crazyhzm 265

Dubbo源码解析（十九）远程调用——开篇

java rpc dubbo 阅读约 34 分钟

远程调用——开篇

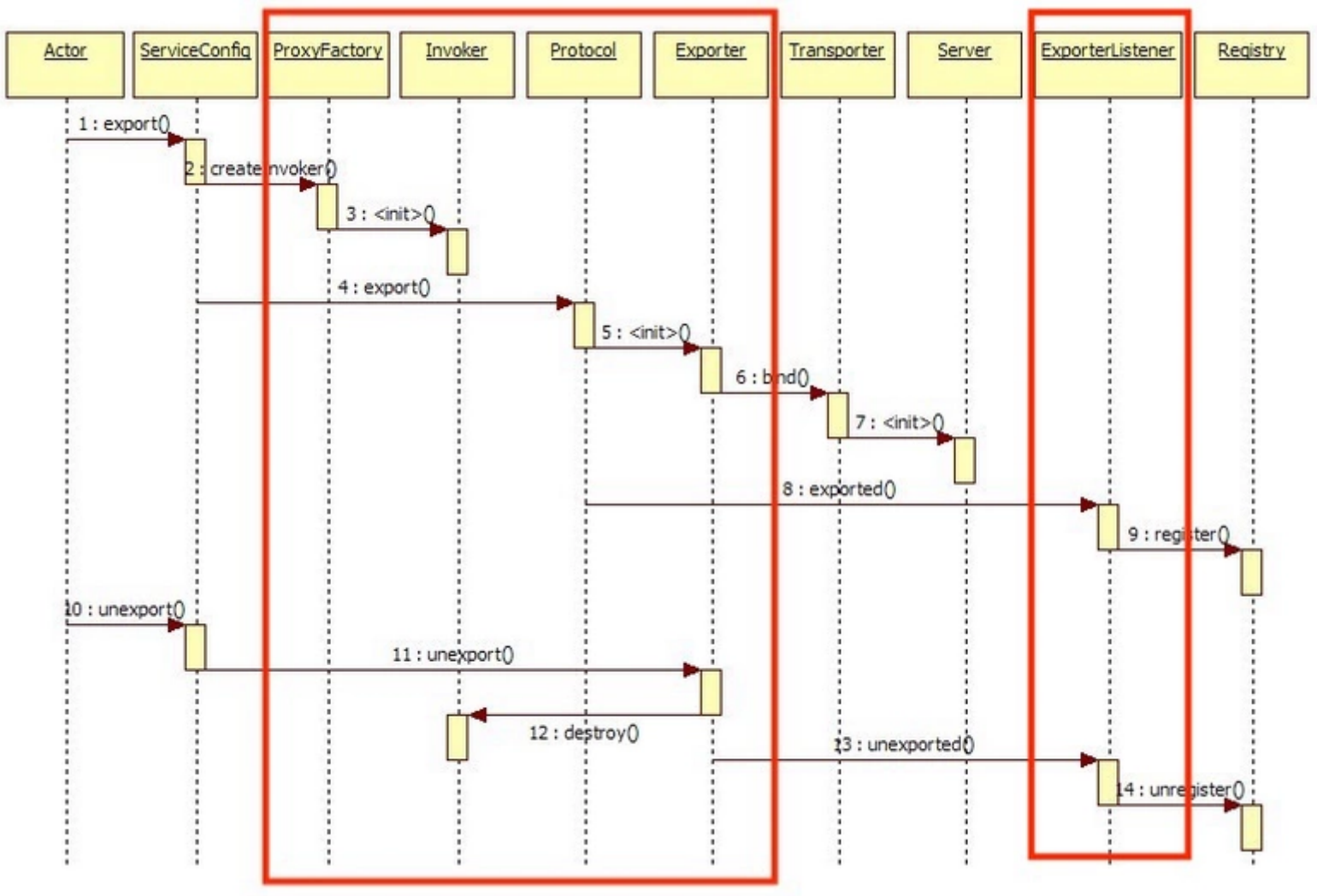
> 目标：介绍之后解读远程调用模块的内容如何编排、介绍dubbo-rpc-api中的包结构设计以及最外层的的源码解析。

前言

最近我面临着一个选择，因为dubbo 2.7.0-release出现在了仓库里，最近一直在进行2.7.0版本的code review，那我之前说这一系列的文章都是讲述2.6.x版本的源代码，我现在要不要选择直接开始讲解2.7.0的版本的源码呢？我最后还是决定继续讲解2.6.x，因为我觉得还是有很多公司在用着2.6.x的版本，并且对于升级2.7.0的计划应该还没那么快，并且在了解2.6.x版本的原理后，再去了解2.7.0新增的特性会更加容易，也能够品位到设计者的意图。当然在结束2.6.x的重要模块讲解后，我也会对2.7.0的新特性以及实现原理做一个全面的分析，2.7.0作为dubbo社区的毕业版，更加强大，敬请期待。

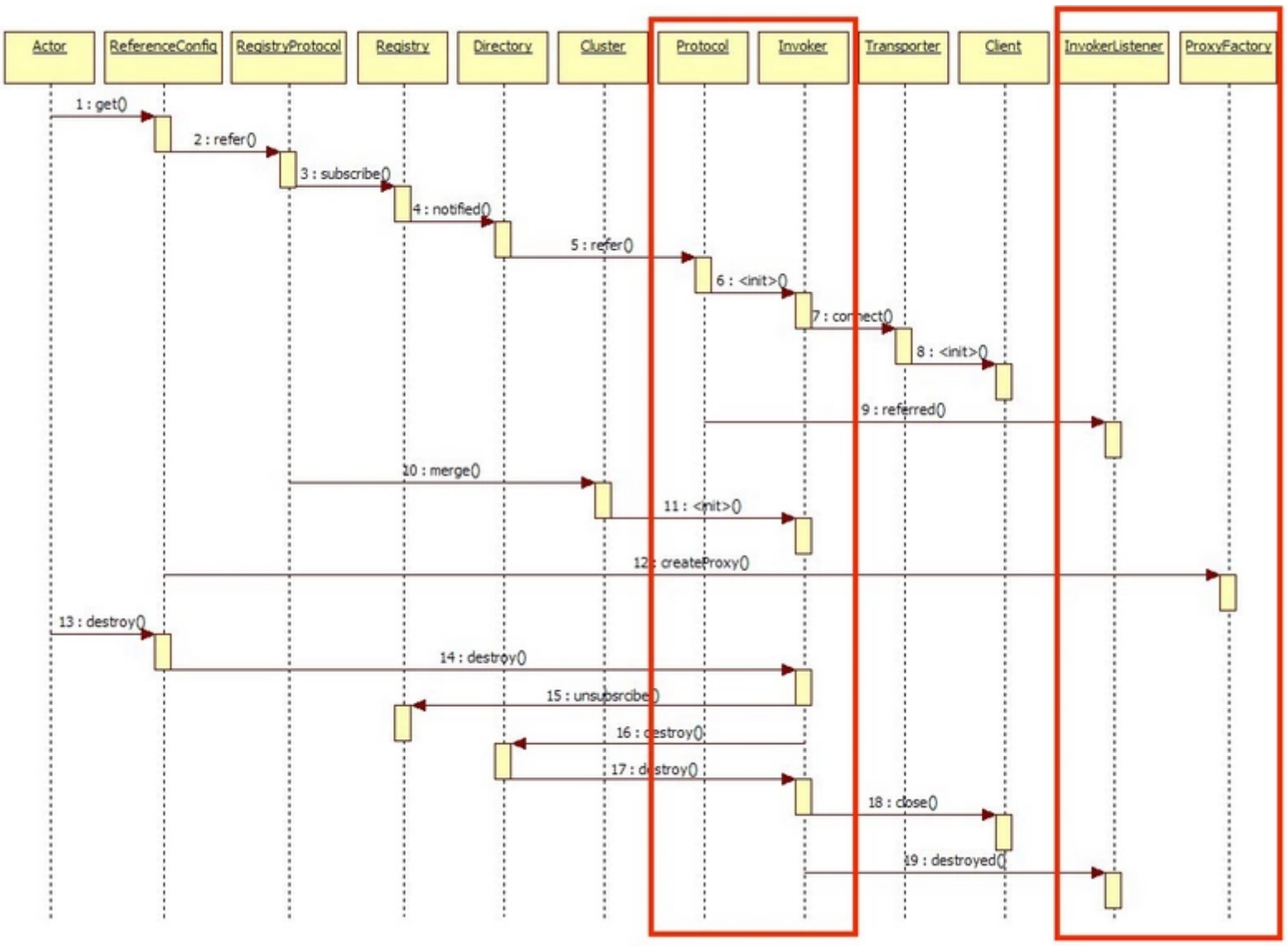
前面讲了很多的内容，现在开始将远程调用RPC，好像又回到我第一篇文章《dubbo源码解析（一）Hello,Dubbo》，在这篇文章开头我讲到了什么叫做RPC，再通俗一点讲，就是我把一个项目的两部分代码分开来，分别放到两台机器上，当我部署在A服务器上的应用想要调用部署在B服务器上的应用等方法，由于不存在同一个内存空间，不能直接调用。而其实整个dubbo都在做远程调用的事情，它涉及到很多内容，比如配置、代理、集群、监控等等，那么这次讲的内容是只关心一对一的调用，dubbo-rpc远程调用模块抽象各种协议，以及动态代理，Proxy层和Protocol层rpc的核心，我将会在本系列中讲到。下面我们来看两张官方文档的图：

1. 暴露服务的时序图：



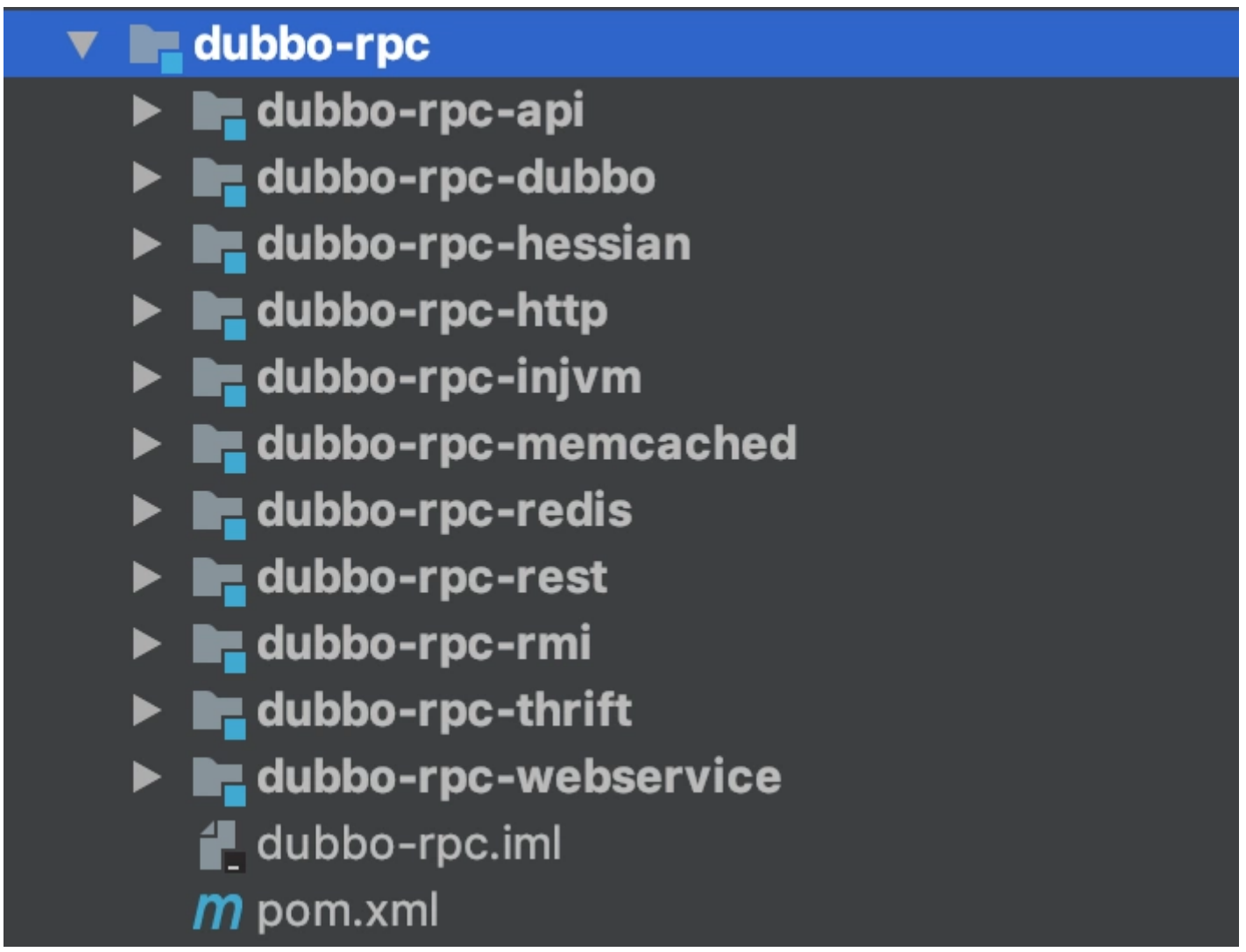
你会发现其中有我们以前讲到的Transporter、Server、Registry，而这次的系列将会讲到的就是红色框框内的部分。

1. 引用服务时序图

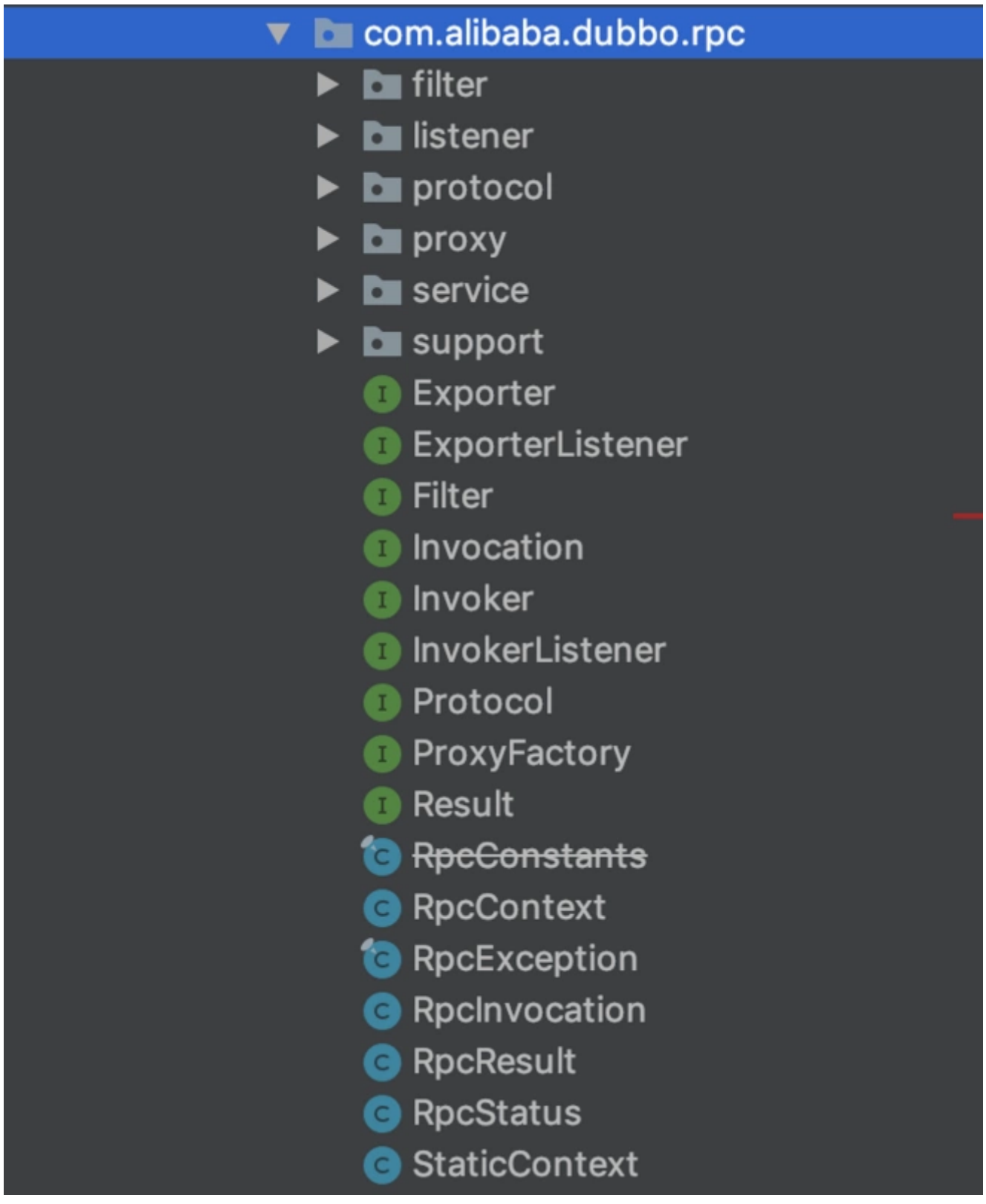


在引用服务时序图中，对应的也是红色框框的部分。

当阅读完该系列后，希望能对这个调用链有所感悟。接下来看看dubbo-rpc的包结构：



可以看到有很多包，很规整，其中dubbo-rpc-api是对协议、暴露、引用、代理等的抽象和实现，是rpc整个设计的核心内容。其他的包则是dubbo支持的9种协议，在官方文档也能查看介绍，并且包括一种本地调用injvm。那么我们再来看看dubbo-rpc-api中包结构：



1. filter包：在进行服务引用时会进行一系列的过滤。其中包括了很多过滤器。
2. listener包：看上面两张服务引用和服务暴露的时序图，发现有两个listener，其中的逻辑实现就在这个包内
3. protocol包：这个包实现了协议的一些公共逻辑
4. proxy包：实现了代理的逻辑。
5. service包：其中包含了一个需要调用的方法等封装抽象。
6. support包：包括了工具类
7. 最外层的实现。

下面的篇幅设计，本文会讲解最外层的源码和service下的源码，support包下的源码我会穿插在其他用到的地方一并讲解，filter、listener、protocol、proxy以及各类协议的实现各自用一篇来讲。

源码分析

（一）Invoker

```java
public interface Invoker<T> extends Node {

    /**
     * get service interface.
     * 获得服务接口
     * @return service interface.
     */
    Class<T> getInterface();

    /**
     * invoke.
     * 调用下一个会话域
     * @param invocation
     * @return result
     * @throws RpcException
     */
    Result invoke(Invocation invocation) throws RpcException;

}
```

该接口是实体域，它是dubbo的核心模型，其他模型都向它靠拢，或者转化成它，它代表了一个可执行体，可以向它发起invoke调用，这个有可能是一个本地的实现，也可能是一个远程的实现，也可能是一个集群的实现。它代表了一次调用

(二) Invocation

```java
public interface Invocation {

    /**
     * get method name.
     * 获得方法名称
     * @return method name.
     * @serial
     */
    String getMethodName();

    /**
     * get parameter types.
     * 获得参数类型
     * @return parameter types.
     * @serial
     */
    Class<?>[] getParameterTypes();

    /**
     * get arguments.
     * 获得参数
     * @return arguments.
     * @serial
     */
    Object[] getArguments();
```

Invocation 是会话域，它持有调用过程中的变量，比如方法名，参数等。

(三) Exporter

```java
public interface Exporter<T> {

    /**
     * get invoker.
     * 获得对应的实体域invoker
     * @return invoker
     */
    Invoker<T> getInvoker();

    /**
     * unexport.
     * 取消暴露
     * <p>
     * <code>
     * getInvoker().destroy();
     * </code>
     */
    void unexport();

}
```

该接口是暴露服务的接口，定义了两个方法分别是获得invoker和取消暴露服务。

(四) ExporterListener

```java
@SPI
public interface ExporterListener {

    /**
     * The exporter exported.
     * 暴露服务
     * @param exporter
     * @throws RpcException
     * @see com.alibaba.dubbo.rpc.Protocol#export(Invoker)
     */
    void exported(Exporter<?> exporter) throws RpcException;

    /**
     * The exporter unexported.
     * 取消暴露
     * @param exporter
     * @throws RpcException
     * @see com.alibaba.dubbo.rpc.Exporter#unexport()
     */
    void unexported(Exporter<?> exporter);

}
```

该接口是服务暴露的监听器接口，定义了两个方法是暴露和取消暴露，参数都是Exporter类型的。

（五）Protocol

```
     * Refer a remote service: <br>
     * 1. When user calls `invoke()` method of `Invoker` object which's returned from `refer()` call, the
protocol
     * needs to correspondingly execute `invoke()` method of `Invoker` object <br>
     * 2. It's protocol's responsibility to implement `Invoker` which's returned from `refer()`. Generally
speaking,
     * protocol sends remote request in the `Invoker` implementation. <br>
     * 3. When there's check=false set in URL, the implementation must not throw exception but try to recover
when
     * connection fails.
     * 引用服务方法
     * @param <T>  Service type 服务类型
     * @param type Service class 服务类名
     * @param url  URL address for the remote service
     * @return invoker service's local proxy
     * @throws RpcException when there's any error while connecting to the service provider
     */
    @Adaptive
    <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException;

    /**
     * Destroy protocol: <br>
     * 1. Cancel all services this protocol exports and refers <br>
     * 2. Release all occupied resources, for example: connection, port, etc. <br>
     * 3. Protocol can continue to export and refer new service even after it's destroyed.
     */
```

该接口是服务域接口，也是协议接口，它是一个可扩展的接口，默认实现的是dubbo协议。定义了四个方法，关键的是服务暴露和引用两个方法。

（六）Filter

```
@SPI
public interface Filter {

    /**
     * do invoke filter.
     * <p>
     * <code>
     * // before filter
     * Result result = invoker.invoke(invocation);
     * // after filter
     * return result;
     * </code>
     *
     * @param invoker    service
     * @param invocation invocation.
     * @return invoke result.
     * @throws RpcException
     * @see com.alibaba.dubbo.rpc.Invoker#invoke(Invocation)
     */
    Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException;

}
```

该接口是invoker调用时过滤器接口，其中就只有一个invoke方法。在该方法中对调用进行过滤

（七）InvokerListener

```java
@SPI
public interface InvokerListener {

    /**
     * The invoker referred
     * 在服务引用的时候进行监听
     * @param invoker
     * @throws RpcException
     * @see com.alibaba.dubbo.rpc.Protocol#refer(Class, com.alibaba.dubbo.common.URL)
     */
    void referred(Invoker<?> invoker) throws RpcException;

    /**
     * The invoker destroyed.
     * 销毁实体域
     * @param invoker
     * @see com.alibaba.dubbo.rpc.Invoker#destroy()
     */
    void destroyed(Invoker<?> invoker);

}
```

该接口是实体域的监听器，定义了两个方法，分别是服务引用和销毁的时候执行的方法。

（八）Result

该接口是实体域执行invoke的结果接口，里面定义了获得结果异常以及附加值等方法。比较好理解我就不贴代码了。

（九）ProxyFactory

```java
@SPI("javassist")
public interface ProxyFactory {

    /**
     * create proxy.
     * 创建一个代理
     * @param invoker
     * @return proxy
     */
    @Adaptive({Constants.PROXY_KEY})
    <T> T getProxy(Invoker<T> invoker) throws RpcException;

    /**
     * create proxy.
     * 创建一个代理
     * @param invoker
     * @return proxy
     */
    @Adaptive({Constants.PROXY_KEY})
    <T> T getProxy(Invoker<T> invoker, boolean generic) throws RpcException;

    /**
     * create invoker.
     * 创建一个实体域
     * @param <T>
     * @param proxy
```

该接口是代理工厂接口，它也是个可扩展接口，默认实现javassist，dubbo提供两种动态代理方法分别是javassist/jdk，该接口定义了三个方法，前两个方法是通过invoker创建代理，最后一个是通过代理来获得invoker。

（十）RpcContext

该类就是远程调用的上下文，贯穿着整个调用，例如A调用B，然后B调用C。在服务B上，RpcContext在B之前将调用信息从A保存到B。开始调用C，并在B调用C后将调用信息从B保存到C。RpcContext保存了调用信息。

```
public class RpcContext {

    /**
     * use internal thread local to improve performance
     * 本地上下文
     */
    private static final InternalThreadLocal<RpcContext> LOCAL = new InternalThreadLocal<RpcContext>() {
        @Override
        protected RpcContext initialValue() {
            return new RpcContext();
        }
    };
    /**
     * 服务上下文
     */
    private static final InternalThreadLocal<RpcContext> SERVER_LOCAL = new InternalThreadLocal<RpcContext>() {
        @Override
        protected RpcContext initialValue() {
            return new RpcContext();
        }
    };

    /**
     * 附加值集合
     */
    private final Map<String, String> attachments = new HashMap<String, String>();
```

该类中最重要的是它的一些属性，因为该上下文就是用来保存信息的。方法我就不介绍了，因为比较简单。

（十一）RpcException

```
/**
 * 不知道异常
 */
public static final int UNKNOWN_EXCEPTION = 0;
/**
 * 网络异常
 */
public static final int NETWORK_EXCEPTION = 1;
/**
 * 超时异常
 */
public static final int TIMEOUT_EXCEPTION = 2;
/**
 * 基础异常
 */
public static final int BIZ_EXCEPTION = 3;
/**
 * 禁止访问异常
 */
public static final int FORBIDDEN_EXCEPTION = 4;
/**
 * 序列化异常
 */
public static final int SERIALIZATION_EXCEPTION = 5;
```

该类是rpc调用抛出的异常类，其中封装了五种通用的错误码。

（十二）RpcInvocation

```
/**
 * 方法名称
 */
private String methodName;

/**
 * 参数类型集合
 */
private Class<?>[] parameterTypes;

/**
 * 参数集合
 */
private Object[] arguments;

/**
 * 附加值
 */
private Map<String, String> attachments;

/**
 * 实体域
 */
private transient Invoker<?> invoker;
```

该类实现了Invocation接口，是rpc的会话域，其中的方法比较简单，主要是封装了上述的属性。

（十三）RpcResult

```
/**
 * 结果
 */
private Object result;

/**
 * 异常
 */
private Throwable exception;

/**
 * 附加值
 */
private Map<String, String> attachments = new HashMap<String, String>();
```

该类实现了Result接口，是rpc的结果实现类，其中关键是封装了以上三个属性。

（十四）RpcStatus

该类是rpc的一些状态监控，其中封装了许多的计数器，用来记录rpc调用的状态。

1.属性

```
/**
 * uri对应的状态集合，key为uri，value为RpcStatus对象
 */
private static final ConcurrentMap<String, RpcStatus> SERVICE_STATISTICS = new ConcurrentHashMap<String,
RpcStatus>();

/**
 * method对应的状态集合，key是uri，第二个key是方法名methodName
 */
private static final ConcurrentMap<String, ConcurrentMap<String, RpcStatus>> METHOD_STATISTICS = new
ConcurrentHashMap<String, ConcurrentMap<String, RpcStatus>>();
/**
 * 已经没用了
 */
private final ConcurrentMap<String, Object> values = new ConcurrentHashMap<String, Object>();
/**
 * 活跃状态
 */
private final AtomicInteger active = new AtomicInteger();
/**
 * 总的数量
 */
private final AtomicLong total = new AtomicLong();
/**
 * 失败的个数
 */
```

以上是该类的属性，可以看到保存了很多的计数器，分别用来记录了失败调用成功调用等累计数。

2.beginCount

```
/**
 * 开始计数
 * @param url
 */
public static void beginCount(URL url, String methodName) {
    // 对该url对应对活跃计数器加一
    beginCount(getStatus(url));
    // 对该方法对活跃计数器加一
    beginCount(getStatus(url, methodName));
}

/**
 * 以原子方式加1
 * @param status
 */
private static void beginCount(RpcStatus status) {
    status.active.incrementAndGet();
}
```

该方法是增加计数。

3.endCount

```
public static void endCount(URL url, String methodName, long elapsed, boolean succeeded) {
    // url对应的状态中计数器减一
    endCount(getStatus(url), elapsed, succeeded);
    // 方法对应的状态中计数器减一
    endCount(getStatus(url, methodName), elapsed, succeeded);
}

private static void endCount(RpcStatus status, long elapsed, boolean succeeded) {
    // 活跃计数器减一
    status.active.decrementAndGet();
    // 总计数器加1
    status.total.incrementAndGet();
    // 总调用时长加上调用时长
    status.totalElapsed.addAndGet(elapsed);
    // 如果最大调用时长小于elapsed，则设置最大调用时长
    if (status.maxElapsed.get() < elapsed) {
        status.maxElapsed.set(elapsed);
    }
    // 如果rpc调用成功
    if (succeeded) {
        // 如果成最大调用成功时长小于elapsed，则设置最大调用成功时长
        if (status.succeededMaxElapsed.get() < elapsed) {
            status.succeededMaxElapsed.set(elapsed);
        }
    } else {
        // 失败计数器加一
```

该方法是计数器减少。

（十五）StaticContext

该类是系统上下文，仅供内部使用。

```
/**
 * 系统名称
 */
private static final String SYSTEMNAME = "system";
/**
 * 系统上下文集合，仅供内部使用
 */
private static final ConcurrentMap<String, StaticContext> context_map = new ConcurrentHashMap<String, StaticContext>
();
/**
 * 系统上下文名称
 */
private String name;
```

上面是该类的属性，它还记录了所有的系统上下文集合。

（十六）EchoService

```
public interface EchoService {

    /**
     * echo test.
     * 回声测试
     * @param message message.
     * @return message.
     */
    Object $echo(Object message);

}
```

该接口是回声服务接口，定义了一个一个回声测试的方法，回声测试用于检测服务是否可用，回声测试按照正常请求流程执行，能够测试整个调用是否通畅，可用于监控，所有服务自动实现该接口，只需将任意服务强制转化为EchoService，就可以用了。

（十七）GenericException

该方法是通用的异常类。

```
/**
 * 异常类名
 */
private String exceptionClass;

/**
 * 异常信息
 */
private String exceptionMessage;
```

比较简单，就封装了两个属性。

（十八）GenericService

```
public interface GenericService {

    /**
     * Generic invocation
     * 通用的会话域
     * @param method         Method name, e.g. findPerson. If there are overridden methods, parameter info is
     *                       required, e.g. findPerson(java.lang.String)
     * @param parameterTypes Parameter types
     * @param args           Arguments
     * @return invocation return value
     * @throws Throwable potential exception thrown from the invocation
     */
    Object $invoke(String method, String[] parameterTypes, Object[] args) throws GenericException;

}
```

该接口是通用的服务接口，同样定义了一个类似invoke的方法

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的开篇，介绍之后解读远程调用模块的内容如何编排、介绍dubbo-rpc-api中的包结构设计以及最外层的的源码解析，其中的逻辑不负责，要关注的是其中的一些概念和dubbo如何去做暴露服务和引用服务，其中很多的接口定义需要弄清楚。接下来我将开始对rpc模块的过滤器进行讲解。