njmbwene lk@(dubbo_group)
http://shiyanjun.cn/archives/325.html
http://blog.csdn.net/killuazoldyck
http://blog.csdn.net/meilong_whpu
http://blog.csdn.net/u010311445/article/category/2745121

http://blog.csdn.net/wangligang85/article/details/46430267

https://blog.csdn.net/pentiumchen/article/category/6528648
【dubbo编解码】
https://blog.csdn.net/prestigeding/article/details/81436774
【dubbo源码分析】
https://blog.csdn.net/u013076044/article/category/7230821
【dubbo SPI扩展】
https://blog.csdn.net/yangxiaobo118/article/details/80696830
【dubbo与netty结合】
http://blog.51cto.com/13933361/2164498

Dubbo源码解析

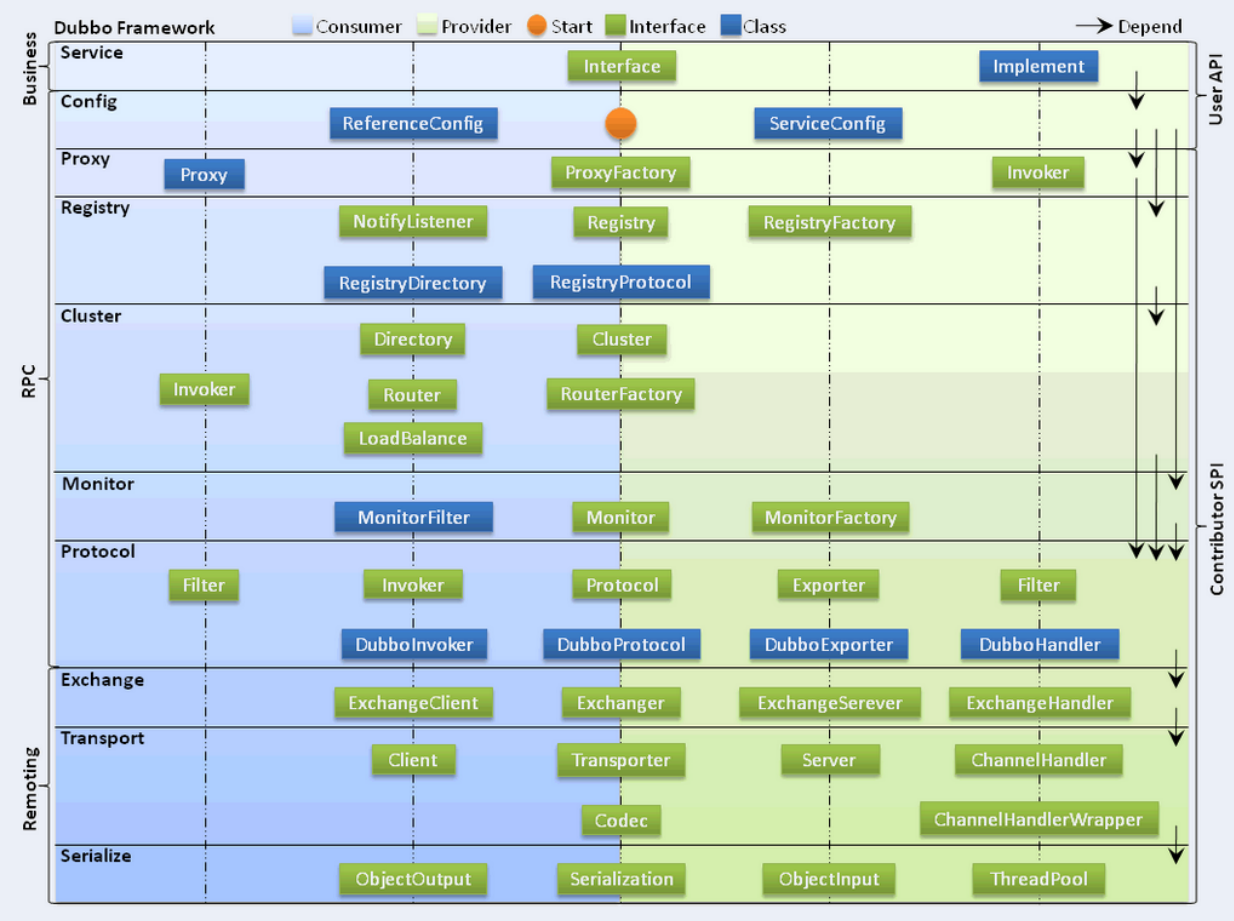
dubbo_group

1. 服务暴露描述
    - ServiceBean
        - 实现了InitializingBean，Spring 容器回调ServiceBean的afterPropertiesSet方法，并检查dubbo的配置，后调用ServiceConfig的export方法
    - ServiceConfig
        - export方法中判断是否延迟注册，如果延迟注册，则启动一个守护线程进行延迟注册，然后调用doExport方法
        - doExport方法中判断配置是否为空，最后调用doExportUrls
        - doExportUrls方法加载所有的注册中心，遍历调用doExportUrlsFor1Protocol对不同的协议进行注册
        - doExportUrlsFor1Protocol方法中检查端口是否可用，组装配置参数用于生成url，通过代理工厂将url转化成invoker对象，然后调用具体协议进行注册，如RegistryProtocol的export方法
    - RegistryProtocol
        - export方法中调用doLocalExport，doLocalExport方法中包装拦截器链，最后调用DubboProtocol的export方法进行本地服务暴露
        - 然后调用FailbackRegistry的register进行注册，真正的实现在ZookeeperRegistry的doRegister中实现
        - 然后再调用subscribe的subscribe进行订阅，真正的订阅在ZookeeperRegistry的doSubscribe中实现
    - DubboProtocol
        - export方法中调用openServer开启netty服务，如果服务不存在createServer方法创建一个新连接，然后调用Exchangers的bind方法

        - Exchangers的bind方法getExchanger(url)得到的默认为HeaderExchanger，HeaderExchanger的bind方法又调用了Transporters中的bind方法，最后调用NettyTransporter的bind方法创建一个NettyServer，在doOpen中进行netty的配置与开启

2. 服务引用过程
    - ReferenceBean
        - 实现了FactoryBean，在getObject方法中调用ReferenceConfig的get方法
    - ReferenceConfig
        - get方法调用init方法，init方法中检查配置，组装配置调用createProxy方法生成代理
        - 根据组装的参数生成URL，通过注册中心配置拼装URL，如果urls只包含一个URL，直接调用RegistryProtocol协议的refer方法进行注册与订阅，然后加入集群容错策略FailoverCluster，最后生成一个invoker
        - 最后调用JavassistProxyFactory
3. 服务调用
    - InvokerInvocationHandler
        - 调用JavassistProxyFactory的代理处理器的invoke方法，然后调用MockClusterInvoker的invoke方法，再调用AbstractClusterInvoker的invoke方法加入负载均衡策略，再调用FailoverClusterInvoker的doInvoke方法加入集群容错策略，然后调用AbstractInvoker的invoke方法，最后调用DubboInvoker的doInvoke方法，通过ExchangeClient调用NettyClient进行netty服务调用

Dubbo的总体架构，如图所示：



Dubbo框架设计一共划分了10个层，而最上面的Service层是留给实际想要使用Dubbo开发分布式服务的开发者实现业务逻辑的接口层。图中左边淡蓝背景的为服务消费方使用的接口，右边淡绿色背

景的为服务提供方使用的接口， 位于中轴线上的为双方都用到的接口。
下面，结合Dubbo官方文档，我们分别理解一下框架分层架构中，各个层次的设计要点：

1. 服务接口层（Service）：该层是与实际业务逻辑相关的，根据服务提供方和服务消费方的业务设计对应的接口和实现。
2. 配置层（Config）：对外配置接口，以ServiceConfig和ReferenceConfig为中心，可以直接new配置类，也可以通过spring解析配置生成配置类。
3. 服务代理层（Proxy）：服务接口透明代理，生成服务的客户端Stub和服务器端Skeleton，以ServiceProxy为中心，扩展接口为ProxyFactory。
4. 服务注册层（Registry）：封装服务地址的注册与发现，以服务URL为中心，扩展接口为RegistryFactory、Registry和RegistryService。可能没有服务注册中心，此时服务提供方直接暴露服务。
5. 集群层（Cluster）：封装多个提供者的路由及负载均衡，并桥接注册中心，以Invoker为中心，扩展接口为Cluster、Directory、Router和LoadBalance。将多个服务提供方组合为一个服务提供方，实现对服务消费方来透明，只需要与一个服务提供方进行交互。
6. 监控层（Monitor）：RPC调用次数和调用时间监控，以Statistics为中心，扩展接口为MonitorFactory、Monitor和MonitorService。
7. 远程调用层（Protocol）：封将RPC调用，以Invocation和Result为中心，扩展接口为Protocol、Invoker和Exporter。Protocol是服务域，它是Invoker暴露和引用的主功能入口，它负责Invoker的生命周期管理。Invoker是实体域，它是Dubbo的核心模型，其它模型都向它靠扰，或转换成它，它代表一个可执行体，可向它发起invoke调用，它有可能是一个本地的实现，也可能是一个远程的实现，也可能一个集群实现。
8. 信息交换层（Exchange）：封装请求响应模式，同步转异步，以Request和Response为中心，扩展接口为Exchanger、ExchangeChannel、ExchangeClient和ExchangeServer。
9. 网络传输层（Transport）：抽象mina和netty为统一接口，以Message为中心，扩展接口为Channel、Transporter、Client、Server和Codec。
10. 数据序列化层（Serialize）：可复用的一些工具，扩展接口为Serialization、 ObjectInput、ObjectOutput和ThreadPool。

集群容错模式：

- Failover Cluster
  失败自动切换，当出现失败，重试其它服务器。(缺省)
  通常用于读操作，但重试会带来更长延迟。
  可通过retries=”2”来设置重试次数(不含第一次)。正是文章刚开始说的那种情况
- Failfast Cluster
  快速失败，只发起一次调用，失败立即报错。
  通常用于非幂等性的写操作，比如新增记录。
- Failsafe Cluster
  失败安全，出现异常时，直接忽略。
  通常用于写入审计日志等操作。
- Failback Cluster
  失败自动恢复，后台记录失败请求，定时重发。
  通常用于消息通知操作。
- Forking Cluster
  并行调用多个服务器，只要一个成功即返回。
  通常用于实时性要求较高的读操作，但需要浪费更多服务资源。
  可通过forks=”2”来设置最大并行数。

- Broadcast Cluster
  广播调用所有提供者，逐个调用，任意一台报错则报错。(2.1.0开始支持)
  通常用于通知所有提供者更新缓存或日志等本地资源信息。

dubbo负载均衡策略：

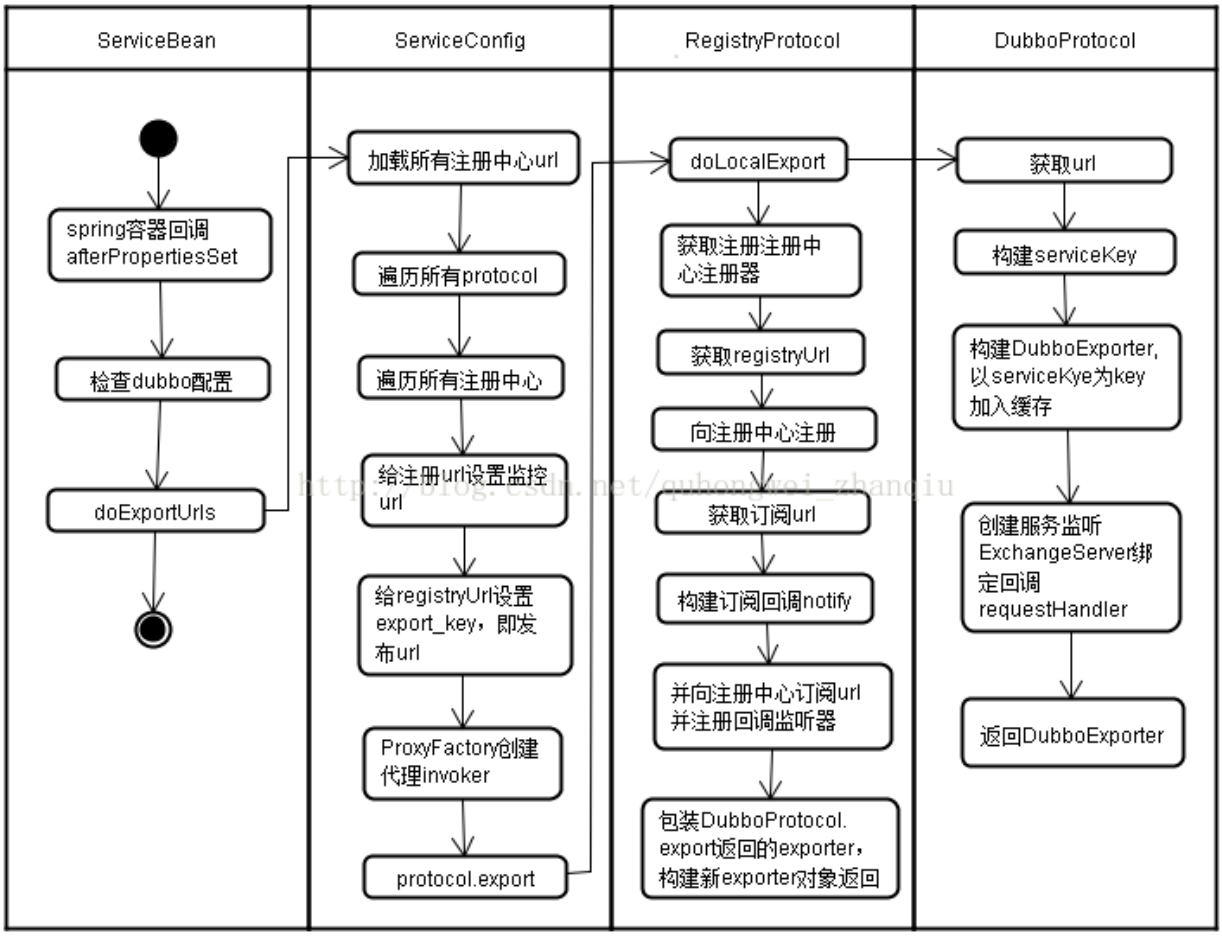
在集群负载均衡时，Dubbo提供了多种均衡策略，缺省为random随机调用。

1. RandomLoadBalance
   随机，按权重设置随机概率。
   在一个截面上碰撞的概率高，但调用量越大分布越均匀，而且按概率使用权重后也比较均匀，有利于动态调整提供者权重。
2. RoundRobin LoadBalance
   轮循，按公约后的权重设置轮循比率。
   存在慢的提供者累积请求问题，比如：第二台机器很慢，但没挂，当请求调到第二台时就卡在那，久而久之，所有请求都卡在调到第二台上。
3. LeastActive LoadBalance
   最少活跃调用数，相同活跃数的随机，活跃数指调用前后计数差。
   使慢的提供者收到更少请求，因为越慢的提供者的调用前后计数差会越大。
4. ConsistentHashLoadBalance
   一致性Hash，相同参数的请求总是发到同一提供者。
   当某一台提供者挂时，原本发往该提供者的请求，基于虚拟节点，平摊到其它提供者，不会引起剧烈变动。

源码分析

dubbo暴露服务过程

发布活动图



一般一个服务有可能即是服务端又是消费端。服务启动的时候会去向注册中心（一般是zk）暴露或者订阅自己需要的服务。我们来看下dubbo是如何把本地服务注册到注册中心的。

我们来看下ServiceBean.java这个类。

```
public class ServiceBean<T> extends ServiceConfig<T> implements InitializingBean,DisposableBean, ApplicationContextAware, ApplicationListener, BeanNameAware
```

ServiceBean实现了Spring提供的接口InitializingBean，当在完成Bean初始化和注入的时候程序会自动执行InitializingBean接口中的afterPropertiesSet方法。

在这个方法中，会对dubbo的配置做初始化赋值，即我们在配置文件中配置的：

provider，application，module，registries，monitor，protocols，path（beanName）
在方法的最后会调用父类ServiceConfig中的export方法。

```
if (! isDelay()) { // 这里配置了delay会返回false，逻辑有点怪
    export();
}
```

export方法判断是否延迟注册，延迟注册使用新的守护线程去执行暴露服务，否则直接暴露服务

```
public synchronized void export() {
```

```
        if (provider != null) {
            if (export == null) {
                export = provider.getExport();
            }
            if (delay == null) {
                delay = provider.getDelay();
            }
        }
        if (export != null && ! export.booleanValue()) { // 判断服务是否暴露
            return;
        }

    // 判断是否延迟注册，如果延迟注册则启动一个新的线程去执行，不影响主线程的运行
    if (delay != null && delay > 0) {
        Thread thread = new Thread(new Runnable() {
            public void run() {
                try {
                    Thread.sleep(delay);
                } catch (Throwable e) {
                }
                doExport();
            }
        });
        thread.setDaemon(true);
        thread.setName("DelayExportServiceThread");
        thread.start();
    } else {
        doExport();
    }
    }
```

在doExport方法中，对要暴露的服务进行了一系列的检查，检查provider，application，module，registries，monitor这些参数是否为空，是否是GenericService类型的服务，检查要注册的bean的引用和方法等。在方法的最后会调用doExportUrls方法。

```
    private void doExportUrls() {
        List<URL> registryURLs = loadRegistries(true); // 加载所有的注册中心，
    服务有可能注册在多个注册中心
        for (ProtocolConfig protocolConfig : protocols) { // 不同协议的注册。du
    bbo/hessian...
            doExportUrlsFor1Protocol(protocolConfig, registryURLs);
        }
    }
```

doExportUrlsFor1Protocol方法比较长，有些地方简写代替。

```
    private void doExportUrlsFor1Protocol(ProtocolConfig protocolConfig, List
    <URL> registryURLs) {
            String name = protocolConfig.getName();
```

```
            if (name == null || name.length() == 0) {
                name = "dubbo";
            }

        // 先从配置中取host，
        // 如果非法则通过InetAddress获取localHost，
        // 如果非法则继续通过Socket取当前localHost。
        String host = getHost();

        // 先从配置中取端口，取不到然后取默认端口，默认端口为空则取一可用端口。
        Integer port = getPort();

        // paramsMap,存放所有配置参数，下面生成url用。
        Map<String, String> map = new HashMap<String, String>();
        if (anyhost) {
            map.put(Constants.ANYHOST_KEY, "true");
        }
        map.put(Constants.SIDE_KEY, Constants.PROVIDER_SIDE);
        map.put(Constants.DUBBO_VERSION_KEY, Version.getVersion());
        map.put(Constants.TIMESTAMP_KEY, String.valueOf(System.currentTimeMil
lis()));
        if (ConfigUtils.getPid() > 0) {
            map.put(Constants.PID_KEY, String.valueOf(ConfigUtils.getPid()));
        }
        appendParameters(map, application);
        appendParameters(map, module);
        appendParameters(map, provider, Constants.DEFAULT_KEY);
        appendParameters(map, protocolConfig);
        appendParameters(map, this);

        // method子标签配置规则解析，retry次数，参数等。没有使用过，不做解释
        if (methods != null && methods.size() > 0) {
            for (MethodConfig method : methods) {
                appendParameters(map, method, method.getName());
                String retryKey = method.getName() + ".retry";
                if (map.containsKey(retryKey)) {
                    String retryValue = map.remove(retryKey);
                    if ("false".equals(retryValue)) {
                        map.put(method.getName() + ".retries", "0");
                    }
                }
                List<ArgumentConfig> arguments = method.getArguments();
                if (arguments != null && arguments.size() > 0) {
                    ..... // 设置参数
                }
            } // end of methods for
        }

        // 获取所有的methods方法
        if (generic) { // 如果是泛化实现，generic=true，method=*表示任意方法
            map.put("generic", String.valueOf(true));
            map.put("methods", Constants.ANY_VALUE);
```

```
        } else {
            String revision = Version.getVersion(interfaceClass, version);
            if (revision != null && revision.length() > 0) {
                map.put("revision", revision);
            }

            String[] methods = Wrapper.getWrapper(interfaceClass).getMethodNa
mes();
            if(methods.length == 0) {
                logger.warn("NO method found in service interface " + interfa
ceClass.getName());
                map.put("methods", Constants.ANY_VALUE);
            }
            else {
                map.put("methods", StringUtils.join(new HashSet<String>(Array
s.asList(methods)), ","));
            }
        }

        // 有需要配置token，默认随机UUID，否则使用配置中的token，作令牌验证用
        if (! ConfigUtils.isEmpty(token)) {
            if (ConfigUtils.isDefault(token)) {
                map.put("token", UUID.randomUUID().toString());
            } else {
                map.put("token", token);
            }
        }

        // injvm不需要暴露服务，标注notify=false
        if ("injvm".equals(protocolConfig.getName())) {
            protocolConfig.setRegister(false);
            map.put("notify", "false");
        }

        // 导出服务
        String contextPath = protocolConfig.getContextpath();
        if ((contextPath == null || contextPath.length() == 0) && provider !=
null) {
            contextPath = provider.getContextpath();
        }

        // 根据参数创建url对象
        URL url = new URL(name, host, port,
            (contextPath == null || contextPath.length() == 0 ? "" : contextP
ath + "/") + path, map);

        // 如果url使用的协议存在扩展，调用对应的扩展来修改原url。目前扩展有override，absent
        if (ExtensionLoader.getExtensionLoader(ConfiguratorFactory.class)
                .hasExtension(url.getProtocol())) {
            url = ExtensionLoader.getExtensionLoader(ConfiguratorFactory.clas
s)
```

```
                .getExtension(url.getProtocol()).getConfigurator(url).con
figure(url);
    }

    String scope = url.getParameter(Constants.SCOPE_KEY);
    //配置为none不暴露
    if (! Constants.SCOPE_NONE.toString().equalsIgnoreCase(scope)) {

        //配置不是remote的情况下做本地暴露 (配置为remote，则表示只暴露远程服
务)
        if (!Constants.SCOPE_REMOTE.toString().equalsIgnoreCase(scope)) {
            // 只有当url协议不是injvm的时候才会在本地暴露服务，如果是injvm则什
么都不做
            exportLocal(url);
        }
        //如果配置不是local则暴露为远程服务.(配置为local，则表示只暴露本地服
务)
        if (! Constants.SCOPE_LOCAL.toString().equalsIgnoreCase(scope) ){
            if (logger.isInfoEnabled()) {
                logger.info("Export dubbo service " + interfaceClass.getN
ame() + " to url " + url);
            }
            if (registryURLs != null && registryURLs.size() > 0
                    && url.getParameter("register", true)) {
                for (URL registryURL : registryURLs) {
                    url = url.addParameterIfAbsent("dynamic", registryUR
L.getParameter("dynamic"));
                    URL monitorUrl = loadMonitor(registryURL);
                    // 如果有monitor信息，则在url上增加monitor配置
                    if (monitorUrl != null) {
                        url = url.addParameterAndEncoded(Constants.MONITO
R_KEY, monitorUrl.toFullString());
                    }

                    // 通过代理工厂将url转化成invoker对象，proxyFactory的实
现是JavassistProxyFactory
                    Invoker<?> invoker = proxyFactory.getInvoker(ref, (Cl
ass) interfaceClass,
                        registryURL.addParameterAndEncoded(Constants.EXPO
RT_KEY, url.toFullString()));

                    // 这里invoker对象协议是registry，protocol根据协议找到Re
gisterProtocol实现类，注1。
                    // 在调用RegisterProtocol类的export方法之前会先调用Prot
ocolListenerWrapper类的export方法
                    // protocol实例转化为ProtocolFilterWrapper，包了一层Reg
istryProtocol
                    Exporter<?> exporter = protocol.export(invoker);
                    exporters.add(exporter);
                }
            } else {
```

```
                Invoker<?> invoker = proxyFactory.getInvoker(ref, (Class)
interfaceClass, url);

                Exporter<?> exporter = protocol.export(invoker);
                exporters.add(exporter);
            }
        }
    }
    this.urls.add(url);
}
```

注1：protocol对象是如何知道在执行的时候是使用的哪个实现类？如何根据url的协议来创建对应的Protocol。 在ServiceConfig初始化的时候，protocol初始化为

```
Protocol protocol = ExtensionLoader.getExtensionLoader(Protocol.class).ge
tAdaptiveExtension();
```

getAdaptiveExtension方法说明是在调用的时候再决定使用什么扩展。

下面为生成的protocol字节码Protocol$Adpative image

最终的注册还是在RegistryProtocol类中的export方法中完成的

```
public <T> Exporter<T> export(final Invoker<T> originInvoker) throws RpcE
xception {
    //export invoker
    // 在本地暴露的时候会开启Netty服务
    final ExporterChangeableWrapper<T> exporter = doLocalExport(originInv
oker);
}
```

我们先来看下本地暴露服务。在本地暴露时，protocol.export(invokerDelegete)会转化为DubboProtocol

```
private <T> ExporterChangeableWrapper<T>  doLocalExport(final Invoker<T>
 originInvoker){
    String key = getCacheKey(originInvoker);
    ExporterChangeableWrapper<T> exporter = (ExporterChangeableWrapper<T
>) bounds.get(key);
    if (exporter == null) {
        synchronized (bounds) {
            exporter = (ExporterChangeableWrapper<T>) bounds.get(key);
            if (exporter == null) {
                final Invoker<?> invokerDelegete = new InvokerDelegete<T>
(originInvoker,
                    getProviderUrl(originInvoker));
                exporter = new ExporterChangeableWrapper<T>((Exporter<T>)
                    protocol.export(invokerDelegete), originInvoker);
```

```
                    bounds.put(key, exporter);
                }
            }
        }
        return (ExporterChangeableWrapper<T>) exporter;
    }
```

因为不是Registry协议，在ProtocolFilterWrapper执行的时候就会走下面的逻辑，在这里有一个方法buildInvokerChain，会创建Dubbo调用过程中的过滤器链，具体类似tomcat中的过滤器链规则。

```
    public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
        if (Constants.REGISTRY_PROTOCOL.equals(invoker.getUrl().getProtocol
    ())) {
            return protocol.export(invoker);
        }
        return protocol.export(buildInvokerChain(invoker, Constants.SERVICE_F
    ILTER_KEY, Constants.PROVIDER));
    }
```

DubboProtocol中的export方法会缓存生成的exporter对象，然后方法最后开启Netty服务

```
    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, exporterMa
    p);
    exporterMap.put(key, exporter);

    // 这里会判断是否是stub服务（不是太懂，默认是false的）
    //export an stub service for dispaching event
    Boolean isStubSupportEvent = url.getParameter(Constants.STUB_EVENT_KEY,Co
    nstants.DEFAULT_STUB_EVENT);

    // 判断是否是callback服务
    Boolean isCallbackservice = url.getParameter(Constants.IS_CALLBACK_SERVIC
    E, false);
    if (isStubSupportEvent && !isCallbackservice){
        String stubServiceMethods = url.getParameter(Constants.STUB_EVENT_MET
    HODS_KEY);
        if (stubServiceMethods == null || stubServiceMethods.length() == 0 ){
            // log
        } else {
            stubServiceMethodsMap.put(url.getServiceKey(), stubServiceMethod
    s);
        }
    }
```

```
    // 开启Netty服务
    openServer(url);

    return exporter;


}
```

openServer方法会根据key（ip+port）判断server是否存在，不存在则调用createServer(URL url)方法创建一个链接。

```
    private ExchangeServer createServer(URL url) {
        //默认开启heartbeat
        url = url.addParameterIfAbsent(Constants.HEARTBEAT_KEY, String.valueO
    f(Constants.DEFAULT_HEARTBEAT));
        ...
        ExchangeServer server = Exchangers.bind(url, requestHandler);
        ...
        return server;
    }
```

Exchangers的bind方法getExchanger(url)得到的默认为HeaderExchanger，HeaderExchanger的bind方法又调用了Transporters中的bind方法，看getTransporter方法又碰到了熟悉的适配加载机制，Transporter类的注解为@SPI(“netty”)，默认会调用NettyTransporter的bind方法，NettyTransporter的bind方法直接new了一个NettyServer，NettyServer初始化的时候调用父类的构造方法，父类的构造方法中调用了NettyServer的doOpen方法，NettyServer的链接创建在doOpen方法中完成。

```
    Exchangers
    public static ExchangeServer bind(URL url, ExchangeHandler handler) throw
    s RemotingException {
        // 断点调到这里codec为dubbo，应该是序列化用。
        url = url.addParameterIfAbsent(Constants.CODEC_KEY, "exchange");
        return getExchanger(url).bind(url, handler);
    }
    HeaderExchanger
    public ExchangeServer bind(URL url, ExchangeHandler handler) throws Remot
    ingException {
        return new HeaderExchangeServer(Transporters.bind(url,
            new DecodeHandler(new HeaderExchangeHandler(handler))));
        // 这里应当是适配者模式
    }
    Transporters
    public static Server bind(URL url, ChannelHandler... handlers) throws Rem
    otingException {
        return getTransporter().bind(url, handler);
    }
    public static Transporter getTransporter() {
        // 适配扩展点加载
        return ExtensionLoader.getExtensionLoader(Transporter.class).getAdapt
    iveExtension();
```

```
    }
```

在NettyServer的doOpen方法中可以看到这里绑定了netty的端口和链接。方法到这里本地的Server端口都已经暴露完毕。

```
    protected void doOpen() throws Throwable {
        NettyHelper.setNettyLoggerFactory();
        ExecutorService boss = Executors.newCachedThreadPool(new NamedThreadF
    actory("NettyServerBoss", true));
        ExecutorService worker = Executors.newCachedThreadPool(new NamedThrea
    dFactory("NettyServerWorker", true));
        ChannelFactory channelFactory = new NioServerSocketChannelFactory(bos
    s, worker,
                getUrl().getPositiveParameter(Constants.IO_THREADS_KEY, Constant
    s.DEFAULT_IO_THREADS));
        bootstrap = new ServerBootstrap(channelFactory);

    final NettyHandler nettyHandler = new NettyHandler(getUrl(), this);
    channels = nettyHandler.getChannels();
    // https://issues.jboss.org/browse/NETTY-365
    // https://issues.jboss.org/browse/NETTY-379
    // final Timer timer = new HashedWheelTimer(new NamedThreadFactory("Netty
    IdleTimer", true));
    bootstrap.setPipelineFactory(new ChannelPipelineFactory() {
        public ChannelPipeline getPipeline() {
            NettyCodecAdapter adapter = new NettyCodecAdapter(getCodec() ,get
    Url(), NettyServer.this);
            ChannelPipeline pipeline = Channels.pipeline();
            /*int idleTimeout = getIdleTimeout();
            if (idleTimeout > 10000) {
                pipeline.addLast("timer", new IdleStateHandler(timer, idleTim
    eout / 1000, 0, 0));
            }*/
            pipeline.addLast("decoder", adapter.getDecoder());
            pipeline.addLast("encoder", adapter.getEncoder());
            pipeline.addLast("handler", nettyHandler);
            return pipeline;
        }
    });
    // bind
    channel = bootstrap.bind(getBindAddress());
```

}

回到RegistryProtocol中，来看下在zk中暴露服务。首先先获取zk的配置信息，然后获取需要暴露的url，然后调用registry.register方法将url注册到zookeeper上去。

```
    public <T> Exporter<T> export(final Invoker<T> originInvoker) throws RpcE
    xception {
        //export invoker
```

```
    // 在本地暴露服务
    final ExporterChangeableWrapper<T> exporter = doLocalExport(originInv
oker);
    //registry provider
    // 拿到zookeeper的注册信息
    final Registry registry = getRegistry(originInvoker);
    // 获取需要暴露provider的url对象，dubbo的注册订阅通信都是以url作为参数传递
的
    final URL registedProviderUrl = getRegistedProviderUrl(originInvoke
r);
    registry.register(registedProviderUrl); // 注册服务
    // 订阅override数据
    // FIXME 提供者订阅时，会影响同一JVM即暴露服务，又引用同一服务的的场景，
    // 因为subscribed以服务名为缓存的key，导致订阅信息覆盖。
    final URL overrideSubscribeUrl = getSubscribedOverrideUrl(registedPro
viderUrl);
    final OverrideListener overrideSubscribeListener = new OverrideListen
er(overrideSubscribeUrl);
    overrideListeners.put(overrideSubscribeUrl, overrideSubscribeListene
r);

// 暴露的同时订阅服务，另外会在zk上创建configurators节点信息
registry.subscribe(overrideSubscribeUrl, overrideSubscribeListener);
//保证每次export都返回一个新的exporter实例
return new Exporter<T>() {
    ...
};
}
```

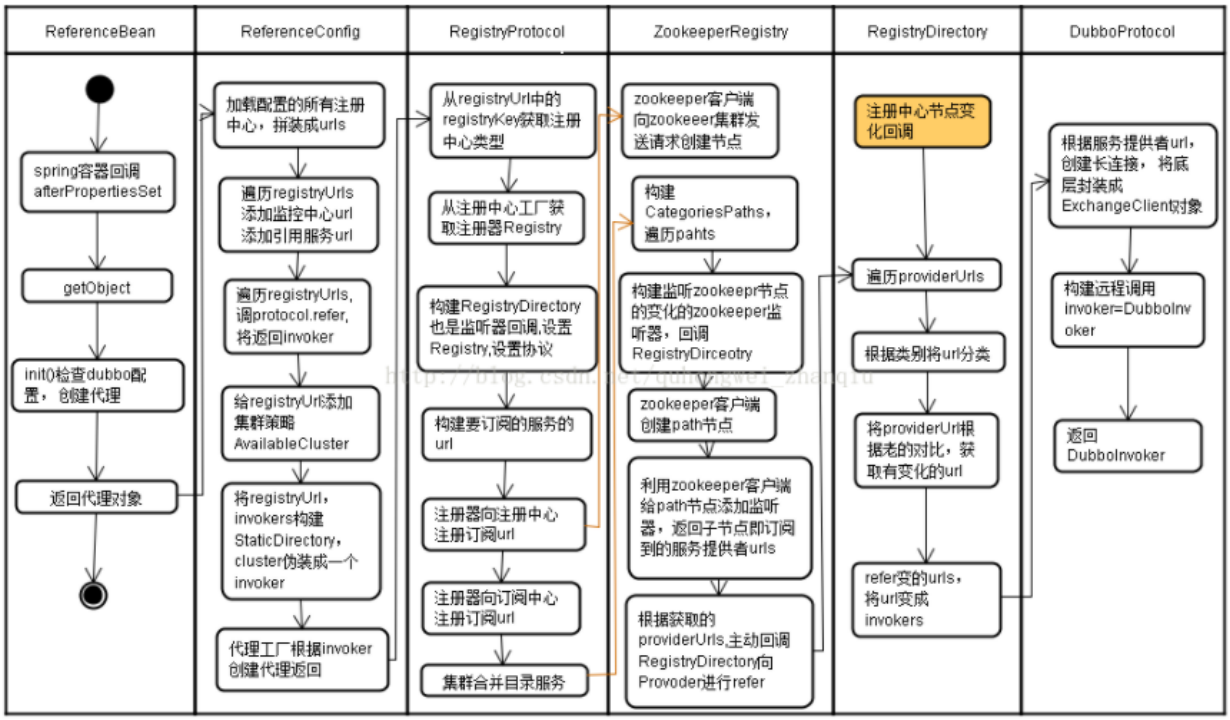
调试代码可以发现，这里的registry对象为ZookeeperRegistry，ZookeeperRegistry继承了FailbackRegistry，默认调用父类的register方法，然后调用ZookeeperRegistry的doRegister方法，这里就比较简单了。直接调用zkClient在zookeeper上创建一个节点信息，这样就把服务丢到zk上去了

```
protected void doRegister(URL url) {
    zkClient.create(toUrlPath(url), url.getParameter(Constants.DYNAMIC_KE
Y, true));
}
```

暴露完服务之后，还会调用registry的subscribe方法，这里主要是加了注册订阅Listener，在创建出其他节点之后会调用notify方法。notify方法会做两件事，1. 会将url改动更新到缓存的配置文件中。2. 会通知listener变动，此通知为全量通知。

dubbo服务引用过程

发布服务活动图：



ReferenceBean实现了FactoryBean的getObject方法

```
public Object getObject() throws Exception {
        return get();
    }
```

ReferenceConfig实现get（）方法

```
public synchronized T get() {
        if (destroyed){
            throw new IllegalStateException("Already destroyed!");
        }
        if (ref == null) {
            init();
        }
        return ref;
    }
```

init（）方法

```
 private void init() {
     ……
      ref = createProxy(map);
  }
```

createProxy 方法

```
private T createProxy(Map<String, String> map) {
```

```
    ……

} else {
    // 通过注册中心配置拼装URL
    List<URL> us = loadRegistries(false);
    if (us != null && us.size() > 0) {
        for (URL u : us) {
            URL monitorUrl = loadMonitor(u);
               if (monitorUrl != null) {
                   map.put(Constants.MONITOR_KEY, URL.encode(monitorUrl.t
oFullString()));
               }
            urls.add(u.addParameterAndEncoded(Constants.REFER_KEY, String
Utils.toQueryString(map)));
          }
    }
    if (urls.size() == 1) {
        //通过协议 注册与订阅
         invoker = refprotocol.refer(interfaceClass, urls.get(0));
     } else {
         List<Invoker<?>> invokers = new ArrayList<Invoker<?>>();
         URL registryURL = null;
         for (URL url : urls) {
             invokers.add(refprotocol.refer(interfaceClass, url));
             if (Constants.REGISTRY_PROTOCOL.equals(url.getProtocol())) {
                 registryURL = url; // 用了最后一个registry url
             }
         }
         if (registryURL != null) { // 有 注册中心协议的URL
             // 对有注册中心的Cluster 只用 AvailableCluster
             URL u = registryURL.addParameter(Constants.CLUSTER_KEY, Avai
lableCluster.NAME);
             invoker = cluster.join(new StaticDirectory(u, invokers));
         }  else { // 不是 注册中心的URL
             invoker = cluster.join(new StaticDirectory(invokers));
         }
     }
 }
……
 // 创建服务代理
 return (T) proxyFactory.getProxy(invoker);
   }
```

RegistryProtocol 的refer方法

```
public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {
        ……
        return doRefer(cluster, registry, type, url);
    }
```

doRefer方法

```
private <T> Invoker<T> doRefer(Cluster cluster, Registry registry, Class<
T> type, URL url) {
    RegistryDirectory<T> directory = new RegistryDirectory<T>(type, url);
        directory.setRegistry(registry);
        directory.setProtocol(protocol);
        URL subscribeUrl = new URL(Constants.CONSUMER_PROTOCOL, NetUtils.
getLocalHost(), 0, type.getName(), directory.getUrl().getParameters());
        if (! Constants.ANY_VALUE.equals(url.getServiceInterface())
                && url.getParameter(Constants.REGISTER_KEY, true)) {
                //FailbackRegistry 的 register 方法
                registry.register(subscribeUrl.addParameters(Constants.CA
TEGORY_KEY, Constants.CONSUMERS_CATEGORY,
                    Constants.CHECK_KEY, String.valueOf(false)));
        }
        //-->FailbackRegistry.subscribe() ---> ZookeeperRegistry.doSubscr
ibe()
        directory.subscribe(subscribeUrl.addParameter(Constants.CATEGORY_
KEY,
                Constants.PROVIDERS_CATEGORY
                + "," + Constants.CONFIGURATORS_CATEGORY
                + "," + Constants.ROUTERS_CATEGORY));
        return cluster.join(directory);
    }
```

FailbackRegistry的registry方法

```
public void register(URL url) {
        super.register(url);
       ……
        try {
            // 向服务器端发送注册请求
            doRegister(url);
        } catch (Exception e) {
           ……
        }
    }
```

ZookeeperRegistry的doRegister方法

```
protected void doRegister(URL url) {
        try {
            zkClient.create(toUrlPath(url), url.getParameter(Constants.DY
NAMIC_KEY, true));
        } catch (Throwable e) {
            throw new RpcException("Failed to register " + url + " to zoo
keeper " + getUrl() + ", cause: " + e.getMessage(), e);
        }
```

```
    }
```

StaticDirectory中加入路由信息，父类的构造方法中加入

```
public AbstractDirectory(URL url, URL consumerUrl, List<Router> routers)
{
        if (url == null)
            throw new IllegalArgumentException("url == null");
        this.url = url;
        this.consumerUrl = consumerUrl;
        setRouters(routers);
    }

protected void setRouters(List<Router> routers){
        // copy list
        routers = routers == null ? new  ArrayList<Router>() : new ArrayL
ist<Router>(routers);
        // append url router
        String routerkey = url.getParameter(Constants.ROUTER_KEY);
        if (routerkey != null && routerkey.length() > 0) {
            RouterFactory routerFactory = ExtensionLoader.getExtensionLoa
der(RouterFactory.class).getExtension(routerkey);
            routers.add(routerFactory.getRouter(url));
        }
        // append mock invoker selector
        routers.add(new MockInvokersSelector());
        Collections.sort(routers);
        this.routers = routers;
    }
```

默认集群 FailoverCluster

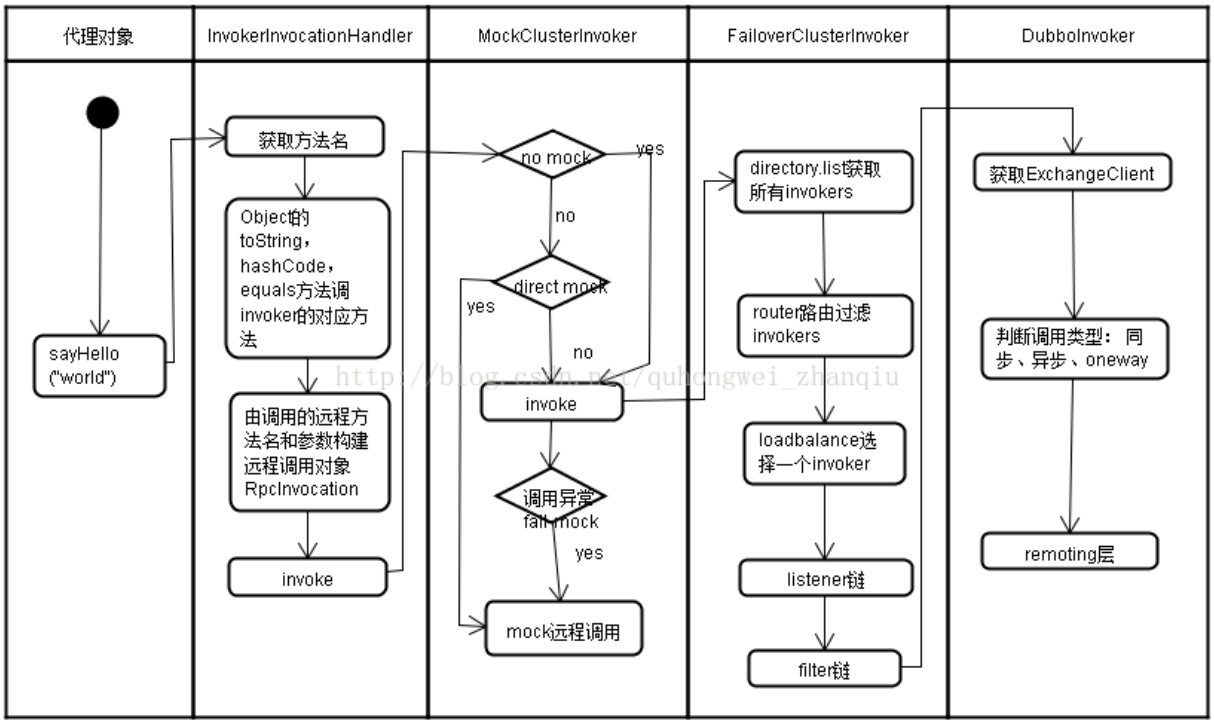
```
public class FailoverCluster implements Cluster {

    public final static String NAME = "failover";

    public <T> Invoker<T> join(Directory<T> directory) throws RpcExceptio
n {
        return new FailoverClusterInvoker<T>(directory);
    }

}
```

服务调用过程



代理工厂由StubProxyFactoryWrapper包装了JavassistProxyFactory，最终由JavassistProxyFactory生成代理对象

```
JavassistProxyFactory
public <T> T getProxy(Invoker<T> invoker, Class<?>[] interfaces) {
        return (T) Proxy.getProxy(interfaces).newInstance(new InvokerInvo
cationHandler(invoker));
    }
```

1. 调用用户接口方法时调用InvokerInvocationHandler的invoke方法

   public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
   String methodName = method.getName();
   Class<?>[] parameterTypes = method.getParameterTypes();
   ……
   //这里的invoker即为MockClusterInvoker
   return invoker.invoke(new RpcInvocation(method, args)).recreate();
   }
2. MockClusterInvoker根据参数提供了三种调用策略
   - 不需要mock， 直接调用FailoverClusterInvoker，默认调用策略
   - 强制mock，调用mock
   - 先调FailoverClusterInvoker，调用失败在mock
3. AbstractClusterInvoker的invoke方法
   通过目录服务查找到所有订阅的服务提供者的Invoker对象
   路由服务根据策略来过滤选择调用的Invokers
   通过负载均衡策略LoadBalance来选择一个Invoker

   public Result invoke(final Invocation invocation) throws RpcException {

   ```
       checkWheatherDestoried();
   ```

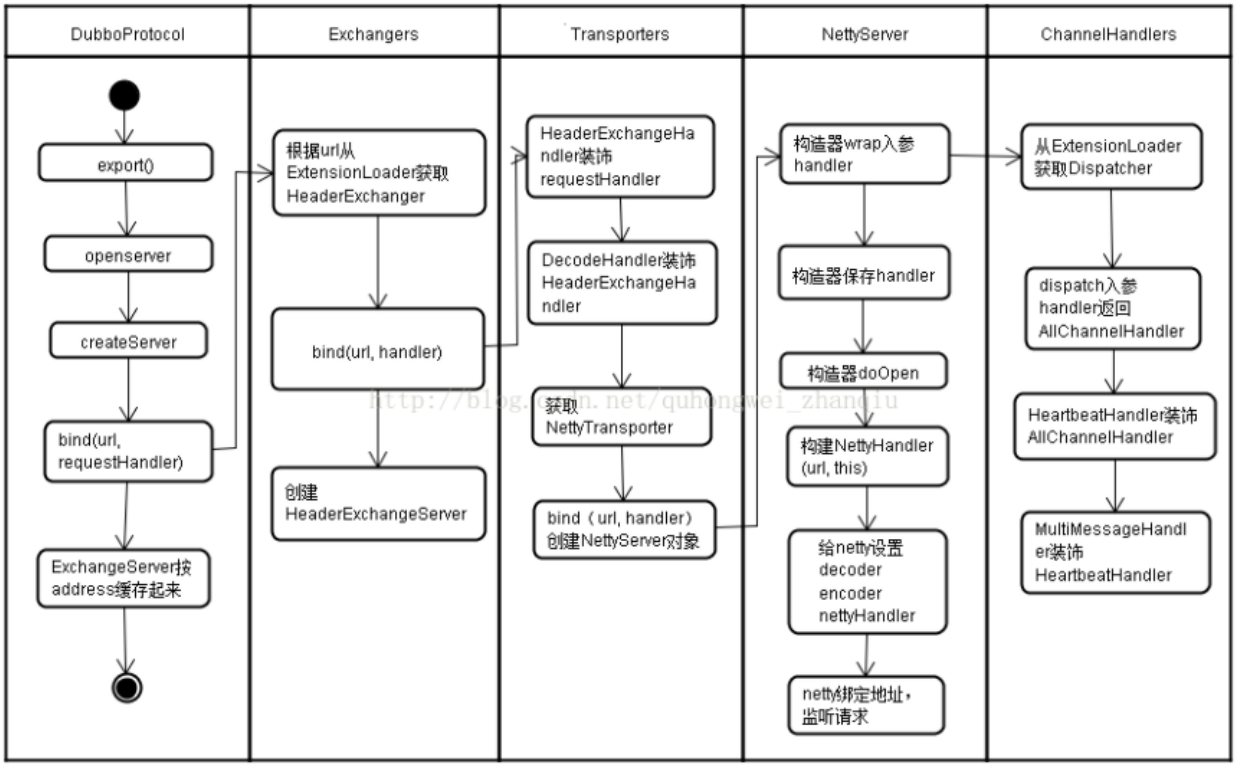
```
        LoadBalance loadbalance;

        List<Invoker<T>> invokers = list(invocation);
        if (invokers != null && invokers.size() > 0) {
            loadbalance = ExtensionLoader.getExtensionLoader(LoadBalance.clas
        s).getExtension(invokers.get(0).getUrl()
                    .getMethodParameter(invocation.getMethodName(),Constants.
        LOADBALANCE_KEY, Constants.DEFAULT_LOADBALANCE));
        } else {
            loadbalance = ExtensionLoader.getExtensionLoader(LoadBalance.clas
        s).getExtension(Constants.DEFAULT_LOADBALANCE);
        }
        RpcUtils.attachInvocationIdIfAsync(getUrl(), invocation);
        return doInvoke(invocation, invokers, loadbalance);


    }
```

4. 执行选择的Invoker.inoker(invocation)
   经过监听器链，默认没有
   经过过滤器链，内置实现了很多
   执行到远程调用的DubboInvoker
5. DubboInvoker
   根据url 也就是根据服务提供者的长连接，这里封装成交互层对象ExchangeClient供这里调用
   判断远程调用类型同步，异步还是oneway模式，ExchangeClient发起远程调用，底层remoting不在这里描述了
   获取调用结果：
   - Oneway返回空RpcResult
   - 异步，直接返回空RpcResult, ResponseFuture回调
   - 同步， ResponseFuture模式同步转异步，等待响应返回

Dubbo通信层之暴露服务



Dubbo通信层之引用服务