crazyhzm 265

Dubbo源码解析（二十二）远程调用——Protocol

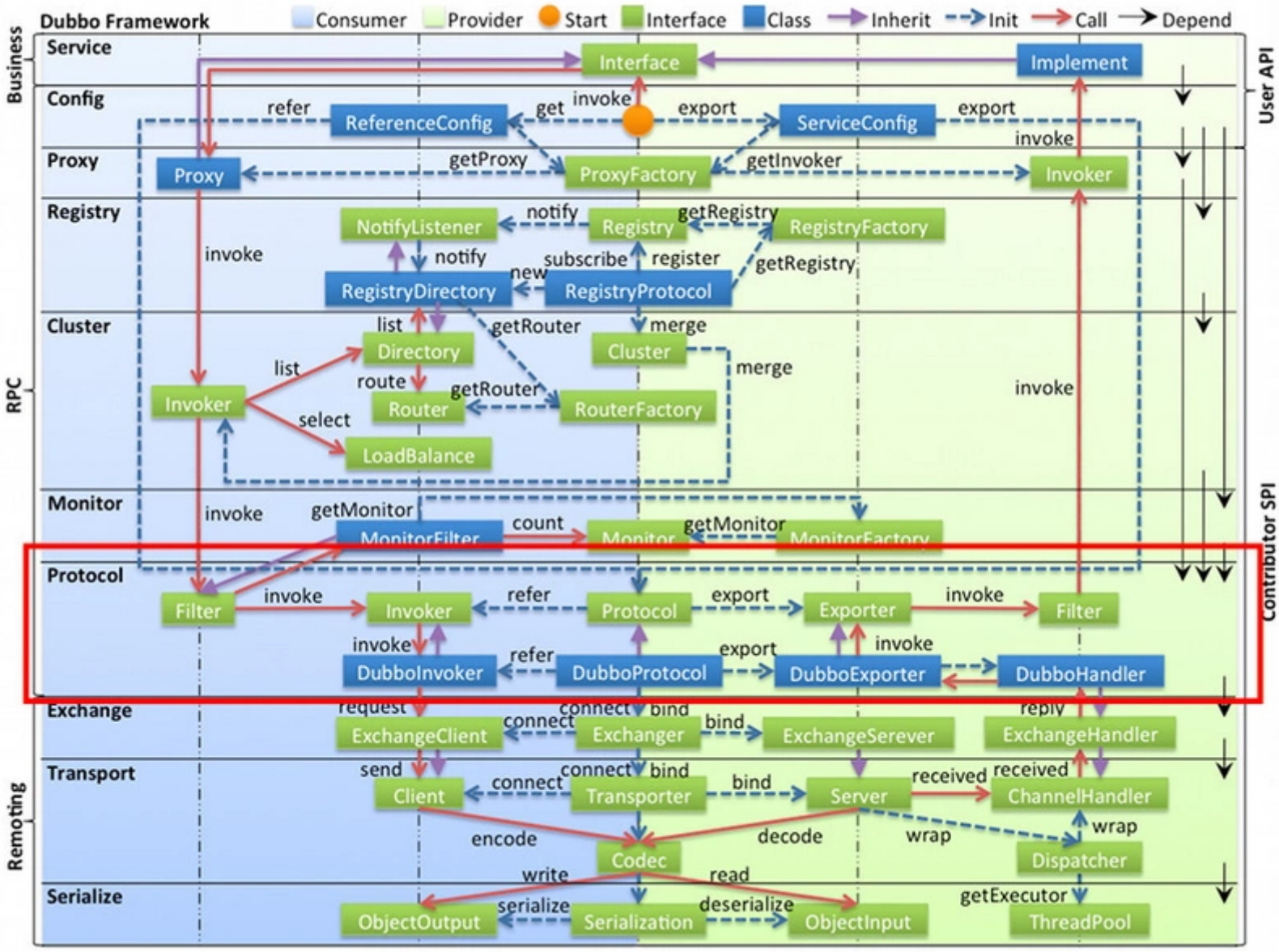
java dubbo 阅读约 30 分钟

远程调用——Protocol

> 目标：介绍远程调用中协议的设计和实现，介绍dubbo-rpc-api中的各种protocol包的源码，是重点内容。

前言

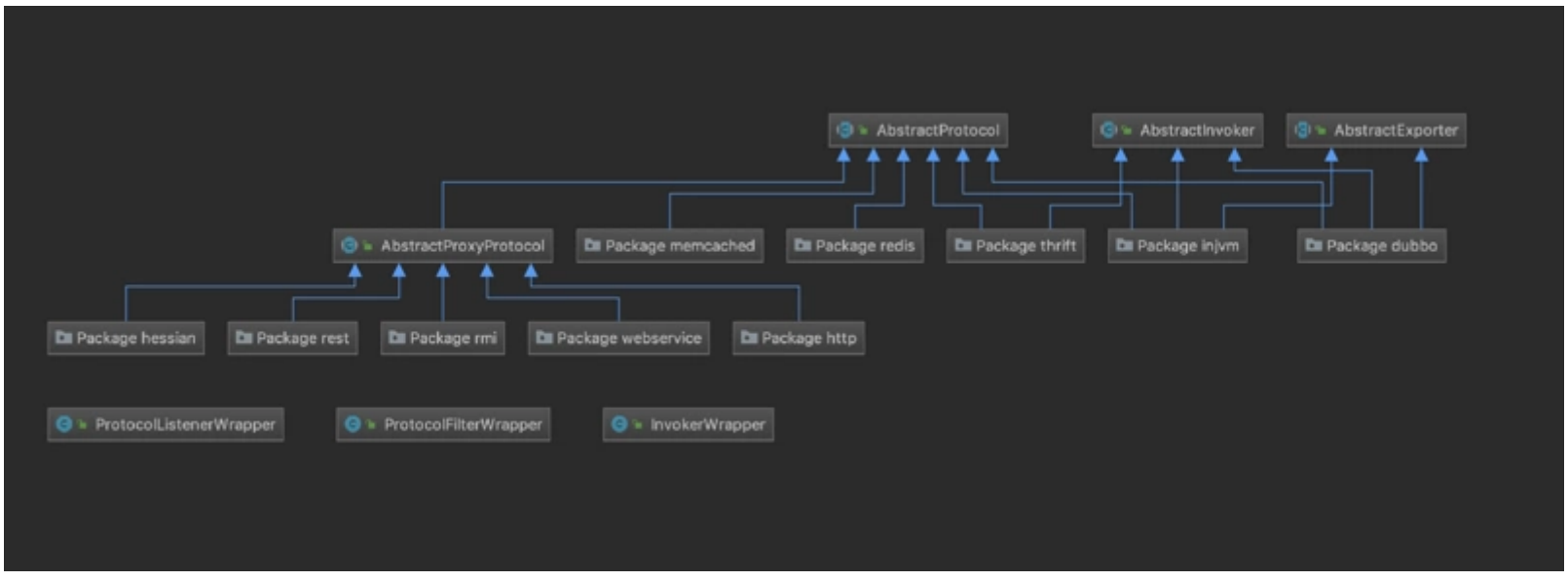
在远程调用中协议是非常重要的一层，看下面这张图：



该层是在信息交换层之上，分为了并且夹杂在服务暴露和服务引用中间，为了有一个约定的方式进行调用。

dubbo支持不同协议的扩展，比如http、thrift等等，具体的可以参照官方文档。本文讲解的源码大部分是对于公共方法的实现，而具体的服务暴露和服务引用会在各个协议实现中讲到。

下面是该包下面的类图：



源码分析

（一）AbstractProtocol

该类是协议的抽象类，实现了Protocol接口，其中实现了一些公共的方法，抽象方法在它的子类AbstractProxyProtocol中定义。

1.属性

```
/**
 * 服务暴露者集合
 */
protected final Map<String, Exporter<?>> exporterMap = new ConcurrentHashMap<String, Exporter<?>>();

/**
 * 服务引用者集合
 */
//TODO SOFEREFENCE
protected final Set<Invoker<?>> invokers = new ConcurrentHashSet<Invoker<?>>();
```

2.serviceKey

```
protected static String serviceKey(URL url) {
    // 获得绑定的端口号
    int port = url.getParameter(Constants.BIND_PORT_KEY, url.getPort());
    return serviceKey(port, url.getPath(), url.getParameter(Constants.VERSION_KEY),
            url.getParameter(Constants.GROUP_KEY));
}

protected static String serviceKey(int port, String serviceName, String serviceVersion, String serviceGroup) {
    return ProtocolUtils.serviceKey(port, serviceName, serviceVersion, serviceGroup);
}
```

该方法是为了得到服务key group+"/"+serviceName+":"+serviceVersion+":"+port

3.destroy

```
@Override
public void destroy() {
    // 遍历服务引用实体
    for (Invoker<?> invoker : invokers) {
        if (invoker != null) {
            // 从集合中移除
            invokers.remove(invoker);
            try {
                if (logger.isInfoEnabled()) {
                    logger.info("Destroy reference: " + invoker.getUrl());
                }
                // 销毁
                invoker.destroy();
            } catch (Throwable t) {
                logger.warn(t.getMessage(), t);
            }
        }
    }
    // 遍历服务暴露者
    for (String key : new ArrayList<String>(exporterMap.keySet())) {
        // 从集合中移除
        Exporter<?> exporter = exporterMap.remove(key);
        if (exporter != null) {
            try {
                if (logger.isInfoEnabled()) {
                    logger.info("Unexport service: " + exporter.getInvoker().getUrl());
```

该方法是对invoker和exporter的销毁。

（二）AbstractProxyProtocol

该类继承了AbstractProtocol类，其中利用了代理工厂对AbstractProtocol中的两个集合进行了填充，并且对异常做了处理。

1.属性

```
/**
 * rpc的异常类集合
 */
private final List<Class<?>> rpcExceptions = new CopyOnWriteArrayList<Class<?>>();

/**
 * 代理工厂
 */
private ProxyFactory proxyFactory;
```

2.export

```
@Override
@SuppressWarnings("unchecked")
public <T> Exporter<T> export(final Invoker<T> invoker) throws RpcException {
    // 获得uri
    final String uri = serviceKey(invoker.getUrl());
    // 获得服务暴露者
    Exporter<T> exporter = (Exporter<T>) exporterMap.get(uri);
    if (exporter != null) {
        return exporter;
    }
    // 新建一个线程
    final Runnable runnable = doExport(proxyFactory.getProxy(invoker, true), invoker.getInterface(),
invoker.getUrl());
    exporter = new AbstractExporter<T>(invoker) {
        /**
         * 取消暴露
         */
        @Override
        public void unexport() {
            super.unexport();
            // 移除该key对应的服务暴露者
            exporterMap.remove(uri);
            if (runnable != null) {
                try {
                    // 启动线程
                    runnable.run();
```

其中分为两个步骤，创建一个exporter，放入到集合汇中。在创建exporter时对unexport方法进行了重写。

3.refer

```
@Override
public <T> Invoker<T> refer(final Class<T> type, final URL url) throws RpcException {
    // 通过代理获得实体域
    final Invoker<T> target = proxyFactory.getInvoker(doRefer(type, url), type, url);
    Invoker<T> invoker = new AbstractInvoker<T>(type, url) {
        @Override
        protected Result doInvoke(Invocation invocation) throws Throwable {
            try {
                // 获得调用结果
                Result result = target.invoke(invocation);
                Throwable e = result.getException();
                // 如果抛出异常，则抛出相应异常
                if (e != null) {
                    for (Class<?> rpcException : rpcExceptions) {
                        if (rpcException.isAssignableFrom(e.getClass())) {
                            throw getRpcException(type, url, invocation, e);
                        }
                    }
                }
                return result;
            } catch (RpcException e) {
                // 抛出未知异常
                if (e.getCode() == RpcException.UNKNOWN_EXCEPTION) {
                    e.setCode(getErrorCode(e.getCause()));
                }
                throw e;
```

该方法是服务引用，先从代理工厂中获得Invoker对象target，然后创建了真实的invoker在重写方法中调用代理的方法，最后加入到集合。

```
protected abstract <T> Runnable doExport(T impl, Class<T> type, URL url) throws RpcException;

protected abstract <T> T doRefer(Class<T> type, URL url) throws RpcException;
```

可以看到其中抽象了服务引用和暴露的方法，让各类协议各自实现。

（三）AbstractInvoker

该类是invoker的抽象方法，因为协议被夹在服务引用和服务暴露中间，无论什么协议都有一些通用的Invoker和exporter的方法实现，而该类就是实现了Invoker的公共方法，而把doInvoke抽象出来，让子类只关注这个方法。

1.属性

```
/**
 * 服务类型
 */
private final Class<T> type;

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

/**
 * 附加值
 */
private final Map<String, String> attachment;

/**
 * 是否可用
 */
private volatile boolean available = true;

/**
 *  是否销毁
 */
private AtomicBoolean destroyed = new AtomicBoolean(false);
```

2.convertAttachment

```
private static Map<String, String> convertAttachment(URL url, String[] keys) {
    if (keys == null || keys.length == 0) {
        return null;
    }
    Map<String, String> attachment = new HashMap<String, String>();
    // 遍历key，把值放入附加值集合中
    for (String key : keys) {
        String value = url.getParameter(key);
        if (value != null && value.length() > 0) {
            attachment.put(key, value);
        }
    }
    return attachment;
}
```

该方法是转化为附加值，把url中的值转化为服务调用invoker的附加值。

3.invoke

```
@Override
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 (attachment != null && attachment.size() > 0) {
        invocation.addAttachmentsIfAbsent(attachment);
    }
    // 把上下文的附加值放入会话域
    Map<String, String> contextAttachments = RpcContext.getContext().getAttachments();
    if (contextAttachments != null && contextAttachments.size() != 0) {
        /**
         * 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
```

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。

（四）AbstractExporter

该类和AbstractInvoker类似，也是在服务暴露中实现了一些公共方法。

1.属性

```
/**
 * 实体域
 */
private final Invoker<T> invoker;

/**
 * 是否取消暴露服务
 */
private volatile boolean unexported = false;
```

2.unexport

```
@Override
public void unexport() {
    // 如果已经消取消暴露，则之间返回
    if (unexported) {
        return;
    }
    // 设置为true
    unexported = true;
    // 销毁该实体域
    getInvoker().destroy();
}
```

（五）InvokerWrapper

该类是Invoker的包装类，其中用到类装饰模式，不过并没有实现实际的功能增强。

```
public class InvokerWrapper<T> implements Invoker<T> {

    /**
     * invoker对象
     */
    private final Invoker<T> invoker;

    private final URL url;

    public InvokerWrapper(Invoker<T> invoker, URL url) {
        this.invoker = invoker;
        this.url = url;
    }

    @Override
    public Class<T> getInterface() {
        return invoker.getInterface();
    }

    @Override
    public URL getUrl() {
        return url;
    }

    @Override
    public boolean isAvailable() {
```

（六）ProtocolFilterWrapper

该类实现了Protocol接口，其中也用到了装饰模式，是对Protocol的装饰，是在服务引用和暴露的方法上加上了过滤器功能。

1.buildInvokerChain

```
private static <T> Invoker<T> buildInvokerChain(final Invoker<T> invoker, String key, String group) {
    Invoker<T> last = invoker;
    // 获得过滤器的所有扩展实现类实例集合
    List<Filter> filters =
ExtensionLoader.getExtensionLoader(Filter.class).getActivateExtension(invoker.getUrl(), key, group);
    if (!filters.isEmpty()) {
        // 从最后一个过滤器开始循环，创建一个带有过滤器链的invoker对象
        for (int i = filters.size() - 1; i >= 0; i--) {
            final Filter filter = filters.get(i);
            // 记录last的invoker
            final Invoker<T> next = last;
            // 新建last
            last = new Invoker<T>() {

                @Override
                public Class<T> getInterface() {
                    return invoker.getInterface();
                }

                @Override
                public URL getUrl() {
                    return invoker.getUrl();
                }

                @Override
                public boolean isAvailable() {
```

该方法就是创建带 Filter 链的 Invoker 对象。倒序的把每一个过滤器串连起来，形成一个invoker。

2.export

```
@Override
public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
    // 如果是注册中心，则直接暴露服务
    if (Constants.REGISTRY_PROTOCOL.equals(invoker.getUrl().getProtocol())) {
        return protocol.export(invoker);
    }
    // 服务提供侧暴露服务
    return protocol.export(buildInvokerChain(invoker, Constants.SERVICE_FILTER_KEY, Constants.PROVIDER));
}
```

该方法是在服务暴露上做了过滤器链的增强，也就是加上了过滤器。

3.refer

```
@Override
public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {
    // 如果是注册中心，则直接引用
    if (Constants.REGISTRY_PROTOCOL.equals(url.getProtocol())) {
        return protocol.refer(type, url);
    }
    // 消费者侧引用服务
    return buildInvokerChain(protocol.refer(type, url), Constants.REFERENCE_FILTER_KEY, Constants.CONSUMER);
}
```

该方法是在服务引用上做了过滤器链的增强，也就是加上了过滤器。

（七）ProtocolListenerWrapper

该类也实现了Protocol，也是装饰了Protocol接口，但是它是在服务引用和暴露过程中加上了监听器的功能。

1.export

```
@Override
public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
    // 如果是注册中心，则暴露该invoker
    if (Constants.REGISTRY_PROTOCOL.equals(invoker.getUrl().getProtocol())) {
        return protocol.export(invoker);
    }
    // 创建一个暴露者监听器包装类对象
    return new ListenerExporterWrapper<T>(protocol.export(invoker),
            Collections.unmodifiableList(ExtensionLoader.getExtensionLoader(ExporterListener.class)
                    .getActivateExtension(invoker.getUrl(), Constants.EXPORTER_LISTENER_KEY)));
}
```

该方法是在服务暴露上做了监听器功能的增强，也就是加上了监听器。

2.refer

```
@Override
public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {
    // 如果是注册中心。则直接引用服务
    if (Constants.REGISTRY_PROTOCOL.equals(url.getProtocol())) {
        return protocol.refer(type, url);
    }
    // 创建引用服务监听器包装类对象
    return new ListenerInvokerWrapper<T>(protocol.refer(type, url),
            Collections.unmodifiableList(
                    ExtensionLoader.getExtensionLoader(InvokerListener.class)
                            .getActivateExtension(url, Constants.INVOKER_LISTENER_KEY)));
}
```

该方法是在服务引用上做了监听器功能的增强，也就是加上了监听器。

后记

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

该文章讲解了远程调用中关于协议的部分，其实就是讲了一些公共的方法，并且把关键方法抽象出来让子类实现，具体的方法实现都在各个协议中自己实现。接下来我将开始对rpc模块的代理进行讲解。

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——概要篇

这次源码解析借鉴《肥朝》前辈的dubbo源码解析，进行源码学习。总结起来就是先总体,后局部.也就是先把需要注意的概念先抛...

• 阅读 631

Dubbo 源码分析 - SPI 机制

SPI全称为ServiceProviderInterface，是Java提供的一种服务发现机制。SPI的本质是将接口实现类的全限定名配置在文件中，并由服...

coolblog • 阅读 250

React源码分析(二)

上一篇文章讲到了React调用ReactDOM.render首次渲染组件的前几个过程的源码,包括创建元素、根据元素实例化对应组件,利用事...

莫凡 • 阅读 43

vuex源码分析（二）

继上面讲完contructor函数，我们继续来讲后面的内容get和set方法都有es6里边的语法，get语法是将对象属性绑定到查询该属性...

云中歌 • 阅读 1

dubbo源码解析（一）Hello,Dubbo

你好，dubbo，初次见面，我想和你交个朋友。先给出一套官方的说法：ApacheDubbo是一款高性能、轻量级基于Java的RPC开源...

CrazyHzm • 阅读 633

结合Dubbo源码分析Spi

如前所说，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