Dubbo提供者暴露流程分析,2.7.4.1版本

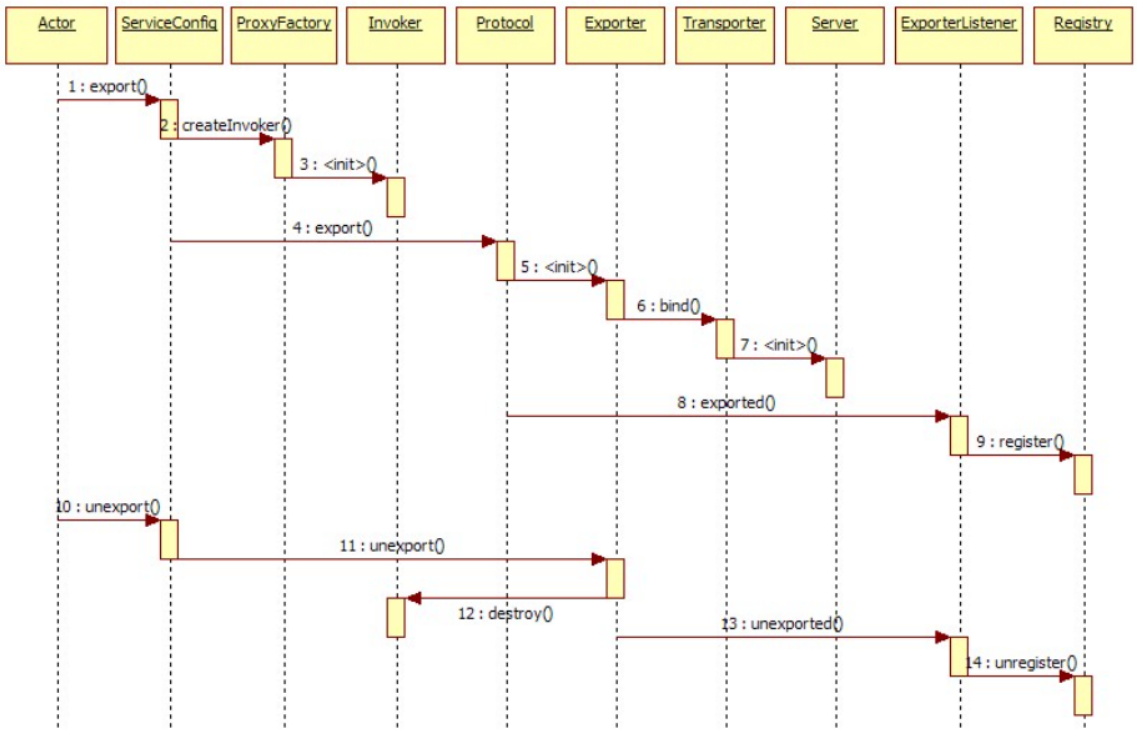
1、各层说明

- **Config 配置层**：对外配置接口，以 `ServiceConfig`，`ReferenceConfig` 为中心，可以直接初始化配置类，也可以通过 spring 解析配置生成配置类
- **Proxy 服务代理层**：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`
- **Registry 注册中心层**：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`，`RegistryService`
- **Cluster 路由层**：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`，`LoadBalance`
- **Monitor 监控层**：RPC 调用次数和调用时间监控，以 `Statistics` 为中心，扩展接口为 `MonitorFactory`，`Monitor`，`MonitorService`
- **Protocol 远程调用层**：封装 RPC 调用，以 `Invocation`，`Result` 为中心，扩展接口为 `Protocol`，`Invoker`，`Exporter`
- **Exchange 信息交换层**：封装请求响应模式，同步转异步，以 `Request`，`Response` 为中心，扩展接口为 `Exchanger`，`ExchangeChannel`，`ExchangeClient`，`ExchangeServer`
- **Transport 网络传输层**：抽象 mina 和 netty 为统一接口，以 `Message` 为中心，扩展接口为 `Channel`，`Transporter`，`Client`，`Server`，`Codec`
- **Serialize 数据序列化层**：可复用的一些工具，扩展接口为 `Serialization`，`ObjectInput`，`ObjectOutput`，`ThreadPool`

2、官网的时序图

暴露服务时序

展开总设计图右边服务提供方暴露服务的蓝色初始化链，时序图如下：

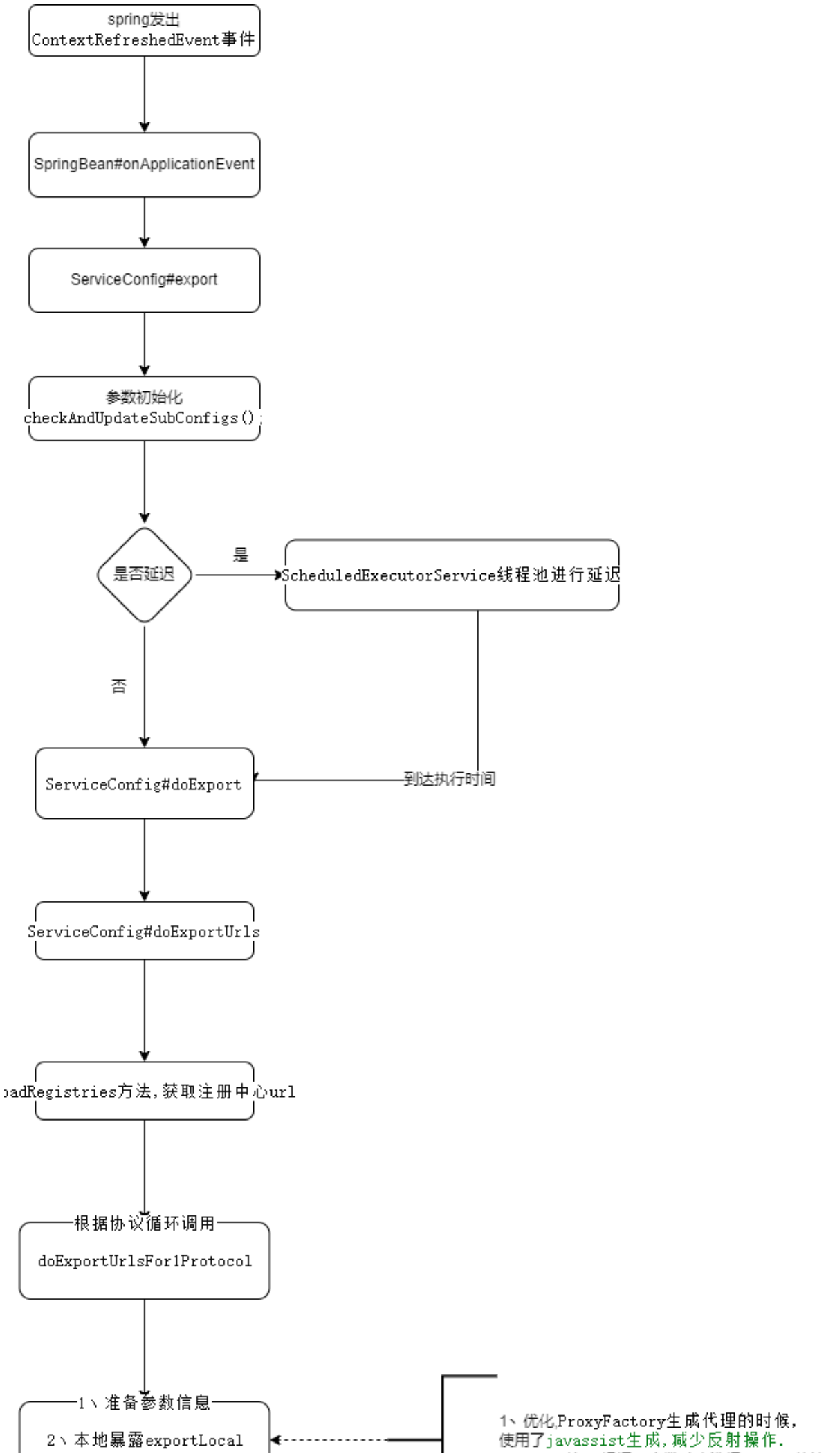


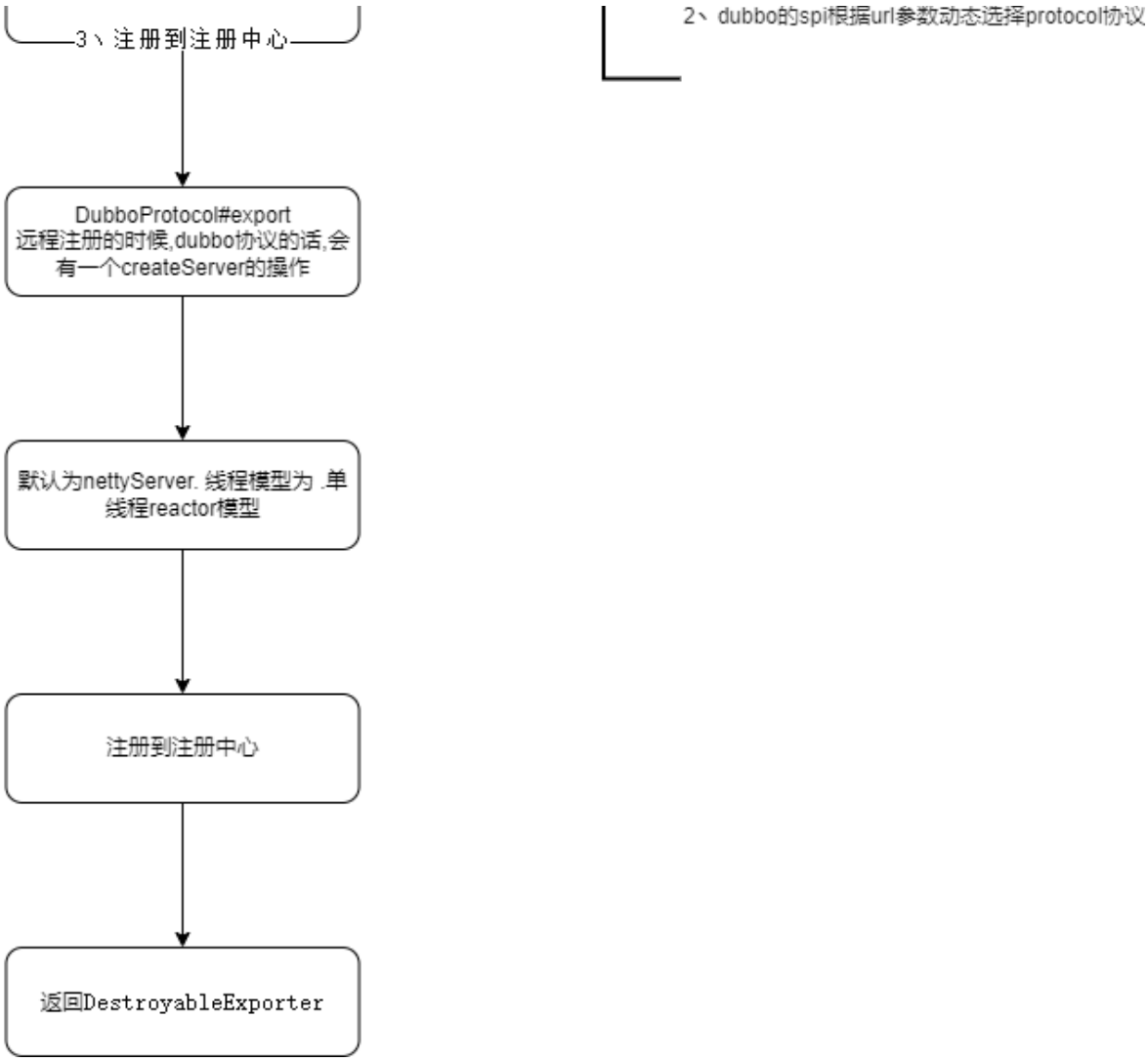
3、领域模型

在 Dubbo 的核心领域模型中：

- 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，其它模型都向它靠拢，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。
- Invocation 是会话域，它持有调用过程中的变量，比如方法名，参数等。

4、流程图


spring发出
ContextRefreshedEvent事件
SpringBean#onApplicationEvent
ServiceConfig#export
参数初始化
checkAndUpdateSubConfigs()
是否延迟
是
ScheduledExecutorService线程池进行延迟
否
到达执行时间
ServiceConfig#doExport
ServiceConfig#doExportUrls
adRegistries方法,获取注册中心url
根据协议循环调用
doExportUrlsFor1Protocol
1、准备参数信息
2、本地暴露exportLocal
1、优化ProxyFactory生成代理的时候,
使用了javassist生成,减少反射操作.



5、代码分析

1、spring启动流程,在结束刷新容器时,发送了上下文刷新事件

```
org.springframework.context.support.AbstractApplicationContext#finishRefresh
protected void finishRefresh() {
   // Clear context-level resource caches (such as ASM metadata from scanning).
   clearResourceCaches();

   // Initialize lifecycle processor for this context.
   initLifecycleProcessor();

   // Propagate refresh to lifecycle processor first.
   getLifecycleProcessor().onRefresh();

   // Publish the final event.
   publishEvent(new ContextRefreshedEvent(this));

   // Participate in LiveBeansView MBean, if active.
   LiveBeansView.registerApplicationContext(this);
}
```

2、提供者Servicebean监听了ContextRefreshedEvent事件

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

        @Override
        public void onApplicationEvent(ContextRefreshedEvent event) {
            //是否注册过了
            if (!isExported() && !isUnexported()) {
                if (logger.isInfoEnabled()) {
                logger.info("The service ready on spring started. service: " + getInterface());
                }
              export();
            }
        }

        @Override
        public void export() {
            super.export();
            // 这个可以忽略, 刷新了提供者在同一个项目被引用时,进行了刷新.
            publishExportEvent();
        }

       public synchronized void export() {
           //检查和更新一些参数.
        checkAndUpdateSubConfigs();

        if (!shouldExport()) {
            return;
        }
        //懒加载,就等到时间执行doExport();
        if (shouldDelay()) {
            DELAY_EXPORT_EXECUTOR.schedule(this::doExport, getDelay(), TimeUnit.MILLISECONDS);
        } else {
            doExport();
        }
    }
    protected synchronized void doExport() {
        if (unexported) {
            throw new IllegalStateException("The service " + interfaceClass.getName() + " has already unexported!");
        }
        if (exported) {
            return;
        }
        exported = true;

        if (StringUtils.isEmpty(path)) {
            path = interfaceName;
        }
        doExportUrls();
    }
```

```

}
```

3、核心逻辑

```
private void doExportUrls() {
    //这个方法根据注册协议拼接了注册的url.注册方式,地址等等参数
    //如:registry://10.200.6.209:32181/org.apache.dubbo.registry.RegistryService?application=etbc-system&dubbo=2.0.2&organization=cc.eslink&owner=xing.lu.si&pid=19708&qos.enable=false&registry=zookeeper&release=2.7.4.1&timestamp=1662444973606
    List<URL> registryURLs = loadRegistries(true);
    for (ProtocolConfig protocolConfig : protocols) {
        String pathKey = URL.buildKey(getContextPath(protocolConfig).map(p -> p + "/" + path).orElse(path), group, version);
        ProviderModel providerModel = new ProviderModel(pathKey, ref, interfaceClass);
        //存入提供者缓存, 目前看源码只是为了 rest协议时 使用. dubbo协议(默认)使用netty交互,没有用到
        ApplicationModel.initProviderModel(pathKey, providerModel);
        doExportUrlsFor1Protocol(protocolConfig, registryURLs);
    }
}
```

```
private void doExportUrlsFor1Protocol(ProtocolConfig protocolConfig, List<URL> registryURLs) {
    //    …… 省略很长的参数拼接,拼接后的参数如下.
}
```

```
map = {HashMap@14879} size = 25
  "release" -> "2.7.4.1"
  "methods" -> "noticeTypes,publishWenZhang,publishNotice,queryNoticeInfo,publishPTNotice"
  "deprecated" -> "false"
  "dubbo" -> "2.0.2"
  "loadbalance" -> "roundrobin"
  "pid" -> "4108"
  "interface" -> "cc.eslink.etbc.center.service.INoticeService"
  "threadpool" -> "cached"
  "qos.enable" -> "false"
  "timeout" -> "10000"
  "bind.port" -> "18035"
  "dynamic" -> "true"
  "dispatcher" -> "message"
  "timestamp" -> "1662446282046"
  "anyhost" -> "true"
  "owner" -> "xing.lu.si"
  "side" -> "provider"
  "threads" -> "1000"
  "generic" -> "false"
  "retries" -> "0"
  "delay" -> "-1"
  "bind.ip" -> "10.30.2.162"
  "application" -> "etbc-system"
  "organization" -> "cc.eslink"
  "bean.name" -> "ServiceBean:cc.eslink.etbc.center.service.INoticeService"
```

```java
private void doExportUrlsFor1Protocol(ProtocolConfig protocolConfig, List<URL> registryURLs) {
    // 参数如上. 根据参数拼接出url

 //dubbo://10.30.2.162:18035/cc.eslink.etbc.center.service.INoticeService?anyhost=true&application=etbc-system&bean.name=ServiceBean:cc.eslink.etbc.center.service.INoticeService&bind.ip=10.30.2.162&bind.port=18035&delay=-1&deprecated=false&dispatcher=message&dubbo=2.0.2&dynamic=true&generic=false&interface=cc.eslink.etbc.center.service.INoticeService&loadbalance=roundrobin&methods=noticeTypes,publishWenZhang,publishNotice,queryNoticeInfo,publishPTNotice&organization=cc.eslink&owner=xing.lu.si&pid=4108&qos.enable=false&release=2.7.4.1&retries=0&side=provider&threadpool=cached&threads=1000&timeout=10000&timestamp=1662446282046
     URL url = new URL(name, host, port, getContextPath(protocolConfig).map(p -> p + "/" + path).orElse(path), map);

    //注册本地. 这里如果同服务调用 走本地的策略.
    if (!SCOPE_REMOTE.equalsIgnoreCase(scope)) {
        //方法如下
                exportLocal(url);
       }


}
```

```java
private void exportLocal(URL url) {
    //重新赋值了协议为本地, host和port
    URL local = URLBuilder.from(url)
```

```
            .setProtocol(LOCAL_PROTOCOL)
            .setHost(LOCALHOST_VALUE)
            .setPort(0)
            .build();
    //这里用的dubbospi的特性, 根据url的protocol值来选择合适的实现类. 这里因为LOCAL_PROTOCOL=injvm,则走org.apache.dubbo.rpc.protocol.injvm.InjvmProtocol#export
    Exporter<?> exporter = protocol.export(
            PROXY_FACTORY.getInvoker(ref, (Class) interfaceClass, local));
    exporters.add(exporter);
    logger.info("Export dubbo service " + interfaceClass.getName() + " to local registry url : " + local);
}

//实际就是,放在ConcurrentHashMap  exporterMap中.  在消费者引用的时候,会从exporterMap中拿
InjvmExporter(Invoker<T> invoker, String key, Map<String, Exporter<?>> exporterMap) {
        super(invoker);
        this.key = key;
        this.exporterMap = exporterMap;
        exporterMap.put(key, this);
    }
```

```
//注册中心注册,只保留核心代码
for (URL registryURL : registryURLs) {

    //通过dubbo spi 使用JavassistProxyFactory 生成包装的invoker
    Invoker<?> invoker = PROXY_FACTORY.getInvoker(ref, (Class) interfaceClass, registryURL.addParameterAndEncoded(EXPORT_KEY, url.toFullString()));
    //包装成统一的wrapperInvoker
    DelegateProviderMetaDataInvoker wrapperInvoker = new DelegateProviderMetaDataInvoker(invoker, this);
    //根据协议走, RegistryProtocol
    Exporter<?> exporter = protocol.export(wrapperInvoker);
    exporters.add(exporter);
}
```

```
public <T> Exporter<T> export(final Invoker<T> originInvoker) throws RpcException {
    URL registryUrl = getRegistryUrl(originInvoker);
    // url to export locally
    URL providerUrl = getProviderUrl(originInvoker);

    final URL overrideSubscribeUrl = getSubscribedOverrideUrl(providerUrl);
    final OverrideListener overrideSubscribeListener = new OverrideListener(overrideSubscribeUrl, originInvoker);
    overrideListeners.put(overrideSubscribeUrl, overrideSubscribeListener);

    providerUrl = overrideUrlWithConfig(providerUrl, overrideSubscribeListener);
    // 核心代码 根据协议进行处理
    final ExporterChangeableWrapper<T> exporter = doLocalExport(originInvoker, providerUrl);
    final Registry registry = getRegistry(originInvoker);
```

```java
    final URL registeredProviderUrl = getRegisteredProviderUrl(providerUrl, registryUrl);
    ProviderInvokerWrapper<T> providerInvokerWrapper = ProviderConsumerRegTable.registerProvider(originInvoker,
            registryUrl, registeredProviderUrl);
    //to judge if we need to delay publish
    boolean register = providerUrl.getParameter(REGISTER_KEY, true);
       // 注册到注册中心
    if (register) {
        register(registryUrl, registeredProviderUrl);
        providerInvokerWrapper.setReg(true);
    }

    // Deprecated! Subscribe to override rules in 2.6.x or before.
    registry.subscribe(overrideSubscribeUrl, overrideSubscribeListener);

    exporter.setRegisterUrl(registeredProviderUrl);
    exporter.setSubscribeUrl(overrideSubscribeUrl);
    //Ensure that a new exporter instance is returned every time export
    return new DestroyableExporter<>(exporter);
}
```

DubboProtocol.java | RegistryProtocol.java | ServiceConfig.java

```java

    @Override
    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, exporterMap);
        exporterMap.put(key, exporter);

        //export an stub service for dispatching event
        Boolean isStubSupportEvent = url.getParameter(STUB_EVENT_KEY, DEFAULT_STUB_EVENT);
        Boolean isCallbackservice = url.getParameter(IS_CALLBACK_SERVICE, defaultValue: false);
        if (isStubSupportEvent && !isCallbackservice) {
            String stubServiceMethods = url.getParameter(STUB_EVENT_METHODS_KEY);
            if (stubServiceMethods == null || stubServiceMethods.length() == 0) {
                if (logger.isWarnEnabled()) {
                    logger.warn(new IllegalStateException("consumer [" + url.getParameter(IN
                            "], has set stubproxy support event ,but no stub methods founded.
                }

            } else {
                stubServiceMethodsMap.put(url.getServiceKey(), stubServiceMethods);
            }
        }

        openServer(url);
        optimizeSerialization(url);

        return exporter;
    }
```

dubbo-2.7.4.1.jar › org › apache › dubbo › remoting › transport › netty4 › NettyServer › doOpen

HeaderExchanger.java | Transporters.java | NettyTransporter.java | DubboProtocol.java | Exchangers.java | NettyServer.java | AbstractServer.java

```java
86      */
87     @Override
88     protected void doOpen() throws Throwable {
89         bootstrap = new ServerBootstrap();
90
91         bossGroup = new NioEventLoopGroup( nThreads: 1, new DefaultThreadFactory( poolName: "NettyServerBoss",  daemon: true));
92         workerGroup = new NioEventLoopGroup(getUrl().getPositiveParameter(IO_THREADS_KEY, Constants.DEFAULT_IO_THREADS),
93                 new DefaultThreadFactory( poolName: "NettyServerWorker",  daemon: true));
94
95         final NettyServerHandler nettyServerHandler = new NettyServerHandler(getUrl(),  handler: this);
96         channels = nettyServerHandler.getChannels();
97
98         bootstrap.group(bossGroup, workerGroup)
99                 .channel(NioServerSocketChannel.class)
100                .childOption(ChannelOption.TCP_NODELAY, Boolean.TRUE)
101                .childOption(ChannelOption.SO_REUSEADDR, Boolean.TRUE)
102                .childOption(ChannelOption.ALLOCATOR, PooledByteBufAllocator.DEFAULT)
103                .childHandler((ChannelInitializer) (ch) → {
106                        // FIXME: should we use getTimeout()?
107                        int idleTimeout = UrlUtils.getIdleTimeout(getUrl());
108                        NettyCodecAdapter adapter = new NettyCodecAdapter(getCodec(), getUrl(),  handler: NettyServer.this);
109                        ch.pipeline()//.addLast("logging",new LoggingHandler(LogLevel.INFO))//for debug
110                                .addLast( s: "decoder", adapter.getDecoder())
111                                .addLast( s: "encoder", adapter.getEncoder())
112                                .addLast( s: "server-idle-handler", new IdleStateHandler( readerIdleTime: 0,  writerIdleTime: 0, idleTimeout, MIL
113                                .addLast( s: "handler", nettyServerHandler);
114                });
116        // bind
117        ChannelFuture channelFuture = bootstrap.bind(getBindAddress());
118        channelFuture.syncUninterruptibly();
119        channel = channelFuture.channel();
120
121    }
```

1个线程

根据配置

故为单线程reactor模型

```java
public NettyServer(URL url, ChannelHandler handler) throws RemotingException {
    //在创建服务的时候，可以ChannelHandlers.wrap 进行了处理器的包装. 这里就是配置的线程模型.
    super(url, ChannelHandlers.wrap(handler, ExecutorUtil.setThreadName(url, SERVER_THREAD_POOL_NAME)));
}
```