crazyhzm 265

Dubbo源码解析（十三）远程通信——Grizzly

java grizzly dubbo 阅读约 26 分钟

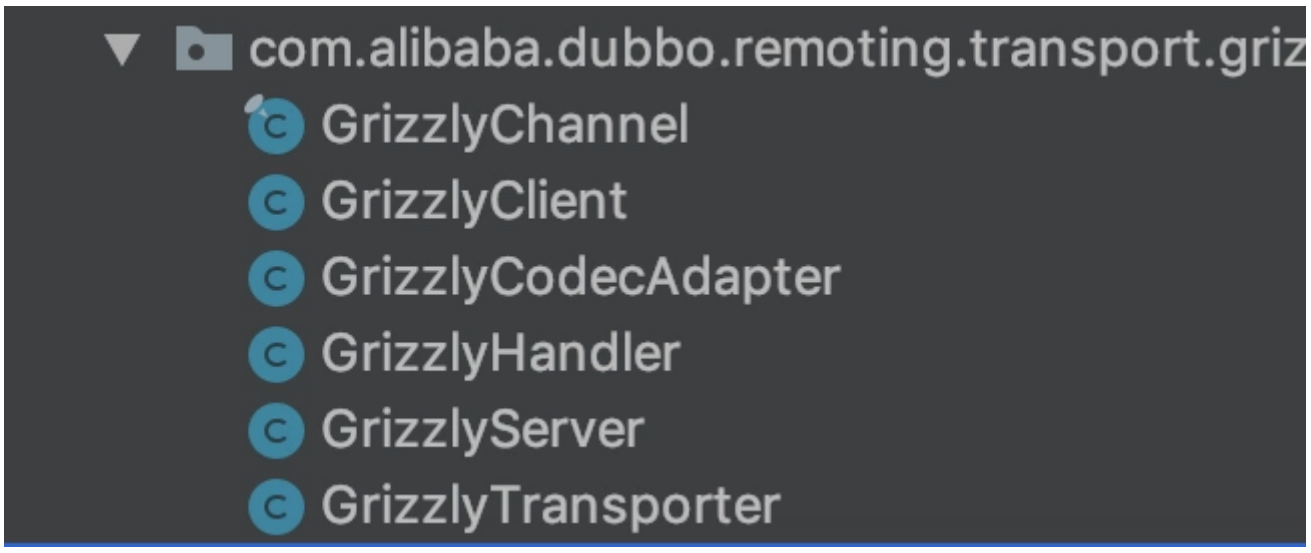
远程通讯——Grizzly

> 目标：介绍基于Grizzly的来实现的远程通信、介绍dubbo-remoting-grizzly内的源码解析。

前言

Grizzly NIO框架的设计初衷是帮助开发者更好地利用Java NIO API，构建强大的可扩展的服务器应用。关于Grizzly我也没有很熟悉，所以只能根据grizzly在dubbo的远程通讯中应用稍微讲解一下。

下面是dubbo-remoting-grizzly下的包结构：



源码分析

（一）GrizzlyChannel

1.属性

```
private static final Logger logger = LoggerFactory.getLogger(GrizzlyChannel.class);

/**
 * 通道key
 */
private static final String CHANNEL_KEY = GrizzlyChannel.class.getName() + ".CHANNEL";

/**
 * 通道属性
 */
private static final Attribute<GrizzlyChannel> ATTRIBUTE =
Grizzly.DEFAULT_ATTRIBUTE_BUILDER.createAttribute(CHANNEL_KEY);

/**
 * Grizzly的连接实例
 */
private final Connection<?> connection;
```

可以看到，该类中的ATTRIBUTE和connection都是Grizzly涉及到的属性，ATTRIBUTE中封装了GrizzlyChannel的实例还有Connection实例。Grizzly把连接的一些连接的方法定义在了Connection接口中，包括获得远程地址、检测通道是否连接等方法。

2.send

```
@Override
@SuppressWarnings("rawtypes")
public void send(Object message, boolean sent) throws RemotingException {
    super.send(message, sent);

    int timeout = 0;
    try {
        // 发送消息，获得GrizzlyFuture实例
        GrizzlyFuture future = connection.write(message);
        if (sent) {
            // 获得延迟多少时间获得响应
            timeout = getUrl().getPositiveParameter(Constants.TIMEOUT_KEY, Constants.DEFAULT_TIMEOUT);
            // 获得请求的值
            future.get(timeout, TimeUnit.MILLISECONDS);
        }
    } catch (TimeoutException e) {
        throw new RemotingException(this, "Failed to send message " + message + " to " + getRemoteAddress()
                + "in timeout(" + timeout + "ms) limit", e);
    } catch (Throwable e) {
        throw new RemotingException(this, "Failed to send message " + message + " to " + getRemoteAddress() + ",
cause: " + e.getMessage(), e);
    }
}
```

该方法是发送消息的方法，调用了connection的write方法，它会返回一个GrizzlyFuture实例，GrizzlyFuture继承了java.util.concurrent.Future。

其他方法比较简单，实现的方法都基本调用connection的方法或者Grizzly.DEFAULT_ATTRIBUTE_BUILDER的方法。

（二）GrizzlyHandler

该类是Grizzly的通道处理器，它继承了BaseFilter。

```
/**
 * url
 */
private final URL url;

/**
 * 通道处理器
 */
private final ChannelHandler handler;
```

该类有两个属性，这两个属性应该很熟悉了。而该类有handleConnect、handleClose、handleRead、handleWrite、exceptionOccurred的实现方法，分别调用了ChannelHandler中封装的五个方法。举个例子：

```
@Override
public NextAction handleConnect(FilterChainContext ctx) throws IOException {
    // 获得Connection连接实例
    Connection<?> connection = ctx.getConnection();
    // 获得GrizzlyChannel通道
    GrizzlyChannel channel = GrizzlyChannel.getOrAddChannel(connection, url, handler);
    try {
        // 连接
        handler.connected(channel);
    } catch (RemotingException e) {
        throw new IOException(StringUtils.toString(e));
    } finally {
        GrizzlyChannel.removeChannelIfDisconnected(connection);
    }
    return ctx.getInvokeAction();
}
```

可以看到获得GrizzlyChannel通道就调用了handler.connected进行连接。而其他四个方法也差不多，感兴趣的可以自行查看代码。

（三）GrizzlyClient

该类是Grizzly的客户端实现类，继承了AbstractClient类

```
/**
 * Grizzly中的传输对象
 */
private TCPNIOTransport transport;

/**
 * 连接实例
 */
private volatile Connection<?> connection; // volatile, please copy reference to use
```

可以看到属性中有TCPNIOTransport的实例，在该类中实现的客户端方法都调用了transport中的方法，TCPNIOTransport中封装了创建，连接，断开连接等方法。

我们来看创建客户端的逻辑：

```
@Override
protected void doOpen() throws Throwable {
    // 做一些过滤，用于处理消息
    FilterChainBuilder filterChainBuilder = FilterChainBuilder.stateless();
    filterChainBuilder.add(new TransportFilter());
    filterChainBuilder.add(new GrizzlyCodecAdapter(getCodec(), getUrl(), this));
    filterChainBuilder.add(new GrizzlyHandler(getUrl(), this));
    // 传输构建者
    TCPNIOTransportBuilder builder = TCPNIOTransportBuilder.newInstance();
    // 获得线程池配置实例
    ThreadPoolConfig config = builder.getWorkerThreadPoolConfig();
    // 设置线程池的配置，包括核心线程数等
    config.setPoolName(CLIENT_THREAD_POOL_NAME)
            .setQueueLimit(-1)
            .setCorePoolSize(0)
            .setMaxPoolSize(Integer.MAX_VALUE)
            .setKeepAliveTime(60L, TimeUnit.SECONDS);
    // 设置创建属性
    builder.setTcpNoDelay(true).setKeepAlive(true)
            .setConnectionTimeout(getConnectTimeout())
            .setIOStrategy(SameThreadIOStrategy.getInstance());
    // 创建一个transport
    transport = builder.build();
    transport.setProcessor(filterChainBuilder.build());
    // 创建客户端
    transport.start();
```

可以看到首先是设置及一些过滤，在上面对信息先处理，然后设置了线程池的配置，创建transport，并且用transport来进行创建客户端。

（四）GrizzlyServer

该类是Grizzly的服务器实现类，继承了AbstractServer类。

```
/**
 * 连接该服务器的客户端通道集合
 */
private final Map<String, Channel> channels = new ConcurrentHashMap<String, Channel>(); // <ip:port, channel>

/**
 * 传输实例
 */
private TCPNIOTransport transport;
```

该类中有两个属性，其中transport用法跟GrizzlyClient中的一样。

我也就讲解一个doOpen方法：

```
@Override
protected void doOpen() throws Throwable {
    // 增加过滤器，来处理信息
    FilterChainBuilder filterChainBuilder = FilterChainBuilder.stateless();
    filterChainBuilder.add(new TransportFilter());

    filterChainBuilder.add(new GrizzlyCodecAdapter(getCodec(), getUrl(), this));
    filterChainBuilder.add(new GrizzlyHandler(getUrl(), this));
    TCPNIOTransportBuilder builder = TCPNIOTransportBuilder.newInstance();
    // 获得线程池配置
    ThreadPoolConfig config = builder.getWorkerThreadPoolConfig();
    config.setPoolName(SERVER_THREAD_POOL_NAME).setQueueLimit(-1);
    // 获得url配置中线程池类型
    String threadpool = getUrl().getParameter(Constants.THREADPOOL_KEY, Constants.DEFAULT_THREADPOOL);
    if (Constants.DEFAULT_THREADPOOL.equals(threadpool)) {
        // 优先从url获得线程池的线程数，默认线程数为200
        int threads = getUrl().getPositiveParameter(Constants.THREADS_KEY, Constants.DEFAULT_THREADS);
        // 设置线程池配置
        config.setCorePoolSize(threads).setMaxPoolSize(threads)
                .setKeepAliveTime(0L, TimeUnit.SECONDS);
        // 如果是cached类型的线程池
    } else if ("cached".equals(threadpool)) {
        int threads = getUrl().getPositiveParameter(Constants.THREADS_KEY, Integer.MAX_VALUE);
        // 设置核心线程数为0、最大线程数为threads
        config.setCorePoolSize(0).setMaxPoolSize(threads)
                .setKeepAliveTime(60L, TimeUnit.SECONDS);
```

该方法是创建服务器，可以看到操作跟GrizzlyClient中的差不多。

（五）GrizzlyTransporter

该类实现了Transporter接口，是基于Grizzly的传输层实现。

```
public class GrizzlyTransporter implements Transporter {

    public static final String NAME = "grizzly";

    @Override
    public Server bind(URL url, ChannelHandler listener) throws RemotingException {
        // 返回GrizzlyServer实例
        return new GrizzlyServer(url, listener);
    }

    @Override
    public Client connect(URL url, ChannelHandler listener) throws RemotingException {
        // // 返回GrizzlyClient实例
        return new GrizzlyClient(url, listener);
    }

}
```

可以看到，bind和connect方法分别就是创建了GrizzlyServer和GrizzlyClient实例。这里我建议查看一下《dubbo源码解析（九）远程通信——Transport层》。

（六）GrizzlyCodecAdapter

该类是Grizzly编解码类，继承了BaseFilter。

1.属性和构造方法

```
/**
 * 编解码器
 */
private final Codec2 codec;

/**
 * url
 */
private final URL url;

/**
 * 通道处理器
 */
private final ChannelHandler handler;

/**
 * 缓存大小
 */
private final int bufferSize;

/**
 * 空缓存区
 */
private ChannelBuffer previousData = ChannelBuffers.EMPTY_BUFFER;

public GrizzlyCodecAdapter(Codec2 codec, URL url, ChannelHandler handler) {
```

2.handleWrite

```
@Override
public NextAction handleWrite(FilterChainContext context) throws IOException {
    Connection<?> connection = context.getConnection();
    GrizzlyChannel channel = GrizzlyChannel.getOrAddChannel(connection, url, handler);
    try {
        // 分配一个1024的动态缓冲区
        ChannelBuffer channelBuffer = ChannelBuffers.dynamicBuffer(1024); // Do not need to close

        // 获得消息
        Object msg = context.getMessage();
        // 编码
        codec.encode(channel, channelBuffer, msg);

        // 检测是否连接
        GrizzlyChannel.removeChannelIfDisconnected(connection);
        // 分配缓冲区
        Buffer buffer = connection.getTransport().getMemoryManager().allocate(channelBuffer.readableBytes());
        // 把channelBuffer的数据写到buffer
        buffer.put(channelBuffer.toByteBuffer());
        buffer.flip();
        buffer.allowBufferDispose(true);
        // 设置到上下文
        context.setMessage(buffer);
    } finally {
        GrizzlyChannel.removeChannelIfDisconnected(connection);
    }
```

该方法是写数据，可以发现编码调用的是 codec.encode，其他的我都在注释里写明了，关键还是对前面两篇文章的一些内容需要理解。

3.handleRead

```
@Override
public NextAction handleRead(FilterChainContext context) throws IOException {
    Object message = context.getMessage();
    Connection<?> connection = context.getConnection();
    Channel channel = GrizzlyChannel.getOrAddChannel(connection, url, handler);
    try {
        // 如果接收的是一个数据包
        if (message instanceof Buffer) { // receive a new packet
            Buffer grizzlyBuffer = (Buffer) message; // buffer

            ChannelBuffer frame;

            // 如果缓冲区可读
            if (previousData.readable()) {
                // 如果该缓冲区是动态的缓冲区
                if (previousData instanceof DynamicChannelBuffer) {
                    // 写入数据
                    previousData.writeBytes(grizzlyBuffer.toByteBuffer());
                    frame = previousData;
                } else {
                    // 获得需要的缓冲区大小
                    int size = previousData.readableBytes() + grizzlyBuffer.remaining();
                    // 新建一个动态缓冲区
                    frame = ChannelBuffers.dynamicBuffer(size > bufferSize ? size : bufferSize);
                    // 写入previousData中的数据
                    frame.writeBytes(previousData, previousData.readableBytes());
```

该方法是读数据，直接调用了codec.decode进行解码。

后记

> 该部分相关的源码解析地址：https://github.com/CrazyHZM/i...

该文章讲解了基于grizzly的来实现的远程通信、介绍dubbo-remoting-grizzly内的源码解析，关键需要对grizzly有所了解。下一篇我会讲解基于http实现远程通信部分。