crazyhzm 265

Dubbo源码解析（四十七）服务端处理请求过程

dubbo java 阅读约 70 分钟

2.7大揭秘——服务端处理请求过程

> 目标：从源码的角度分析服务端接收到请求后的一系列操作，最终把客户端需要的值返回。

前言

上一篇讲到了消费端发送请求的过程，该篇就要将服务端处理请求的过程。也就是当服务端收到请求数据包后的一系列处理以及如何返回最终结果。我们也知道消费端在发送请求的时候已经做了编码，所以我们也需要在服务端接收到数据包后，对协议头和协议体进行解码。不过本篇不讲解如何解码。有兴趣的可以翻翻我以前的文章，有讲到关于解码的逻辑。接下来就开始讲解服务端收到请求后的逻辑。

处理过程

假设远程通信的实现还是用netty4，解码器将数据包解析成 Request 对象后，NettyHandler 的 messageReceived 方法紧接着会收到这个对象，所以第一步就是NettyServerHandler的channelRead。

（一）NettyServerHandler的channelRead

可以参考《dubbo源码解析（十七）远程通信——Netty4》的（三）NettyServerHandler

```
public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
    // 看是否在缓存中命中，如果没有命中，则创建NettyChannel并且缓存。
    NettyChannel channel = NettyChannel.getOrAddChannel(ctx.channel(), url, handler);
    try {
        // 接受消息
        handler.received(channel, msg);
    } finally {
        // 如果通道不活跃或者断掉，则从缓存中清除
        NettyChannel.removeChannelIfDisconnected(ctx.channel());
    }
}
```

NettyServerHandler是基于netty4实现的服务端通道处理实现类，而该方法就是用来接收请求，接下来就是执行AbstractPeer的received。

（二）AbstractPeer的received

可以参考《dubbo源码解析（九）远程通信——Transport层》的（一）AbstractPeer

```
public void received(Channel ch, Object msg) throws RemotingException {
    // 如果通道已经关闭，则直接返回
    if (closed) {
        return;
    }
    handler.received(ch, msg);
}
```

该方法比较简单，之前也讲过AbstractPeer类就做了装饰模式中装饰角色，只是维护了通道的正在关闭和关闭完成两个状态。然后到了MultiMessageHandler的received

（三）MultiMessageHandler的received

可以参考《dubbo源码解析（九）远程通信——Transport层》的（八）MultiMessageHandler

```
public void received(Channel channel, Object message) throws RemotingException {
    // 如果消息是MultiMessage类型的，也就是多消息类型
    if (message instanceof MultiMessage) {
        // 强制转化为MultiMessage
        MultiMessage list = (MultiMessage) message;
        // 把各个消息进行发送
        for (Object obj : list) {
            handler.received(channel, obj);
        }
    } else {
        // 直接发送
        handler.received(channel, message);
    }
}
```

该方法也比较简单，就是对于多消息的处理。

（四）HeartbeatHandler的received

可以参考《dubbo源码解析（十）远程通信——Exchange层》的（二十）HeartbeatHandler。其中就是对心跳事件做了处理。如果不是心跳请求，那么接下去走到AllChannelHandler的received。

（五）AllChannelHandler的received

可以参考《dubbo源码解析（九）远程通信——Transport层》的（十一）AllChannelHandler。该类处理的是连接、断开连接、捕获异常以及接收到的所有消息都分发到线程池。所以这里的received方法就是把请求分发到线程池，让线程池去执行该请求。

还记得我在之前文章里面讲到到Dispatcher接口吗，它是一个线程派发器。分别有五个实现：

Dispatcher实现类	对应的handler	用途
AllDispatcher	AllChannelHandler	所有消息都派发到线程池，包括请求，响应，连接事件，断开事件等
ConnectionOrderedDispatcher	ConnectionOrderedChannelHandler	在 IO 线程上，将连接和断开事件放入队列，有序逐个执行，其它消息派发到线程池
DirectDispatcher	无	所有消息都不派发到线程池，全部在 IO 线程上直接执行
ExecutionDispatcher	ExecutionChannelHandler	只有请求消息派发到线程池，不含响应。其它消息均在 IO 线程上执行
MessageOnlyDispatcher	MessageOnlyChannelHandler	只有请求和响应消息派发到线程池，其它消息均在 IO 线程上执行

这些Dispatcher的实现类以及对应的Handler都可以在《dubbo源码解析（九）远程通信——Transport层》中查看相关实现。dubbo默认all为派发策略。所以我在这里讲了AllChannelHandler的received。把消息送到线程池后，可以看到首先会创建一个ChannelEventRunnable实体。那么接下来就是线程接收并且执行任务了。

（六）ChannelEventRunnable的run

ChannelEventRunnable实现了Runnable接口，主要是用来接收消息事件，并且根据事件的种类来分别执行不同的操作。来看看它的run方法：

```
public void run() {
    // 如果是接收的消息
    if (state == ChannelState.RECEIVED) {
        try {
            // 直接调用下一个received
            handler.received(channel, message);
        } catch (Exception e) {
            logger.warn("ChannelEventRunnable handle " + state + " operation error, channel is " + channel
                    + ", message is " + message, e);
        }
    } else {
        switch (state) {
            //如果是连接事件请求
        case CONNECTED:
            try {
                // 执行连接
                handler.connected(channel);
            } catch (Exception e) {
                logger.warn("ChannelEventRunnable handle " + state + " operation error, channel is " + channel,
e);
            }
            break;
            // 如果是断开连接事件请求
        case DISCONNECTED:
            try {
                // 执行断开连接
```

可以看到把消息分为了几种类别，因为请求和响应消息出现频率明显比其他类型消息高，也就是RECEIVED，所以单独先做处理，根据不同的类型的消息，会被执行不同的逻辑，我们这里主要看state为RECEIVED的，那么如果是RECEIVED，则会执行下一个received方法。

（七）DecodeHandler的received

可以参考《dubbo源码解析（九）远程通信——Transport层》的（七）DecodeHandler。可以看到received方法中根据消息的类型进行不同的解码。而DecodeHandler 存在的意义就是保证请求或响应对象可在线程池中被解码，解码完成后，就会分发到HeaderExchangeHandler的received。

（八）HeaderExchangeHandler的received

```
public void received(Channel channel, Object message) throws RemotingException {
    // 设置接收到消息的时间戳
    channel.setAttribute(KEY_READ_TIMESTAMP, System.currentTimeMillis());
    // 获得通道
    final ExchangeChannel exchangeChannel = HeaderExchangeChannel.getOrAddChannel(channel);
    try {
        // 如果消息是Request类型
        if (message instanceof Request) {
            // handle request.
            // 强制转化为Request
            Request request = (Request) message;
            // 如果该请求是事件心跳事件或者只读事件
            if (request.isEvent()) {
                // 执行事件
                handlerEvent(channel, request);
            } else {
                // 如果是正常的调用请求，且需要响应
                if (request.isTwoWay()) {
                    // 处理请求
                    handleRequest(exchangeChannel, request);
                } else {
                    // 如果不需要响应，则继续下一步
                    handler.received(exchangeChannel, request.getData());
                }
            }
        } else if (message instanceof Response) {
```

该方法中就对消息进行了细分，事件请求、正常的调用请求、响应、telnet命令请求等，并且针对不同的消息类型做了不同的逻辑调用。我们这里主要看正常的调用请求。见下一步。

（九）HeaderExchangeHandler的handleRequest

```
void handleRequest(final ExchangeChannel channel, Request req) throws RemotingException {
    // 创建一个Response实例
    Response res = new Response(req.getId(), req.getVersion());
    // 如果请求被破坏了
    if (req.isBroken()) {
        // 获得请求的数据包
        Object data = req.getData();

        String msg;
        // 如果数据为空
        if (data == null) {
            //消息设置为空
            msg = null;
            // 如果在这之前已经出现异常，也就是数据为Throwable类型
        } else if (data instanceof Throwable) {
            // 响应消息把异常信息返回
            msg = StringUtils.toString((Throwable) data);
        } else {
            // 返回请求数据
            msg = data.toString();
        }
        res.setErrorMessage("Fail to decode request due to: " + msg);
        // 设置错误请求的状态码
        res.setStatus(Response.BAD_REQUEST);

        // 发送该消息
```

该方法是处理正常的调用请求，主要做了正常、异常调用的情况处理，并且加入了状态码，然后发送执行后的结果给客户端。接下来看下一步。

（十）DubboProtocol的requestHandler实例的reply

这里我默认是使用dubbo协议，所以执行的是DubboProtocol的requestHandler的reply方法。可以参考《dubbo源码解析（二十四）远程调用——dubbo协议》的（三）DubboProtocol

```
public CompletableFuture<Object> reply(ExchangeChannel channel, Object message) throws RemotingException {

    // 如果请求消息不属于会话域，则抛出异常
    if (!(message instanceof Invocation)) {
        throw new RemotingException(channel, "Unsupported request: "
                + (message == null ? null : (message.getClass().getName() + ": " + message))
                + ", channel: consumer: " + channel.getRemoteAddress() + " --> provider: " +
channel.getLocalAddress());
    }

    //强制类型转化
    Invocation inv = (Invocation) message;
    // 获得暴露的服务invoker
    Invoker<?> invoker = getInvoker(channel, inv);
    // need to consider backward-compatibility if it's a callback
    // 如果是回调服务
    if (Boolean.TRUE.toString().equals(inv.getAttachments().get(IS_CALLBACK_SERVICE_INVOKE))) {
        // 获得 方法定义
        String methodsStr = invoker.getUrl().getParameters().get("methods");
        boolean hasMethod = false;
        // 如果只有一个方法定义
        if (methodsStr == null || !methodsStr.contains(",")) {
            // 设置会话域中是否有一致的方法定义标志
            hasMethod = inv.getMethodName().equals(methodsStr);
        } else {
            // 分割不同的方法
```

上述代码有些变化，是最新的代码，加入了CompletableFuture，这个我会在后续的异步化改造中讲到。这里主要关注的是又开始跟客户端发请求一样执行invoke调用链了。

（十一）ProtocolFilterWrapper的CallbackRegistrationInvoker的invoke

可以直接参考《dubbo源码解析（四十六）消费端发送请求过程》的（六）ProtocolFilterWrapper的内部类CallbackRegistrationInvoker的invoke。

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

可以直接参考《dubbo源码解析（四十六）消费端发送请求过程》的（七）ProtocolFilterWrapper的buildInvokerChain方法中的invoker实例的invoke方法。

（十三）EchoFilter的invoke

可以参考《dubbo源码解析（二十）远程调用——Filter》的（八）EchoFilter

```
public Result invoke(Invoker<?> invoker, Invocation inv) throws RpcException {
    // 如果调用的方法是回声测试的方法 则直接返回结果，否则 调用下一个调用链
    if (inv.getMethodName().equals($ECHO) && inv.getArguments() != null && inv.getArguments().length == 1) {
        // 创建一个默认的AsyncRpcResult返回
        return AsyncRpcResult.newDefaultAsyncResult(inv.getArguments()[0], inv);
    }
    return invoker.invoke(inv);
}
```

该过滤器就是对回声测试的调用进行拦截，上述源码跟连接中源码唯一区别就是改为了AsyncRpcResult。

（十四）ClassLoaderFilter的invoke

可以参考《dubbo源码解析（二十）远程调用——Filter》的（三）ClassLoaderFilter，用来做类加载器的切换。

（十五）GenericFilter的invoke

可以参考《dubbo源码解析（二十）远程调用——Filter》的（十一）GenericFilter。

```
public Result invoke(Invoker<?> invoker, Invocation inv) throws RpcException {
    // 如果是泛化调用
    if ((inv.getMethodName().equals($INVOKE) || inv.getMethodName().equals($INVOKE_ASYNC))
            && inv.getArguments() != null
            && inv.getArguments().length == 3
            && !GenericService.class.isAssignableFrom(invoker.getInterface())) {
        // 获得请求名字
        String name = ((String) inv.getArguments()[0]).trim();
        // 获得请求参数类型
        String[] types = (String[]) inv.getArguments()[1];
        // 获得请求参数
        Object[] args = (Object[]) inv.getArguments()[2];
        try {
            // 获得方法
            Method method = ReflectUtils.findMethodByMethodSignature(invoker.getInterface(), name, types);
            // 获得该方法的参数类型
            Class<?>[] params = method.getParameterTypes();
            if (args == null) {
                args = new Object[params.length];
            }
            // 获得附加值
            String generic = inv.getAttachment(GENERIC_KEY);

            if (StringUtils.isBlank(generic)) {
                generic = RpcContext.getContext().getAttachment(GENERIC_KEY);
```

跟连接内的代码对比，第一个就是对过滤器设计发生了变化，这个我在异步化改造里面会讲到，第二个是新增了protobuf-json的泛化序列化方式。

（十六）ContextFilter的invoke

可以参考《dubbo源码解析（二十）远程调用——Filter》的（六）ContextFilter，最新代码几乎差不多，除了因为对Filter的设计做了修改以外，还有新增了tag路由的相关逻辑，tag相关部分我会在后续文章中讲解，该类主要是做了初始化rpc上下文。

（十七）TraceFilter的invoke

可以参考《dubbo源码解析（二十四）远程调用——dubbo协议》的（十三）TraceFilter，该过滤器是增强的功能是通道的跟踪，会在通道内把最大的调用次数和现在的调用数量放进去。方便使用telnet来跟踪服务的调用次数等。

（十八）TimeoutFilter的invoke

可以参考《dubbo源码解析（二十）远程调用——Filter》的（十三）TimeoutFilter，该过滤器是当服务调用超时的时候，记录告警日志。

（十九）MonitorFilter的invoke

```
public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
    // 如果开启监控
    if (invoker.getUrl().hasParameter(MONITOR_KEY)) {
        // 设置开始监控时间
        invocation.setAttachment(MONITOR_FILTER_START_TIME, String.valueOf(System.currentTimeMillis()));
        // 对同时在线数量加1
        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)) {
            // 收集监控对数据，并且更新监控数据
            collect(invoker, invocation, result, RpcContext.getContext().getRemoteHost(),
Long.valueOf(invocation.getAttachment(MONITOR_FILTER_START_TIME)), false);
            // 同时在线监控数减1
            getConcurrent(invoker, invocation).decrementAndGet(); // count down
        }
    }

    @Override
    public void onError(Throwable t, Invoker<?> invoker, Invocation invocation) {
        if (invoker.getUrl().hasParameter(MONITOR_KEY)) {
            // 收集监控对数据，并且更新监控数据
            collect(invoker, invocation, null, RpcContext.getContext().getRemoteHost(),
Long.valueOf(invocation.getAttachment(MONITOR_FILTER_START_TIME)), true);
            // 同时在线监控数减1
            getConcurrent(invoker, invocation).decrementAndGet(); // count down
        }
    }

    private void collect(Invoker<?> invoker, Invocation invocation, Result result, String remoteHost, long start,
```

在invoke里面只是做了记录开始监控的时间以及对同时监控的数量加1操作，当结果回调时，会对结果数据做搜集计算，最后通过监控服务记录和发送最新信息。

（二十）ExceptionFilter的invoke

可以参考《dubbo源码解析（二十）远程调用——Filter》的（九）ExceptionFilter，该过滤器主要是对异常的处理。

（二十一）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。

（二十二）DelegateProviderMetaDataInvoker的invoke

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

该类也是用了装饰模式，不过该类是invoker和配置中心的适配类，其中也没有进行实际的功能增强。

（二十三）AbstractProxyInvoker的invoke

可以参考《dubbo源码解析（二十三）远程调用——Proxy》的（二）AbstractProxyInvoker。不过代码已经有更新了，下面贴出最新的代码。

```
public Result invoke(Invocation invocation) throws RpcException {
    try {
        // 执行下一步
        Object value = doInvoke(proxy, invocation.getMethodName(), invocation.getParameterTypes(),
invocation.getArguments());
        // 把返回结果用CompletableFuture包裹
        CompletableFuture<Object> future = wrapWithFuture(value, invocation);
        // 创建AsyncRpcResult实例
        AsyncRpcResult asyncRpcResult = new AsyncRpcResult(invocation);
        future.whenComplete((obj, t) -> {
            AppResponse result = new AppResponse();
            // 如果抛出异常
            if (t != null) {
                // 属于CompletionException异常
                if (t instanceof CompletionException) {
                    // 设置异常信息
                    result.setException(t.getCause());
                } else {
                    // 直接设置异常
                    result.setException(t);
                }
            } else {
                // 如果没有异常，则把结果放入异步结果内
                result.setValue(obj);
            }
            // 完成
```

这里主要是因为异步化改造而出现的代码变化，我会在异步化改造中讲到这部分。现在主要来看下一步。

（二十四）JavassistProxyFactory的getInvoker方法中匿名类的doInvoke

这里默认代理实现方式是Javassist。可以参考《dubbo源码解析（二十三）远程调用——Proxy》的（六）JavassistProxyFactory。其中Wrapper 是一个抽象类，其中 invokeMethod 是一个抽象方法。dubbo 会在运行时通过 Javassist 框架为 Wrapper 生成实现类，并实现 invokeMethod 方法，该方法最终会根据调用信息调用具体的服务。以 DemoServiceImpl 为例，Javassist 为其生成的代理类如下。

```
/** Wrapper0 是在运行时生成的，大家可使用 Arthas 进行反编译 */
public class Wrapper0 extends Wrapper implements ClassGenerator.DC {
    public static String[] pns;
    public static Map pts;
    public static String[] mns;
    public static String[] dmns;
    public static Class[] mts0;

    // 省略其他方法

    public Object invokeMethod(Object object, String string, Class[] arrclass, Object[] arrobject) throws
InvocationTargetException {
        DemoService demoService;
        try {
            // 类型转换
            demoService = (DemoService)object;
        }
        catch (Throwable throwable) {
            throw new IllegalArgumentException(throwable);
        }
        try {
            // 根据方法名调用指定的方法
            if ("sayHello".equals(string) && arrclass.length == 1) {
                return demoService.sayHello((String)arrobject[0]);
            }
        }
```

然后就是直接调用的是对应的方法了。到此，方法执行完成了。

结果返回

可以看到我上述讲到的（八）HeaderExchangeHandler的received和（九）HeaderExchangeHandler的handleRequest有好几处channel.send方法的调用，也就是当结果返回的返回的时候，会主动发送执行结果给客户端。当然发送的时候还是会对结果Response对象进行编码，编码逻辑我就先不在这里阐述。

当客户端接收到这个返回的消息时候，进行解码后，识别为Response 对象，将该对象派发到线程池中，该过程跟服务端接收到调用请求到逻辑是一样的，可以参考上述的解析，区别在于到（八）HeaderExchangeHandler的received方法的时候，执行的是handleResponse方法。

（九）HeaderExchangeHandler的handleResponse

```
static void handleResponse(Channel channel, Response response) throws RemotingException {
    // 如果响应不为空，并且不是心跳事件的响应，则调用received
    if (response != null && !response.isHeartbeat()) {
        DefaultFuture.received(channel, response);
    }
}
```

（十）DefaultFuture的received

可以参考《dubbo源码解析（十）远程通信——Exchange层》的（七）DefaultFuture，不过该类的继承的是CompletableFuture，因为对异步化的改造，该类已经做了一些变化。

```
public static void received(Channel channel, Response response) {
    received(channel, response, false);
}

public static void received(Channel channel, Response response, boolean timeout) {
    try {
        // future集合中移除该请求的future，（响应id和请求id一一对应的）
        DefaultFuture future = FUTURES.remove(response.getId());
        if (future != null) {
            //获得超时
            Timeout t = future.timeoutCheckTask;
            // 如果没有超时，则取消timeoutCheckTask
            if (!timeout) {
                // decrease Time
                t.cancel();
            }
            // 接收响应结果
            future.doReceived(response);
        } else {
            logger.warn("The timeout response finally returned at "
                    + (new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS").format(new Date()))
                    + ", response " + response
                    + (channel == null ? "" : ", channel: " + channel.getLocalAddress()
                    + " -> " + channel.getRemoteAddress()));
        }
    } finally {
```

该方法中主要是对超时的处理，还有对应的请求和响应的匹配，也就是返回相应id的future。

（十一）DefaultFuture的doReceived

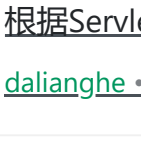
可以参考《dubbo源码解析（十）远程通信——Exchange层》的（七）DefaultFuture，不过因为运用了CompletableFuture，所以该方法完全重写了。这部分的改造我也会在异步化改造中讲述到。

```
private void doReceived(Response res) {
    // 如果结果为空，则抛出异常
    if (res == null) {
        throw new IllegalStateException("response cannot be null");
    }
    // 如果结果的状态码为ok
    if (res.getStatus() == Response.OK) {
        // 则future调用完成
        this.complete(res.getResult());
    } else if (res.getStatus() == Response.CLIENT_TIMEOUT || res.getStatus() == Response.SERVER_TIMEOUT) {
        // 如果超时，则返回一个超时异常
        this.completeExceptionally(new TimeoutException(res.getStatus() == Response.SERVER_TIMEOUT, channel,
res.getErrorMessage()));
    } else {
        // 否则返回一个RemotingException
        this.completeExceptionally(new RemotingException(channel, res.getErrorMessage()));
    }
}
```

随后用户线程即可从 DefaultFuture 实例中获取到相应结果。

后记

该文章讲解了dubbo调服务端接收到请求后一系列处理的所有步骤以及服务端怎么把结果发送给客户端，是目前最新代码的解析。因为里面涉及到很多流程，所以可以自己debug一步一步看一看整个过程。可以看到本文涉及到异步化改造已经很多了，这也是我在讲解异步化改造前想先讲解这些过程的原因。只有弄清楚这些过程，才能更加理解对于异步化改造的意义。下一篇文将讲解异步化改造。



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

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

SpringMVC之源码分析--请求过程

根据Servlet规范，当用户请求到达应用时，由Servlet的service()方法进行处理，对于SpringMVC而言，处理用户请求的入口为Dispa...

dalianghe • 阅读 16

Node.js源码解析-HTTP请求响应过程