Dubbo SPI扩展点自定义适配

getAdaptiveExtension

createAdaptiveExtension

getAdaptiveExtensionClass

getExtensionClasses

获取当前扩展点的所有适配，从META-INF/。。。文件，其中如果有Class的类加了@Adaptive注解，那么将这个适配Class放入缓存，这个操作是遍历执行，如果有2个类都配置，会报错
```
if(cachedAdaptiveClass == null) {
                                cachedAdaptiveClass = clazz;
                            } else if (! cachedAdaptiveClass.equals(clazz)) {
                                throw new IllegalStateException("More than 1
adaptive class found: "
                                        + cachedAdaptiveClass.getClass().getName()
                                        + ", " + clazz.getClass().getName());
                            }
```
，还会验证是否存在Wapper装饰类，包含组合关系，那么会进行装饰

如果缓存存在，直接返回否则执行下面的创建

createAdaptiveExtensionClass

创建一个 扩展点name$Adaptive 的动态适配Class，调用AdaptiveCompiler（这个是SPI动态适配出来的）.compile(code, classLoader) JVM加载

AdaptiveCompiler.compile(code, classLoader)

injectExtension 注入操作

判断是否存在set起始的方法，如果存在通过objectFactory的 getExtension(pt, property); 进行参数的实例注入

因为set的方法 都是组合方法 ，将对应的接口实现类赋值，例如

```
private Protocol pro;

public void setProtocol (Protocol pro){
    this.pro = pro;
}
```

这里 pro为一个接口，方法参数pro为找到的对应适配实现类，进行实例赋值。

```
private ExtensionLoader(Class<?> type) {
        this.type = type;
        objectFactory = (type == ExtensionFactory.class ? null : ExtensionLoader.getExtensionLoader(ExtensionFactory.class).getAdaptiveExtension());
    }
```

objectFactory在 上面方法中进行赋值，返回的是AdaptiveExtensionFactory，因为使用了类的@Adaptive注解

dubbo 服务发布 采用方式

基于Spring采用自定义扩展标签方式实现， 通过 META-INF/spring.handlers文件下寻找对应的Namespacehandler,注册一堆BeanDefinitionParser来解析。Spring启动的时候服务端就会启动发布服务注册服务。

入口

ServiceBean

ServiceBean实现了多个Spring接口
InitializingBean， 实现afterPropertiesSet 方法
DisposableBean，实现destroy 方法
ApplicationContextAware，实现setApplicationContext方法
ApplicationListener， 实现onApplicationEvent 方法
BeanNameAware 实现setBeanName方法
这些方法在Spring启动过程中会自动调用。

入口在afterPropertiesSet中，调用export 方法进入发布服务

读取配置数据，拼接URL

ServiceConfig

export() -> 判断是否配置延时启动，调用doExport()

doExport() -> 读取XML配置项，赋值处理，调用doExportUrls()

doExportUrls() -> 读取配置的所有注册中心信息，
遍历配置的Protocol协议，调用doExportUrlsFor1Protocol

doExportUrlsFor1Protocol() -> 获取本机IP地址及配置的端口，组装URL。
如果设置了注册中心获取Invoker，发布服务exporter。
如果未设置（点对点），也会获取Invoker，发布服务exporter 通过自定义扩展点来实现不同适配调用

StubProxyFactoryWrapper

getInvoker() -> 通过proxyFactory扩展点，会执行StubProxyFactoryWrapper，不过里面直接调用JavassistProxyFactory。

JavassistProxyFactory

getInvoker() -> 执行getWrapper 生成动态代理类，返回AbstractProxyInvoker，其中定义doInvoke方法，方法里面执行动态代理类的invokeMethod方法

服务端的代理类 AbstractProxyInvoker

配置注册中心

RegistryProtocol

生成代理类后，继续回到Config中执行export() ->发布本地服务doLocalExport
封装InvokerDelegete，再次通过自定义适配器protocol方式调用，
这里会先调用装饰类
ProtocolFilterWrapper（ProtocolListenerWrapper（DubboProtocol））
执行过滤链路，之后到dubboProtocol 中的export 通过netty发布服务监听

通过getRegistry，根据配置的注册中心地址获取注册中心Registry

调用注册方法regiter，在注册中心创建节点，注册监听

发布服务

dubboProtocol

创建心跳机制，openServer通过netty发布服务监听

未配置注册中心

StubProxyFactoryWrapper

getInvoker() -> 通过proxyFactory扩展点，会执行StubProxyFactoryWrapper，不过里面直接调用JavassistProxyFactory。

JavassistProxyFactory

getInvoker() -> 执行getWrapper 生成动态代理类，返回AbstractProxyInvoker，其中定义doInvoke方法，方法里面执行动态代理类的invokeMethod方法

服务端的代理类 AbstractProxyInvoker

dubboProtocol

创建心跳机制，openServer通过netty发布服务监听

dubbo 消费调用 采用方式

基于Spring采用自定义扩展标签方式实现， 通过 META-INF/spring.handlers文件下寻找对应的Namespacehandler,注册一堆BeanDefinitionParser来解析。Spring启动的时候客户端就会启动消费调用。

入口

ReferenceBean

ReferenceBean同样实现了多个Spring接口
InitializingBean， 实现afterPropertiesSet 方法
DisposableBean，实现destroy 方法
ApplicationContextAware，实现setApplicationContext方法
FactoryBean， 实现getObject 方法
这些方法在Spring启动过程中会自动调用。

入口在afterPropertiesSet中，调用getObject 方法再调用父类ReferenceConfig中得get方法进行初始化

创建代理类及会话

ReferenceConfig

通过init方法，加载所有配置项内容，通过createProxy获取代理类

如果同一个JVM，依然使用protocol扩展点，进入InjvmProtocol中调用refer方法获取invoker（这里是个DubboInvoker）。
如果点对点连接，设置urls（调用url列表）。否则获取注册中心的配置loadRegistries，设置urls。如果urls只有一个地址，掉用RegistryProtocol中的refer。
遍历urls，对每个地址执行 refprotocol.refer。 验证地址中是否存在注册中心的配置地址，如果存在，使用AvailableCluster.join(new StaticDirectory(u, invokers));返回AbstractClusterInvoker。
否则是点对点多地址，返回FailoverClusterInvoker。然后生成proxyFactory.getProxy(invoker);动态代理类

RegistryProtocol

refer-> 获取注册中心，调用doRefer，注册consumer地址，调用directory.subscribe方法

FailbackRegistry

subscribe->执行doSubscribe

ZookeeperRegistry

doSubscribe->创建其他三文件监听，执行notify

AbstractRegistry

notify-> 做本地缓存及更新操作（RegistryDirectory）路由规则等设置，并且将invoke加入到缓存集合中。
之后调用refreshInvoker->toInvokers->InvokerDelegete方法的参数中调用protocol扩展点，会new DubboInvoker(serviceType, url, this.getClients(url), this.invokers); 其中getClients中会创建netty通信

创建连接

StubProxyFactoryWrapper

做一些参数处理，调用JavassistProxyFactory中得getProxy

JavassistProxyFactory

返回动态代理类Proxy.getProxy(interfaces).newInstance(new InvokerInvocationHandler(invoker))

返回proxy0代理类

服务调用

动态代理类Proxy0

调用动态代理类，执行InvokerInvocationHandler中的invoke

InvokerInvocationHandler

执行MockClusterInvoker中的invoke

MockClusterInvoker

invoke->会按配置项判断执行mock机制，调用AbstractClusterInvoker中invoke

AbstractClusterInvoker

invoke 通过模板方法调用FailoverClusterInvoker中doInvoke

FailoverClusterInvoker

调用AbstractClusterInvoker 中list 通过directory.list(invocation)，获取Invoker列表。调用AbstractClusterInvoker 中 select后调用doselect 进行获取invoke操作。通过AbstractLoadBalance中select

AbstractLoadBalance

通过doSelect模板方法，默认调用RandomLoadBalance中的doSelect实现

RandomLoadBalance

使用 随机+权重方式 计算出 invoke返回，在FailoverClusterInvoker中继续执行Result result = invoker.invoke(invocation); 通过dubboInvoker中invoker获取返回内容

Filter/Listener

执行过滤和监听调用链

dubboInvoker

底层最后通过netty发生请求

服务端接收调用

入口

服务端发布服务时在nettyServer中会执行doOpen，这个方法中会开启一个channel。其中设置NettyHandler进行处理

NettyHandler

通过messageReceived作为入口，执行handler.received。而这里handler是什么呢？在发布服务过程中，进行了多层的调用组装
MultiMessageHandler: 复合消息处理
HeartbeatHandler：心跳消息处理，接收心跳并发送心跳响应
AllChannelHandler：业务线程转化处理器，把接收到的消息封装成ChannelEventRunnable可执行任务给线程池处理
DecodeHandler:业务解码处理器
HeaderExchangeHandler：交互层请求响应处理
ExchangeHandlerAdapter：调用DubboProtocol中定义的ExchangeHandlerAdaptive.replay方法处理消息

ExchangeHandlerAdapter

在调用DubboProtocol中 ExchangeHandler requestHandler = new ExchangeHandlerAdapter() 实例了一个ExchangeHandlerAdapter对象，并且复写了reply方法，最后执行invoker.invoke(inv)，而invoke为Filter(Listener(InvokerDelegete(AbstractProxyInvoker (Wrapper.invokeMethod)))

RMI

服务端发布操作：

Hello hello = new HelloImp();
服务端在实例化时，操作

首先， 实现类必须继承UnicastRemoteObject， 然后增加构造方法 调用父类构造Super();

在UnicastRemoteObject 构造方法中开始调用exportObject方法，进行发布。

服务端开始创建代理类，生成HelloImp实例的代理类，Util中的createProxy 会创建stub副本。
因为所有类继承或实现Remote接口的类，所以强转为Remote类型，
然后封装成一个Target类，进行TCP协议发布服务。会进入到TCPTransport中 开启ServerSocket端口，因为没有传入端口信息，所以端口会随机生成一个。并且会将地址记录到ObjectTable中的objTable与implTable 变量（HashMap）中。

LocateRegistry.createRegistry(端口号)

会先实例一个RegistryImpl对象，然后又会生成一个代理类，这个是生成RegistryImpl 的代理类（这里代理类系统已经存在，所以会返回这个类的实例），同样强制转为Remote类型，然后封装成一个Target类，按照指定端口 新建ServerScoket端口，并且会将地址记录到ObjectTable中的objTable与implTable 变量（HashMap）中

Naming.rebind("rmi://127.0.0.1/Hello",hello);
协议与IP地址与服务名，实例

调用RegistryImpl_Stub中rebind方法，将name等信息写入到注册中心

客户端调用操作：

Hello hello = (Hello)Naming.lookup("rmi://127.0.0.1/Hello");

获取到 registryImp_stub副本，调用其中的newCall方法， 与服务器的SKeleton对象建立连接，然后调用invoke方法，通过 var1.executeCall(); 建立连接,返回一个HelloImp_stub的存根，不过会转为Remote类型

服务端接收调用操作：

在发布服务时，服务端在listen()方法中启动了一个线程
(Thread)AccessController.doPrivileged(new NewThreadAction(new CPTransport.AcceptLoop(this.server), "TCP Accept-" + var2, true)); 具体在run方法中当建立连接后会从线程池中开启线程。
TCPTransport.connectionThreadPool.execute(TCPTransport.this.new ConnectionHandler(var1, var3));
最后执行到handleMessages方法 再执行serviceCall方法， 根据之前存入流中的请求地址和方法来从之前记录的ObjectTable中的objTable中获取对应的Target ，再获取var5.getDispatcher(); 再通过 var6.dispatch(var37, var1); 获取到对应的方法 通过var8.invoke(var1, var10); 执行方法

hello.sayHello("hhh")