Dubbo改动

Dubbo存在两处改动

1. dubbo-rpc-api module中
   com.alibaba.dubbo.rpc.proxy.InvokerInvocationHandler
   改动后内容如下

```
public class InvokerInvocationHandler implements InvocationHandler {

    private final Invoker<?> invoker;

    public InvokerInvocationHandler(Invoker<?> handler) {
        this.invoker = handler;
    }

    public Object invoke(Object proxy, Method method, Object[] args) thro
ws Throwable {
        String methodName = method.getName();
        Class<?>[] parameterTypes = method.getParameterTypes();
        if (method.getDeclaringClass() == Object.class) {
            return method.invoke(invoker, args);
        }
        if ("toString".equals(methodName) && parameterTypes.length == 0)
 {
            return invoker.toString();
        }
        if ("hashCode".equals(methodName) && parameterTypes.length == 0)
 {
            return invoker.hashCode();
        }
        if ("equals".equals(methodName) && parameterTypes.length == 1) {
            return invoker.equals(args[0]);
        }
        //改动了这里，添加了方法，用于处理invocation
        RpcInvocation invocation = new RpcInvocation(method, args);
        doBeforeInvokeReturn(invocation);
        return invoker.invoke(invocation).recreate();
    }
    /**
    * 这个是添加的方法
    * @param invocation 传递进来的invocation对象
    * @return 修改后的对象
    * @throws Throwable
    */
    public Object doBeforeInvokeReturn(RpcInvocation invocation) throws T
hrowable {
        return invocation;
    }

}
```

2. dubbo-rpc-default module 下
   com.alibaba.dubbo.rpc.protocol.dubbo.DubboProtocol

修改后代码如下

```
public class DubboProtocol extends AbstractProtocol {

    public static final String NAME = "dubbo";

    public static final int DEFAULT_PORT = 20880;
    private static final String IS_CALLBACK_SERVICE_INVOKE = "_isCallBack
ServiceInvoke";
    private static DubboProtocol INSTANCE;
    private final Map<String, ExchangeServer> serverMap = new ConcurrentH
ashMap<String, ExchangeServer>(); // <host:port,Exchanger>
    private final Map<String, ReferenceCountExchangeClient> referenceClie
ntMap = new ConcurrentHashMap<String, ReferenceCountExchangeClient>(); //
 <host:port,Exchanger>
    private final ConcurrentMap<String, LazyConnectExchangeClient> ghostC
lientMap = new ConcurrentHashMap<String, LazyConnectExchangeClient>();
    //consumer side export a stub service for dispatching event
    //servicekey-stubmethods
    private final ConcurrentMap<String, String> stubServiceMethodsMap = n
ew ConcurrentHashMap<String, String>();
    private ExchangeHandler requestHandler = new DubboProtocolExchangeHan
dlerAdapter();

    public DubboProtocol() {
        INSTANCE = this;
    }

    public static DubboProtocol getDubboProtocol() {
        if (INSTANCE == null) {
            ExtensionLoader.getExtensionLoader(Protocol.class).getExtensi
on(DubboProtocol.NAME); // load
        }
        return INSTANCE;
    }

    public Collection<ExchangeServer> getServers() {
        return Collections.unmodifiableCollection(serverMap.values());
    }

    public Collection<Exporter<?>> getExporters() {
        return Collections.unmodifiableCollection(exporterMap.values());
    }

    Map<String, Exporter<?>> getExporterMap() {
        return exporterMap;
    }

    private boolean isClientSide(Channel channel) {
        InetSocketAddress address = channel.getRemoteAddress();
        URL url = channel.getUrl();
        return url.getPort() == address.getPort() &&
                NetUtils.filterLocalHost(channel.getUrl().getIp())
```

```
                        .equals(NetUtils.filterLocalHost(address.getAddre
ss().getHostAddress()));
    }

    Invoker<?> getInvoker(Channel channel, Invocation inv) throws Remotin
gException {
        boolean isCallBackServiceInvoke = false;
        boolean isStubServiceInvoke = false;
        int port = channel.getLocalAddress().getPort();
        String path = inv.getAttachments().get(Constants.PATH_KEY);
        //如果是客户端的回调服务.
        isStubServiceInvoke = Boolean.TRUE.toString().equals(inv.getAttac
hments().get(Constants.STUB_EVENT_KEY));
        if (isStubServiceInvoke) {
            port = channel.getRemoteAddress().getPort();
        }
        //callback
        isCallBackServiceInvoke = isClientSide(channel) && !isStubService
Invoke;
        if (isCallBackServiceInvoke) {
            path = inv.getAttachments().get(Constants.PATH_KEY) + "." + i
nv.getAttachments().get(Constants.CALLBACK_SERVICE_KEY);
            inv.getAttachments().put(IS_CALLBACK_SERVICE_INVOKE, Boolean.
TRUE.toString());
        }
        String serviceKey = serviceKey(port, path, inv.getAttachments().g
et(Constants.VERSION_KEY), inv.getAttachments().get(Constants.GROUP_KE
Y));

        DubboExporter<?> exporter = (DubboExporter<?>) exporterMap.get(se
rviceKey);

        if (exporter == null)
            throw new RemotingException(channel, "Not found exported serv
ice: " + serviceKey + " in " + exporterMap.keySet() + ", may be version o
r group mismatch " + ", channel: consumer: " + channel.getRemoteAddress()
 + " --> provider: " + channel.getLocalAddress() + ", message:" + inv);

        return exporter.getInvoker();
    }

    public Collection<Invoker<?>> getInvokers() {
        return Collections.unmodifiableCollection(invokers);
    }

    public int getDefaultPort() {
        return DEFAULT_PORT;
    }

    public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException
 {
        URL url = invoker.getUrl();
```

```
        // export service.
        String key = serviceKey(url);
        DubboExporter<T> exporter = new DubboExporter<T>(invoker, key, ex
porterMap);
        exporterMap.put(key, exporter);

        //export an stub service for dispaching event
        Boolean isStubSupportEvent = url.getParameter(Constants.STUB_EVEN
T_KEY, Constants.DEFAULT_STUB_EVENT);
        Boolean isCallbackservice = url.getParameter(Constants.IS_CALLBAC
K_SERVICE, false);
        if (isStubSupportEvent && !isCallbackservice) {
            String stubServiceMethods = url.getParameter(Constants.STUB_E
VENT_METHODS_KEY);
            if (stubServiceMethods == null || stubServiceMethods.length()
 == 0) {
                if (logger.isWarnEnabled()) {
                    logger.warn(new IllegalStateException("consumer [" +
 url.getParameter(Constants.INTERFACE_KEY) +
                            "], has set stubproxy support event ,but no s
tub methods founded."));
                }
            } else {
                stubServiceMethodsMap.put(url.getServiceKey(), stubServic
eMethods);
            }
        }

        openServer(url);

        return exporter;
    }

    private void openServer(URL url) {
        // find server.
        String key = url.getAddress();
        //client 也可以暴露一个只有server可以调用的服务。
        boolean isServer = url.getParameter(Constants.IS_SERVER_KEY,
true);
        if (isServer) {
            ExchangeServer server = serverMap.get(key);
            if (server == null) {
                serverMap.put(key, createServer(url));
            } else {
                //server支持reset,配合override功能使用
                server.reset(url);
            }
        }
    }

    private ExchangeServer createServer(URL url) {
```

```
        //默认开启server关闭时发送readonly事件
        url = url.addParameterIfAbsent(Constants.CHANNEL_READONLYEVENT_SE
NT_KEY, Boolean.TRUE.toString());
        //默认开启heartbeat
        url = url.addParameterIfAbsent(Constants.HEARTBEAT_KEY, String.va
lueOf(Constants.DEFAULT_HEARTBEAT));
        String str = url.getParameter(Constants.SERVER_KEY, Constants.DEF
AULT_REMOTING_SERVER);

        if (str != null && str.length() > 0 && !ExtensionLoader.getExtens
ionLoader(Transporter.class).hasExtension(str))
            throw new RpcException("Unsupported server type: " + str + ",
 url: " + url);

        url = url.addParameter(Constants.CODEC_KEY, DubboCodec.NAME);
        ExchangeServer server;
        try {
            server = Exchangers.bind(url, requestHandler);
        } catch (RemotingException e) {
            throw new RpcException("Fail to start server(url: " + url +
") " + e.getMessage(), e);
        }
        str = url.getParameter(Constants.CLIENT_KEY);
        if (str != null && str.length() > 0) {
            Set<String> supportedTypes = ExtensionLoader.getExtensionLoad
er(Transporter.class).getSupportedExtensions();
            if (!supportedTypes.contains(str)) {
                throw new RpcException("Unsupported client type: " + st
r);
            }
        }
        return server;
    }

    public <T> Invoker<T> refer(Class<T> serviceType, URL url) throws Rpc
Exception {
        // create rpc invoker.
        DubboInvoker<T> invoker = new DubboInvoker<T>(serviceType, url, g
etClients(url), invokers);
        invokers.add(invoker);
        return invoker;
    }

    private ExchangeClient[] getClients(URL url) {
        //是否共享连接
        boolean service_share_connect = false;
        int connections = url.getParameter(Constants.CONNECTIONS_KEY, 0);
        //如果connections不配置，则共享连接，否则每服务每连接
        if (connections == 0) {
            service_share_connect = true;
            connections = 1;
        }
```

```
        ExchangeClient[] clients = new ExchangeClient[connections];
        for (int i = 0; i < clients.length; i++) {
            if (service_share_connect) {
                clients[i] = getSharedClient(url);
            } else {
                clients[i] = initClient(url);
            }
        }
        return clients;
    }

    /**
     * 获取共享连接
     */
    private ExchangeClient getSharedClient(URL url) {
        String key = url.getAddress();
        ReferenceCountExchangeClient client = referenceClientMap.get(ke
y);
        if (client != null) {
            if (!client.isClosed()) {
                client.incrementAndGetCount();
                return client;
            } else {
                referenceClientMap.remove(key);
            }
        }
        synchronized (key.intern()) {
            ExchangeClient exchangeClient = initClient(url);
            client = new ReferenceCountExchangeClient(exchangeClient, gho
stClientMap);
            referenceClientMap.put(key, client);
            ghostClientMap.remove(key);
            return client;
        }
    }

    /**
     * 创建新连接.
     */
    private ExchangeClient initClient(URL url) {

        // client type setting.
        String str = url.getParameter(Constants.CLIENT_KEY, url.getParame
ter(Constants.SERVER_KEY, Constants.DEFAULT_REMOTING_CLIENT));

        String version = url.getParameter(Constants.DUBBO_VERSION_KEY);
        boolean compatible = (version != null && version.startsWith("1.
0."));
        url = url.addParameter(Constants.CODEC_KEY, DubboCodec.NAME);
        //默认开启heartbeat
```

```
        url = url.addParameterIfAbsent(Constants.HEARTBEAT_KEY, String.va
lueOf(Constants.DEFAULT_HEARTBEAT));

        // BIO存在严重性能问题，暂时不允许使用
        if (str != null && str.length() > 0 && !ExtensionLoader.getExtens
ionLoader(Transporter.class).hasExtension(str)) {
            throw new RpcException("Unsupported client type: " + str +
"," +
                    " supported client type is " + StringUtils.join(Exten
sionLoader.getExtensionLoader(Transporter.class).getSupportedExtensions
(), " "));
        }

        ExchangeClient client;
        try {
            //设置连接应该是lazy的
            if (url.getParameter(Constants.LAZY_CONNECT_KEY, false)) {
                client = new LazyConnectExchangeClient(url, requestHandle
r);
            } else {
                client = Exchangers.connect(url, requestHandler);
            }
        } catch (RemotingException e) {
            throw new RpcException("Fail to create remoting client for se
rvice(" + url + "): " + e.getMessage(), e);
        }
        return client;
    }

    public void destroy() {
        for (String key : new ArrayList<String>(serverMap.keySet())) {
            ExchangeServer server = serverMap.remove(key);
            if (server != null) {
                try {
                    if (logger.isInfoEnabled()) {
                        logger.info("Close dubbo server: " + server.getLo
calAddress());
                    }
                    server.close(getServerShutdownTimeout());
                } catch (Throwable t) {
                    logger.warn(t.getMessage(), t);
                }
            }
        }

        for (String key : new ArrayList<String>(referenceClientMap.keySet
())) {
            ExchangeClient client = referenceClientMap.remove(key);
            if (client != null) {
                try {
                    if (logger.isInfoEnabled()) {
```

```
                        logger.info("Close dubbo connect: " + client.getL
ocalAddress() + "-->" + client.getRemoteAddress());
                    }
                    client.close(getServerShutdownTimeout());
                } catch (Throwable t) {
                    logger.warn(t.getMessage(), t);
                }
            }
        }

        for (String key : new ArrayList<String>(ghostClientMap.keySet()))
 {
            ExchangeClient client = ghostClientMap.remove(key);
            if (client != null) {
                try {
                    if (logger.isInfoEnabled()) {
                        logger.info("Close dubbo connect: " + client.getL
ocalAddress() + "-->" + client.getRemoteAddress());
                    }
                    client.close(getServerShutdownTimeout());
                } catch (Throwable t) {
                    logger.warn(t.getMessage(), t);
                }
            }
        }
        stubServiceMethodsMap.clear();
        super.destroy();
    }


    //这里是修改的地方，原来为匿名内部类，修改为内部类，之后，其中添加了一个方法便于插
入代码
    class DubboProtocolExchangeHandlerAdapter extends ExchangeHandlerAdap
ter {
        public Object reply(ExchangeChannel channel, Object message) thro
ws RemotingException {
            if (message instanceof Invocation) {
                Invocation inv = (Invocation) message;
                Invoker<?> invoker = getInvoker(channel, inv);
                //如果是callback 需要处理高版本调用低版本的问题
                if (Boolean.TRUE.toString().equals(inv.getAttachments().g
et(IS_CALLBACK_SERVICE_INVOKE))) {
                    String methodsStr = invoker.getUrl().getParameters().
get("methods");
                    boolean hasMethod = false;
                    if (methodsStr == null || methodsStr.indexOf(",") ==
 -1) {
                        hasMethod = inv.getMethodName().equals(methodsSt
r);
                    } else {
                        String[] methods = methodsStr.split(",");
                        for (String method : methods) {
```

```
                            if (inv.getMethodName().equals(method)) {
                                hasMethod = true;
                                break;
                            }
                        }
                    }
                    if (!hasMethod) {
                        logger.warn(new IllegalStateException("The method
Name " + inv.getMethodName() + " not found in callback service interface
 ,invoke will be ignored. please update the api interface. url is:" + inv
oker.getUrl()) + " ,invocation is :" + inv);
                        return null;
                    }
                }
                RpcContext.getContext().setRemoteAddress(channel.getRemot
eAddress());

                return invoker.invoke(doBeforeInvokeReturn(inv));
            }
            throw new RemotingException(channel, "Unsupported request: "
+ message == null ? null : (message.getClass().getName() + ": " + messag
e) + ", channel: consumer: " + channel.getRemoteAddress() + " --> provide
r: " + channel.getLocalAddress());
        }

        //这里是新增的插入代码的方法
        public Invocation doBeforeInvokeReturn(Invocation inv){
            return inv;
        }

        @Override
        public void received(Channel channel, Object message) throws Remo
tingException {
            if (message instanceof Invocation) {
                reply((ExchangeChannel) channel, message);
            } else {
                super.received(channel, message);
            }
        }

        @Override
        public void connected(Channel channel) throws RemotingException {
            invoke(channel, Constants.ON_CONNECT_KEY);
        }

        @Override
        public void disconnected(Channel channel) throws RemotingExceptio
n {
            if (logger.isInfoEnabled()) {
                logger.info("disconected from " + channel.getRemoteAddres
s() + ",url:" + channel.getUrl());
            }
```

```
            invoke(channel, Constants.ON_DISCONNECT_KEY);
        }

        private void invoke(Channel channel, String methodKey) {
            Invocation invocation = createInvocation(channel, channel.get
Url(), methodKey);
            if (invocation != null) {
                try {
                    received(channel, invocation);
                } catch (Throwable t) {
                    logger.warn("Failed to invoke event method " + invoca
tion.getMethodName() + "(), cause: " + t.getMessage(), t);
                }
            }
        }

        private Invocation createInvocation(Channel channel, URL url, Str
ing methodKey) {
            String method = url.getParameter(methodKey);
            if (method == null || method.length() == 0) {
                return null;
            }

            RpcInvocation invocation = new RpcInvocation(method, new Clas
s<?>[0], new Object[0]);
            invocation.setAttachment(Constants.PATH_KEY, url.getPath());
            invocation.setAttachment(Constants.GROUP_KEY, url.getParamete
r(Constants.GROUP_KEY));
            invocation.setAttachment(Constants.INTERFACE_KEY, url.getPara
meter(Constants.INTERFACE_KEY));
            invocation.setAttachment(Constants.VERSION_KEY, url.getParame
ter(Constants.VERSION_KEY));
            if (url.getParameter(Constants.STUB_EVENT_KEY, false)) {
                invocation.setAttachment(Constants.STUB_EVENT_KEY, Boolea
n.TRUE.toString());
            }
            return invocation;
        }
    }
}
```