# 一、dubbo的spi

## 1.相关注解

**(1) @SPI**

被标注的Class可以使用扩展点进行扩展

**(2) @Adaptive**

Method被修饰表示该方法需要被装饰（代理生成），Class被修饰表示该类是一个装饰类（dubbo中只有AdaptiveExtensionFactory和AdaptiveCompiler符合）,@SPI和@Adaptive中的value会被动态生成的装饰类引用，作为判断URL的条件。

**(3) @Activate**

可以一次得到多个类(List)的扩展方法用来描述Class的使用方与等级，应用于Filter的排序与筛选

## 2. AdaptiveExtensionFactory

装饰了SpiExtensionFactory（dubboIOC）、SpringExtensionFactory（SpringIOC）用来Setter依赖注入。

ServiceBean和ReferenceBean初始化结束之后会将ApplicationContext赋值给SpringExtensionFactory。

## 3. ExtensionLoader

getExtensionLoader(Class<T> type)

EXTENSION\_LOADERS.get(type) //先走缓存调用，不符合走下一行

EXTENSION\_LOADERS.putIfAbsent(type, new ExtensionLoader<T>(type))//初始化之后缓存

new ExtensionLoader<T>(type)

this.type = type;

初始化objectFactory //提供SPI和Spring的IOC功能(类型是ExtensionFactory)

## 4.getAdaptiveExtension

获取@SPI标注类扩展的装饰者对象，如果没有@Adaptive注解修饰T接口的子类，就会动态创建并编译一个装饰类（仅逻辑包装），例如Protocal$Adaptive。

getAdaptiveExtension()

-createAdaptiveExtension()

--getAdaptiveExtensionClass()

getExtensionClasses()

loadExtensionClasses()

loadDirectory() //对指定文件夹(dubbo/internal等)进行加载（SPI）

loadResource()

loadClass()：【1】对SPI加载的类进行缓存

--createAdaptiveExtensionClass() 【2】动态生成和编译一个代理类Protocal$Adaptive

-injectExtension(createAdaptiveExtension().newInstance()): //依赖注入

【1】loadClass 对SPI扩展文件中所有的类进行遍历，如果满足条件会进行缓存

**cachedAdaptiveClass**

被@Adaptive修饰的类，例如AdaptiveExtensionFactory，则保存至该缓存中

**cachedWrapperClasses**

如果这个类没有@Adaptive注解修饰，且构造函数的参数有这个接口(Protocol)，则保存到该缓存中，例如Protocol接口的扩展点只有ProtocolFilterWrapper和ProtocolListenerWrapper以及QosProtocolWrapper可以命中

**cachedActivates**

剩下的类包含@Activate注解的(例如过滤器Filter)。

**cachedNames(cachedClasses)**

如果该类都不满足以上三个缓存

【2】动态生成的类会对类中标注@Adaptive方法进行动态生成（包装，装饰者模式），没有标注@Adaptive的方法生成的代码中直接抛出异常

getExtension(String name)

在动态生成的Protocal$Adaptive有

ExtensionLoader.getExtensionLoader(Protocol.class).getExtension(extName)

## 5.getExtension

getExtension(String name) 获取接口指定name子类的包装类（例如ProtocolFilterWrapper包着ProtocolListenerWrapper包着DubboProtocol）

getExtension(String name):

cachedInstances.get(name)：//先从缓存中尝试获取

createExtension(name)

--getExtensionClasses()：//见4【1】

--injectExtension(T)：//这里用来依赖注入（仅支持setter反射依赖注入）

objectFactory.getExtension(pt, property)：//提供dubbo的IOC

--SPIExtensionFactory. getExtension(type, name)：//先通过SPI注入

ExtensionLoader.getExtensionLoader(type)

return loader.getAdaptiveExtension()

--SpringExtensionFactory. getExtension(type, name)：//再通过Spring注入

return context.getBean(name);

--injectExtension((T) wrapperClass.getConstructor(type).newInstance(instance));

//利用SPI扫出的Wrapper类对扩展类进行包装AOP

# 二、Dubbo嵌入Spring

## 1.dubbo启动spring容器

com.alibaba.dubbo.container.Main.main(args)

String config = ConfigUtils.getProperty(CONTAINER\_KEY, loader.getDefaultExtensionName()

//获得配置参数，使用spring容器/log4j容器

for (...) containers.add(loader.getExtension(args[i])); //根据config参数获取容器

...

container.start() //逐个启动容器

String configPath = ConfigUtils.getProperty(SPRING\_CONFIG)

//默认遍历classpath:META-INF/spring/\*xml

context = new ClassPathXmlApplicationContext(configPath.......)

Context.start(); //启动容器

## 2.spring读取dubbo的BeanDefinition

### (1)spring解析自定义配置

在类路径下

META-INF

dubbo.xsd dubbo在spring中的约束

spring.handlers spring扩展的dubbo解析类

spring.schemas 配置爱dubbo.xsd为解析类

这三个文件就是解析dubbo标签的类

### (2)spring解析<dubbo>开头的BeanDefinition

ApplicationContext的refresh()方法

refresh()

obtainFreshBeanFactory()

this.refreshBeanFactory()

loadBeanDefinitions(beanFactory)

省略…

doc = doLoadDocument(inputSource, resource) //将输入流变成Document

doLoadBeanDefinitions(InputSource inputSource, Resource resource)

documentReader.registerBeanDefinitions(doc…

doRegisterBeanDefinitions(root)

parseBeanDefinitions(root, this.delegate)

delegate.parseCustomElement(ele) //parse非spring的标签

nameSpace= getNamespaceURI(ele)

//得到dubbo的命名空间(xml文件的头)

handler=getNamespaceHandlerResolver().resolve(nameSpace)

// spring.handlers中配的DubboNamespaceHandler【1】

handler.parse(element)

//将<dubbo>标签解析成BeanDefinition

### (3) DubboNamespaceHandler

在【1】处会找到DubboNamespaceHandler全限定名并利用反射构造其实例，并执行init()

public NamespaceHandler resolve(nameSpace)

handlerName=handlerMappings.get(namespaceUri)

handlerClass = Class.forName(handlerName…)

namespaceHandler= BeanUtils.instantiateClass(handlerClass)

namespaceHandler.init()

init()方法加入10个DubboBeanDefinitionParser以解析不同的标签（策略模式）

# 三、服务发布

## 1.服务发布流程（日志）

(1)暴露本地服务

(2)暴露远程服务

(3)启动netty

(4)打开连接zk

(5)注册到zk

(6)订阅(监听)zk(服务治理平台动态改变相关参数)

## 2.ServiceBean监听机制

ServiceBean实现了ApplicationListener<ContextRefreshedEvent>，所以spring容器初始化完成后会向该ServiceBean发送容器初始化完成Event，ServiceBean这时暴露服务

refresh()

finishRefresh() //refresh的最后一步

publishEvent((ApplicationEvent)(new ContextRefreshedEvent(this)

…… (以上是spring部分)

ServiceBean. onApplicationEvent() //执行监听方法

ServiceBean.export() //可执行相关定时启动

doExport() //进行相参数关检查

doExportUrls() //对不同协议进行暴露(http/dubbo/rsi)

loadRegistries(true)

//从dubbo.properties中组装registry的URL信息

doExportUrlsFor1Protocol(protocolConfig, registryURLs)

//每个protocol注册暴露

exportLocal(URL url) //本地暴露（见3（1））

## 3.暴露本地服务

暴露在jvm中，不用调用zk进行远程通信，例如在在一个服务内，自己调用自己的接口，就没有必要建立网络IP连接进行通信

exportLocal(URL url)

### (1)得到invoker

ProxyFactory$Adaptive.getInvoker(ref, (Class) interfaceClass, local)

-- extension = ExtensionLoader

.getExtensionLoader(com.alibaba.dubbo.rpc.ProxyFactory.class)

.getExtension(extName);

//调用SPI容器中的代理类，@SPI默认是javassist的包装类(Wrapper)

-- extension. getInvoker(T proxy, Class<T> type, URL url) （Extension是包装类）

==StubProxyFactoryWrapper. getInvoker(T proxy, Class<T> type, URL url)

//这个包装类内部调用了javassist

JavassistProxyFactory.getInvoker(proxy, type, url)

Wrapper wrapper = Wrapper.getWrapper(DemoServiceImpl.class)

makeWrapper(Class<?> c)

return new AbstractProxyInvoker<T>(proxy, type, url)

//这个AbstractProxyInvoker实际上就是wrapper的代理类，而wrapper是demoServiceImpl的代理， ProxyFactory$Adaptive.getInvoker实际上返回的就是demoServiceImpl的代理

### (2)暴露invoker

Protocol$Adaptive.export(Invoker)

ExtensionLoader.getExtensionLoader(Protocol.class).getExtension(“injvm”)

//得到Protocol的包装类，包装类最内层为injvm

ProtocolFilterWrapper.export(invoker) //在这一层对invoker包装Filters责任链

ProtocolListenerWrapper.export(invoker) //第二层包装

InJvmProtocol.export(invoker)

return new InJvmExporter(invoker,)

exporterMap.put(key, this)

//key=”…DemoService”, value=InjvmExporter

将被层层包装的invoker最终放到InjvmExporter的Map中，最终返回Exporter

## 4.暴露远程服务（启动netty）

暴露给远程客户端的IP和端口号，通过网络来实现通信

### (1)Adaptive代理类到RegistryProtocol的流程(ServiceConfig中)

proxyFactory.getInvoker(ref服务, 接口, zoo的URL)

//获得一个Invoker，与本地暴露类似，只是其中注册的URL不为本地了，而是远程zoo

Protocol$Adaptive .export(Invoker)

ExtensionLoader.getExtensionLoader(Protocol.class).getExtension(“registry”);

//获得RegistryProtocol的包装类

ProtocolFilterWrapper.export(invoker) //RegistryProtocol不会创建Filter的责任链

ProtocolListenerWrapper. export(invoker)

RegistryProtocol. export(invoker) //真正的暴露逻辑

启动netty见(2)、(3)

连接zk见5

### (2)RegistryProtocol.export(invoker)

doLocalExport(originInvoker)

getCacheKey(originInvoker) //获取提供服务URL的字符串

// dubbo://192.168.142.1:20880/com.alibaba.dubbo.demo.DemoService……

Exporter<T> exporter=protocol.export(invokerDelegete)

==Protocol$Adaptive. export(invokerDelegete) //Url中是Dubbo协议

ExtensionLoader.getExtensionLoader(Protocol.class).getExtension(“dubbo”)

//获得的是dubbo协议的包装类

ProtocolFilterWrapper.export() //ProtocolFilterWrapper包的是DubboProtocol

invoker = buildInvokerChain(invoker, XX, “Provider”) // Filters责任链

ProtocolListenerWrapper.export(invoker);

DubboProtocol.export(invoker) //见(3)

连接zookeeper见5

### (3)DubboProtocol.export(invoker) 启动netty

String key = serviceKey(url) //组装key:com.alibaba.dubbo.demo.DemoService:20880

DubboExporter<T> exporter = new DubboExporter<T>(invoker, key, exporterMap) //创建exporter

exporterMap.put(key, exporter) //与本地暴露不同，本地暴露key只有接口名，远程还有port

openServer(url)

ExchangeServer server = createServer(url)

ExchangeServer server = Exchangers.bind(url, requestHandler)

ExchangeServer server = getExchanger(“header”); //默认是header

return ExtensionLoader.getExtensionLoader(Exchanger.class).getExtension(“header”);

server.bind(url, ExchangeHandler)

-DecodeHandler handler2 = new DecodeHandler(new HeaderExchangeHandler(handler))

//new DecodeHandler/ HeaderExchangeHandler(handler)都是为了赋值handler

-Server server = Transporters.bind(url, handler2) //网络传输层

Transporter$Adaptive = ExtensionLoader.getExtensionLoader(Transporter.class).getAdaptiveExtension()

NettyTransporter.bind(url, handler2) //nettyTransport

new NettyServer(url, listener)

AbstractPeer this.url = url; this.handler = handler;

AbstractEndpoint this.codec; this.timeout/ connectTimeout(URL中)

AbstractServer bindAddress/accepts/idleTimeout

doOpen() //打开netty服务器的详细代码() 启动netty

//设置boss、worker线程池，channelFactory

//设置编解码handler，绑定port

-return new HeaderExchangeServer(server)

this.server = server(NettyServer);

this.heartbeat=600000;

this.heartbeatTimeout=180000

startHeartbeatTimer() //心跳定时器，采用了线程池，若断开就重新连接

## 5.dubbo连接zookeeper (RegistryProtocol.export()第二步)

### (1)创建zookeeper客户端()

创建zk的客户端并将URL注册过去

Registry registry = getRegistry(originInvoker)

return registryFactory.getRegistry(registryUrl) //这里是Adaptive代理类

ExtensionLoader.getExtensionLoader(RegistryFactory.class).getExtension(“zookeeper”)

extension.getRegistry(URL)

--AbstractRegistryFactory.getRegistry(URL url)

//创建一个注册中心并存储到REGISTRIES

new ZookeeperRegistry(url, zookeeperTransporter)

AbstractRegistry

loadProperties()

//目的把某文件中的内容加载为properties(key为接口，value是url)

//若zookeeper报错，则使用本地文件中的缓存数据

FailbackRegistry

retryExecutor.scheduleWithFixedDelay(new Runnable()

//建立线程池，检测注册中心，失败重连

ZookeeperRegistry

--zkClient = zookeeperTransporter.connect(url);

//这里是Adaptive代理类

==ZookeeperTransporter$Adaptive.connect(url)

ZkclientZookeeperTransporter.connect(arg0)

new ZkclientZookeeperClient(url) //创建一个zk的客户端

//真正创建一个socket连接和zookeeper连接

client = new ZkClientWrapper(URL)

client.addListener(new IZkStateListener()

//断开连接时重连

client.start()

-- zkClient.addStateListener(new StateListener() //同样是断开重连

--REGISTRIES.put(key, registry)

### (2)dubbo在zookeeper中创建节点(也在RegistryProtocol.export())

register(registryUrl, registeredProviderUrl);

//registryUrl是zookeeper的地址，registeredProviderUrl是dubbo服务地址

registry.register(registedProviderUrl); // registry是已经起来的zookeeper客户端

AbstractRegistry.register(registedProviderUrl)

FailbackRegistry.register(registedProviderUrl)

doRegister(url);

zkClient.create(toUrlPath(url), true) //创建节点

AbstractZookeeperClient. create(String path, boolean ephemeral)

//这个方法是递归方法，按照/来创建节点

path=/dubbo/com.alibaba.dubbo.demo.DemoService/providers/dubbo%3A%2F%2F192.168.142.1%3A20880%2Fcom.alibaba.dubbo.demo.DemoService……….

/开头的都是持久化节点，其他都是临时节点（与客户端会话绑定，session结束删除）

### (3)dubbo订阅zookeeper 信息(RegistryProtocol.export()中)

registry.subscribe(overrideSubscribeUrl, overrideSubscribeListener)

AbstractRegistry.subscribe(…)

FailbackRegistry.subscribe(…)

--doSubscribe(url, listener)

==ZookeeperRegistry. doSubscribe(url, listener)

listeners.putIfAbsent(listener, new ChildListener(){

ZookeeperRegistry.this.notify(url, listener, toUrlsWithEmpty(url, parentPath, currentChilds)) //创建监听器（这个就是要被加入的监听器）

})

zkClient.create(“/dubbo/com.alibaba.dubbo.demo.DemoService/configurators”, false);

//创建一个持久化节点configurators

zkClient.addChildListener(path, listeners) //创建监听

ZkclientZookeeperClient.createTargetChildListener()

new IZkChildListener()

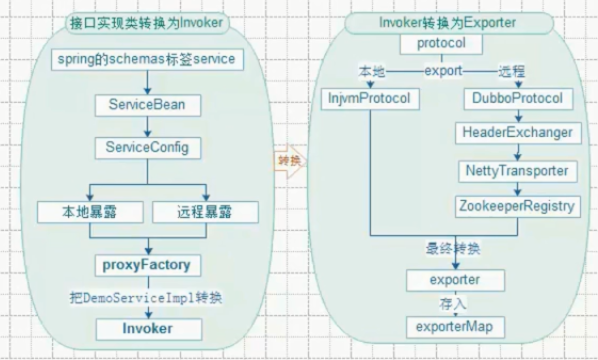
addTargetChildListener(path, targetListener) //加入目标监听

client.subscribeChildChanges(path, listener) //启动加入订阅

//若节点发生改变，则调用FailbackRegistry.notify方法

--FailbackRegistry.notify()

doNotify(url, listener, urls)



# 四、Filter责任链

以LocalExport为例（本地暴露）

Protocol$Adaptive.export(Invoker)

ExtensionLoader.getExtensionLoader(Protocol.class).getExtension(extName);

// extName可从Invoker的URL中获得，所以可以通过URL进行设定

//默认获得的是Protocol的包装类，//ProtocolFilterWrapper和ProtocolListenerWrapper包装的InjvmProtocol

ProtocolFilterWrapper.export(buildInvokerChain(……))

buildInvokerChain(invoker, "service.filter", "provider")

ExtensionLoader.getExtensionLoader(Filter.class).getActivateExtension(invoker.getUrl(), key, group)

//通过SPI扩展获得所有被@Activate修饰的Filter //@Activate注解中包含Filter的顺序与作用范围

for(Filter filter : Filters){

利用每个Filter对invoker进行包装

}

return 所有Filter包装的invoker

# 五、消费端引用服务

流程概述：

①获得与zookeeper的连接

②创建一个RegistryDirectory对象(内部维护了集群容错等信息)，利用RegistryDirectory对象完成对zookeeper进行注册

③RegistryDirectory(实现了NotifyLstener)对zookeeper进行订阅，内部使用ChildListener进行订阅，当ChildListener检测到改变，会激活RegistryDirectory的Notify方法。

④RegistryDirectory的Notify方法会对多个invoker(这里的invoker对应的是服务)进行改变或创建。第一次订阅的时候会拉取providers全部信息，并调用notify()方法创建invoker对象，invoker对象中维护了与provider的TCP长连接Exchanger，但是默认是懒加载，即再第一次调用时才生成。(RegistryDirectory管理了invoker，并在运行期对invoker进行动态更改)。

⑤生成一个外部的invoker(一个注册中心一个invoker)，该invoker是MockClusterWrapper，内部包了一个FailbackClusterInvoker(默认)，FailbackClusterInvoker内部包了RegistryDirectory。

⑥对MockClusterWrapper生成代理类

有两种invoker，一种是服务级别的(例如DubboInvoker)，另一种是注册中心级别的的(例如FailoverClusterInvoker)

## 1.FactoryBean

ReferenceBean实现了接口FactoryBean，所以在getBean的时候容器会调用FactoryBean.getObject()方法

applicationContext.getBean(“demoService”)

getBeanFactory().getBean(“demoService”, DemoService.class)

doGetBean(“demoService”, DemoService.class, null, false)

--Object sharedInstance = getSingleton(beanName) //容器中获取单例的Bean

--getObjectForBeanInstance(sharedInstance, “demoService”, beanName, null)

-FactoryBean<?> factory = (FactoryBean<?>) beanInstance;

//判断是否为FactoryBean

-getObjectFromFactoryBean(factory, beanName,......)

......

factory.getObject() //进入到ReferenceBean的逻辑

## 2.获得Invoker并生成代理

ReferenceConfig.get() //ReferenceBean是ReferenceConfig的子类

init()

...... //进行参数检查

ref = createProxy(map) //为ref创建一个代理对象

loadRegistries(false) //生成注册中心(zookeeper)的URL集合

invoker = refprotocol.refer(interfaceClass, url)) //真正invoker(见(1))

return (T) proxyFactory.getProxy(invoker) //生成代理(见(2))

### (1)获得Invoker

生成一个代理类封装去zookeeper获取远程调用的逻辑，并订阅获取更改消息(新服务上线、旧服务下线)，第一次订阅会拉取全部信息并初始化与provider的TCP连接(若是懒加载则再第一次连接时创建)，并将Exchange保存在invoker中，再对invoker包装责任链

refprotocol.refer(interfaceClass, url) //这是一个Adaptive类

ExtensionLoader.getExtensionLoader(Protocol.class).getExtension(“registry”)

ProtocolFilterWrapper.refer(interfaceClass, url)

ProtocolListenerWrapper.refer(interfaceClass, url)

RegistryProtocol.refer(interfaceClass, url) //两层AOP包装类

Registry registry = registryFactory.getRegistry(url) //获得zookeeper实例

doRefer(cluster, registry, interfaceClass, url)

//单独拎出来分析【1】

【1】去zookeeper注册消费者信息，并订阅，

*最后返回一个MockClusterWrapper(RegistryDirectory(invokers)),一个zoo对应一个*

doRefer(cluster, registry, interfaceClass, url)

RegistryDirectory<T> directory = new RegistryDirectory<T>(type, url);

//生成一个RegistryDirectory，里面存放了调用的URL以及集群容错、负载均衡等组件

//并且持有实际Invoker和接受订阅通知(实现了NotifyListener)，服务变更会通知(订阅)

registry.register(...)

//注册消费者信息给zookeeper(/dubbo/......DemoService/concumers/消费者具体信息)

--directory.subscribe(URL) //【2】订阅zookeeper的节点providers、configurators、routers

//第一次会全量拉取消息，内部调用registry进行订阅

--Invoker invoker = Cluster$Adaptive.join(directory)//URL获取adaptive类(自适应容错机制)

Return MockClusterWrapper.join(arg0);//这里invoker是外层的，不是“字典”里的

Invoker invoker = FailoverCluster.join(directory))

new FailoverClusterInvoker<T>(directory) //被包装的的集群容错类

AbstractClusterInvoker : this.directory = directory

//将directory保存于invoker

new MockClusterInvoker<T>(directory, invoker)

//对FailoverClusterInvoker进行一层外包装(Mock包装)

this.directory = directory;

this.invoker = invoker;

***注：***RegistryDirectory实现了NotifyListener的notify(List<URL> urls)，内部保存了相关集群等逻辑，在【2】处进行订阅(内部其实是给zookeeper的节点new ChildListener()，ChildListener()内部会在zookeeper节点更改时激活NotifyListener的notify方法，并将数据保存于硬盘)。

RegistryDirectory的notify方法会对内部的invoker动态进行改变。(第一次拉取时会进行构建)

directory.subscribe(...)

registry.subscribe(url, this) //利用zookeeper进行订阅，并且自己是一个监听器

doSubscribe(url, listener) //zookeeperRegistry中具体的模板方法

...构建一个ChildListener...(见上面的***注***)

notify(url, listener, urls) //这里的listener就是RegistryDirectory

...

registryDirectory.refreshInvoker(invokerUrls)

//进行构建invoker，其中有筛选功能

toInvokers(invokerUrls) //具体处理逻辑

invoker = new InvokerDelegate<T>(protocol.refer(serviceType, url),...) 【3】单独拎出来 具体的创建invoker

【3】创建invoker

protocol.refer(serviceType, url) //使用具体协议创建远程连接

ExtensionLoader.getExtensionLoader(Protocol.class).getExtension(extName) //Adaptive

ProtocolFilterWrapper.refer(type, URL) //构造责任链

ProtocolListenerWrapper.refer(type, url)

DubboProtocol.refer(Class<T> serviceType, URL url) //建立远程连接等

buildInvokerChain(...) //建立客户端责任链

### (2)生成代理

proxyFactory.getProxy(invoker) //Adaptive代理类

StubProxyFactoryWrapper.getProxy(arg0)

//Wrapper包装类，内部默认包装了JavassistProxyFactory

javassistProxyFactory.getProxy(invoker) //调用javassistProxyFactory进行生成代理

interfaces = new Class<?>[]{invoker.getInterface(), EchoService.class}

//代理类会实现两个接口，一个是DemoService，第二个是EchoService回声测试

return getProxy(invoker, interfaces)

return (T) Proxy.getProxy(interfaces).newInstance(new InvokerInvocationHandler(invoker))

最终将代理返回给最外层调用的ReferenceConfig并保存于其中

最终的包装：

InvokerInvocationHandler //添加一些Object方法的封装(toString()等)

MockClusterInvoker //包装一层Mock层(调用错误返回mock值)

Registrydirectory //包含路由、负载均衡等

FailoverClusterInvoker //容错机制

## 3.服务端调用

# 附录：相关名词解释

## 1.ProxyFactory的作用

为了获得一个接口的代理类，例如获得一个远程接口的代理

ProxyFactory有两个方法，代表两个作用

getInvoker：针对server端，将服务对象，如DemoServiceImpl包装成一个invoker对象。

getProxy：针对client端，用来创建接口的代理对象，例如DemoService接口。

## 2.wrapper的作用

类似spring的BeanWrapper，它就是包装了一个接口/类，可以通过wrapper对实例对象进行赋值/取值/指定方法的调用

## 3.Invoker的作用

是一个可执行的对象，能根据方法的名称、参数得到相应的执行结果

有一个很重要的方法 Result invoke(Invocation invocation)

Invocation作用是包含了需要执行的方法/参数类型/参数值（远程序列化或是本地传过来的）

它有两个实现类RPCInvocation、MockInvocation

Invoker有三种类型：

本地执行类的Invoker

远程通信类的Invoker

多个远程通信执行类的Invoker聚合成集群版的Invoker

## 4.Protocol

(1)export

暴露远程服务（服务端）：将proxyFactory.getInvoker创建的代理类invoker对象，通过协议暴露给外部

(2)refer

应用远程服务（客户端）：

## 5.Exporter

维护invoker的生命周期

## 6.exchanger

信息交换层，封装请求响应模式，同步转异步

## 7.transport

网络传输层，用来抽象netty和mina的统一接口