) {
                // 调用onError，回调错误信息
                listener.onError(e, invoker, invocation);
            }
        }
        // 抛出异常
        throw e;
    }
    // 返回结果
    return asyncResult;
}

```

该方法中是对异常的捕获，调用内部类Listener的onError来回调错误信息。接下来看它经过了哪些拦截器。

(八) ConsumerContextFilter的invoke

```

public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
    // 获得上下文，设置invoker，会话域，本地地址和原创地址
    RpcContext.getContext()
        .setInvoker(invoker)
        .setInvocation(invocation)
        .setLocalAddress(NetUtils.getLocalHost(), 0)
        .setRemoteAddress(invoker.getUrl().getHost(),
            invoker.getUrl().getPort());
    // 如果会话域是RpcInvocation，则设置invoker
    if (invocation instanceof RpcInvocation) {
        ((RpcInvocation) invocation).setInvoker(invoker);
    }
    try {
        // 移除服务端的上下文
        RpcContext.removeServerContext();
        // 调用下一个过滤器
        return invoker.invoke(invocation);
    } finally {
        // 清空上下文
        RpcContext.removeContext();
    }
}

```

```

static class ConsumerContextListener implements Listener {
    @Override
    public void onResponse(Result appResponse, Invoker<?> invoker, Invocation invocation) {
        // 把结果中的附加值放入到上下文中
        RpcContext.getServerContext().setAttachments(appResponse.getAttachments());
    }

    @Override
    public void onError(Throwable t, Invoker<?> invoker, Invocation invocation) {
        // 不做任何处理
    }
}

```

可以参考[《dubbo源码解析（二十）远程调用——Filter》](#)，不过上面的源码是最新的，而链接内的源码是2.6.x的，虽然做了一些变化，比如内部类的设计，后续的过滤器也有同样的实现，但是ConsumerContextFilter作用没有变化，它依旧是在当前的RpcContext中记录本地调用的一次状态信息。该过滤器执行完成后，会回到ProtocolFilterWrapper的invoke中的

```
Result result = filter.invoke(next, invocation);
```

然后继续调用下一个过滤器FutureFilter。

(九) FutureFilter的invoke

```

public Result invoke(final Invoker<?> invoker, final Invocation invocation) throws RpcException {
    // 该方法是真正的调用方法的执行
    fireInvokeCallback(invoker, invocation);
    // need to configure if there's return value before the invocation in order to help invoker to judge if it's
    // necessary to return future.
    return invoker.invoke(invocation);
}

```

```

class FutureListener implements Listener {
    @Override
    public void onResponse(Result result, Invoker<?> invoker, Invocation invocation) {
        if (result.hasException()) {
            // 处理异常结果
            fireThrowCallback(invoker, invocation, result.getException());
        } else {
            // 处理正常结果
            fireReturnCallback(invoker, invocation, result.getValue());
        }
    }

    @Override
    public void onError(Throwable t, Invoker<?> invoker, Invocation invocation) {
    }
}

```

可以参考[《dubbo源码解析（二十四）远程调用——dubbo协议》](#)中的（十四）FutureFilter，其中会有部分结构不一样，跟ConsumerContextFilter一样，因为后续版本对Filter接口进行了新的设计，增加了onResponse方法，把返回的执行逻辑放到onResponse中去了。其他逻辑没有很大变化。等该过滤器执行完成后，还是回到ProtocolFilterWrapper的invoke中的，继续调用下一个过滤器MonitorFilter。

(十) MonitorFilter的invoke

```

public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
    // 如果开启监控
    if (invoker.getUrl().hasParameter(MONITOR_KEY)) {
        // 设置监控开始时间
        invocation.setAttachment(MONITOR_FILTER_START_TIME, String.valueOf(System.currentTimeMillis()));
        // 获得当前的调用数，并且增加
        getConcurrent(invoker, invocation).incrementAndGet(); // count up
    }
    return invoker.invoke(invocation); // proceed invocation chain
}

```

```

class MonitorListener implements Listener {

    @Override
    public void onResponse(Result result, Invoker<?> invoker, Invocation invocation) {
        // 如果开启监控
        if (invoker.getUrl().hasParameter(MONITOR_KEY)) {
            // 执行监控，搜集数据
            collect(invoker, invocation, result, RpcContext.getContext().getRemoteHost(),
                    Long.valueOf(invocation.getAttachment(MONITOR_FILTER_START_TIME)), false);
            // 减少当前调用数
            getConcurrent(invoker, invocation).decrementAndGet(); // count down
        }
    }

    @Override
    public void onError(Throwable t, Invoker<?> invoker, Invocation invocation) {
        // 如果开启监控
        if (invoker.getUrl().hasParameter(MONITOR_KEY)) {
            // 执行监控，搜集数据
            collect(invoker, invocation, null, RpcContext.getContext().getRemoteHost(),
                    Long.valueOf(invocation.getAttachment(MONITOR_FILTER_START_TIME)), true);
            // 减少当前调用数
            getConcurrent(invoker, invocation).decrementAndGet(); // count down
        }
    }
}

```

可以看到该过滤器实际用来做监控，监控服务的调用数量等。其中监控的逻辑不是本文重点，所以不细讲。接下来调用的是ListenerInvokerWrapper的invoke。

(十一) ListenerInvokerWrapper的invoke

```
public Result invoke(Invocation invocation) throws RpcException {
    return invoker.invoke(invocation);
}
```

可以参考[《dubbo源码解析（二十一）远程调用——Listener》](#)，这里用到了装饰者模式，直接调用了invoker。该类里面做了服务启动的监听器。我们直接关注下一个invoke。

(十二) AsyncToSyncInvoker的invoke

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

    try {
        // 如果是同步的调用
        if (InvokeMode.SYNC == ((RpcInvocation)invocation).getInvokeMode()) {
            // 从异步结果中get结果
            asyncResult.get();
        }
    } catch (InterruptedException e) {
        throw new RpcException("Interrupted unexpectedly while waiting for remoting result to return! method: "
+ invocation.getMethodName() + ", provider: " + getUrl() + ", cause: " + e.getMessage(), e);
    } catch (ExecutionException e) {
        Throwable t = e.getCause();
        if (t instanceof TimeoutException) {
            throw new RpcException(RpcException.TIMEOUT_EXCEPTION, "Invoke remote method timeout. method: " +
invocation.getMethodName() + ", provider: " + getUrl() + ", cause: " + e.getMessage(), e);
        } else if (t instanceof RemotingException) {
            throw new RpcException(RpcException.NETWORK_EXCEPTION, "Failed to invoke remote method: " +
invocation.getMethodName() + ", provider: " + getUrl() + ", cause: " + e.getMessage(), e);
        }
    } catch (Throwable e) {
        throw new RpcException(e.getMessage(), e);
    }
    // 返回异步结果
    return asyncResult;
}
```

AsyncToSyncInvoker类从名字上就很好理解，它的作用是把异步结果转化为同步结果。新的改动中每个调用只要不是oneway方式调用都会先以异步调用开始，然后根据配置的情况如果是同步调用，则会在这个类中进行异步结果转同步的处理。当然，这里先是执行了invoke，然后就进入下一个AbstractInvoker的invoke了。

(十三) AbstractInvoker的invoke

```

public Result invoke(Invocation inv) throws RpcException {
    // if invoker is destroyed due to address refresh from registry, let's allow the current invoke to proceed
    // 如果服务引用销毁，则打印告警日志，但是通过
    if (destroyed.get()) {
        logger.warn("Invoker for service " + this + " on consumer " + NetUtils.getLocalHost() + " is destroyed,
            + ", dubbo version is " + Version.getVersion() + ", this invoker should not be used any longer");
    }
    RpcInvocation invocation = (RpcInvocation) inv;
    // 会话域中加入该调用链
    invocation.setInvoker(this);
    // 把附加值放入会话域
    if (CollectionUtils.isNotEmptyMap(attachment)) {
        invocation.addAttachmentsIfAbsent(attachment);
    }
    // 把上下文的附加值放入会话域
    Map<String, String> contextAttachments = RpcContext.getContext().getAttachments();
    if (CollectionUtils.isNotEmptyMap(contextAttachments)) {
        /**
         * invocation.addAttachmentsIfAbsent(context){@Link RpcInvocation#addAttachmentsIfAbsent(Map)}should not
         * be used here,
         * because the {@link RpcContext#setAttachment(String, String)} is passed in the Filter when the call is
         * triggered
         * by the built-in retry mechanism of the Dubbo. The attachment to update RpcContext will no longer work,
         * which is
         * a mistake in most cases (for example, through Filter to RpcContext output traceId and spanId and other
         * information).
    }

```

可以参考[《dubbo源码解析（二十二）远程调用——Protocol》](#)的（三）AbstractInvoker。该方法做了一些公共的操作，比如服务引用销毁的检测，加入附加值，加入调用链实体域到会话域中等。然后执行了doInvoke抽象方法。各协议自己去实现。然后就是执行到doInvoke方法了。使用的协议不一样，doInvoke的逻辑也有所不同，我这里举的例子是使用dubbo协议，所以我就介绍DubboInvoker的doInvoke，其他自行查看具体的实现。此次的异步改造加入了InvokeMode，我会在后续中介绍这个。

（十四）DubboInvoker的doInvoke

```

protected Result doInvoke(final Invocation invocation) throws Throwable {
    // rpc会话域
    RpcInvocation inv = (RpcInvocation) invocation;
    // 获得方法名
    final String methodName = RpcUtils.getMethodName(invocation);
    // 把path放入到附加值中
    inv.setAttachment(PATH_KEY, getUrl().getPath());
    // 把版本号放入到附加值
    inv.setAttachment(VERSION_KEY, version);

    // 当前的客户端
    ExchangeClient currentClient;
    // 如果数组内就一个客户端，则直接取出
    if (clients.length == 1) {
        currentClient = clients[0];
    } else {
        // 取模轮询 从数组中取，当取到最后一个时，从头开始
        currentClient = clients[index.getAndIncrement() % clients.length];
    }
    try {
        // 是否是单向发送
        boolean isOneway = RpcUtils.isOneway(getUrl(), invocation);
        // 获得超时时间
        int timeout = getUrl().getMethodParameter(methodName, TIMEOUT_KEY, DEFAULT_TIMEOUT);
        // 如果是单向发送
        if (isOneway) {

```

可以参考[《dubbo源码解析（二十四）远程调用——dubbo协议》](#)的（一）DubboInvoker，不过链接内的文章的源码是2.6.x版本的，而上述的源码是最新版本的，其中就有对于异步的改动，比如加入了异步返回结果、除了单向调用，一律都先处理成AsyncRpcResult等。具体的AsyncRpcResult以及其中用到的CompletableFuture我会在下文介绍。

上述源码中执行currentClient.request或者currentClient.send，代表把请求放入channel中，交给channel来处理请求。最后来看一个currentClient.request，因为这其中涉及到了Future的构建。

(十五) ReferenceCountExchangeClient的request

```
public CompletableFuture<Object> request(Object request, int timeout) throws RemotingException {  
    return client.request(request, timeout);  
}
```

ReferenceCountExchangeClient是一个记录请求数的类，用了适配器模式，对ExchangeClient做了功能增强。

可以参考[《dubbo源码解析（二十四）远程调用——dubbo协议》](#)的（八）ReferenceCountExchangeClient。

(十六) HeaderExchangeClient的request

```
public CompletableFuture<Object> request(Object request, int timeout) throws RemotingException {  
    return channel.request(request, timeout);  
}
```

该类也是用了适配器模式，该类主要的作用就是增加了心跳功能，可以参考[《dubbo源码解析（十）远程通信——Exchange层》](#)的（四）HeaderExchangeClient。然后进入HeaderExchangeChannel的request。

(十七) HeaderExchangeChannel的request

可以参考[《dubbo源码解析（十）远程通信——Exchange层》](#)的（二）HeaderExchangeChannel，在这个request方法中就可以看到

```
// 创建DefaultFuture对象，可以从future中主动获得请求对应的响应信息  
DefaultFuture future = new DefaultFuture(channel, req, timeout);
```

生成了需要的future。异步请求结果就是从这个future中获取。关于DefaultFuture也可以参考[《dubbo源码解析（十）远程通信——Exchange层》](#)的（七）DefaultFuture。

后面channel.send方法就是跟远程通信有关了，例如使用netty作为通信实现，则会使用netty实现的客户端进行通信。

(十八) AbstractPeer的send

可以参考[《dubbo源码解析（九）远程通信——Transport层》](#)的（一）AbstractPeer，其中send方法比较简单，根据sent配置项去做消息发送。接下来看AbstractClient的send

(十九) AbstractClient的send

可以参考[《dubbo源码解析（九）远程通信——Transport层》](#)的（四）AbstractClient。

```
public void send(Object message, boolean sent) throws RemotingException {  
    // 如果需要重连或者没有链接，则连接  
    if (needReconnect && !isConnected()) {  
        connect();  
    }  
    // 获得通道  
    Channel channel = getChannel();  
    //TODO Can the value returned by getChannel() be null? need improvement.  
    if (channel == null || !channel.isConnected()) {  
        throw new RemotingException(this, "message can not send, because channel is closed . url:" + getUrl());  
    }  
    // 通过通道发送消息  
    channel.send(message, sent);  
}
```

该方法中做了重连的逻辑，然后就是通过通道发送消息，dubbo有几种通信的实现，我这里就按照默认的netty4实现来讲解，所以下一步走到了NettyChannel的send。

(二十) NettyChannel的send

可以参考[《dubbo源码解析（十七）远程通信——Netty4》](#)的（一）NettyChannel。这里其中先执行了下面父类AbstractChannel的send，检查了一下通道是否关闭，然后再走下面的逻辑。当执行writeAndFlush方法后，消息就被发送。

dubbo数据包可以查看[《dubbo源码解析（十）远程通信——Exchange层》](#)的（二十五）ExchangeCodec，后续关于netty发送消息，以及netty出站数据在发出之前还需要进行编码操作我就先不做介绍，主要是跟netty知识点强相关，只是dubbo做了一些自己的编码，以及集成了各类序列化方式。

后记

该文章讲解了dubbo调用服务的方法所经历的所有步骤，直到调用消息发送到服务端为止，是目前最新代码的解析。下一篇文将讲解服务端收到方法调用的请求后，如何处理以及如何把调用结果返回的过程。

阅读 752 · 更新于 11月8日

赞 2

收藏 2

赞赏

分享

本作品系原创，作者保留所有权利，未经作者允许，禁止转载和演绎



crazyhzm

265

关注作者

0 条评论

得票 · 时间



撰写评论 ...

提交评论

推荐阅读

[dubbo源码解析（二）Dubbo扩展机制SPI](#)

前一篇文章《dubbo源码解析（一）Hello,Dubbo》是对dubbo整个项目大体的介绍，而从这篇文章开始，我将会从源码来解读dub...

[CrazyHzm](#) · 阅读 355

[聊聊Dubbo - Dubbo可扩展机制源码解析](#)

摘要：在Dubbo可扩展机制实战中，我们了解了Dubbo扩展机制的一些概念，初探了Dubbo中LoadBalance的实现，并自己实现了...

[猫耳](#) · 阅读 21

[结合Dubbo源码分析Spi](#)

如前所述，DubboSPI的目的是获取一个指定实现类的对象。那么Dubbo是通过什么方式获取的呢？其实是调用ExtensionLoader.ge...

[hnxydq](#) · 阅读 15

[Spring-boot+Dubbo应用启停源码分析](#)

背景介绍DubboSpringBoot工程致力于简化DubboRPC框架在SpringBoot应用场景的开发。同时也整合了SpringBoot特性：你有没...

[阿里云云栖社区](#) · 阅读 13