# 深度剖析Apache Dubbo

# Dubbo基础

## 1.1 初始Dubbo

Register

Consumer

Monitor

Provider

Container

1. subscribe

3.notify

1.register

0.start

5.invoke

5.count

init

async

sync

1. Provider为服务提供者集群，服务提供者负责暴露提供的服务，并将服务注册到服务注册中心。
2. Consumer为服务消费者集群，服务消费者通过RPC远程调用服务者提供的服务。
3. Register负责服务注册与发现
4. Monitor为监控中心，统计服务的调用次数和调用时间。

各组件的关系：

1. 服务提供方在启动时会将自己提供的服务注册到服务注册中心。
2. 服务消费方在启动时会去服务注册中心订阅自己需要的服务的地址列表，然后服务注册中心异步把消费方需要的服务接口的提供者的地址列表返回给服务消费方，服务消费方根据路由规则的负载均衡算法选者一个服务提供者IP进行调用。
3. 监控平台主要用来统计服务的调用次数和调用耗时，即服务消费者和提供者在内存中累计调用服务的次数和耗时，并每分钟定时发送一次统计数据到监控中心，监控中心则使用数据回执图表来显示。监控平台不是分布式系统必须的，但是这些数据有足浴系统的运维和调优。服务提供者和消费者可以直接配置监控平台的地址，也可以通过服务注册中心获取。

## 1.2 初始Dubbo

### 1.2.1 Demo

# Dubbo框架内核原理剖析

## 2.1 Dubbo 分层架构概述

主要模块：

1. Service和Config层为API接口层，是为了让Dubbo使用方便地发布服务和引用服务；对于服务提供方来说需要实现服务接口，然后使用ServiceConfig API来发布服务。对于服务消费方来说需要使用ReferenceConfig对服务接口进行代理。
2. 其他各层均为SPI（Service Provider Interface，服务提供者接口）层，
3. Proxy服务代理层：该层主要是对服务消费端使用的接口进行代理，把本地调用透明地转换为远程调用；另外对服务提供方的服务实现类进行代理。
4. Registry服务注册中心层：服务提供者启动时会把服务注册到服务注册中心，消费者启动时会去服务注册中心获取服务提供者的地址列表。Registry层主要功能是封装服务地址的注册与发现逻辑。
5. Cluster路由层：封装多个服务提供者的路由规则、负责均衡、集群容错的实现，并桥接服务注册中心；扩展接口Cluster对应的实现类有FailoverCluster（失败重试）、FailbackCluster（失败自动恢复）、FailfastCluster（快速失败）、FailsafeCluster（失败安全）、ForkingCluster(并行调用)等
6. Monitor监控层：用来统计RPC调用次数和调用耗时时间，扩展接口为MonitorFactory.
7. Protocol远程调用层：封装RPC调用逻辑，扩展接口为Protocol，对应实现有RegistryProtocol、DubboProtocol、InjvmProtocol等。
8. Exchange信息交换层：封装请求响应模式，同步转异步，扩展接口为Exchanger，对应的扩展实现有HeaderExchanger等。
9. Transport网络传输层：Mina和Netty抽象统一接口。扩展接口为Channel。
10. Serialize数据序列化层：提供可以复用的一些工具，扩展接口为Serialization，对应的扩展实现有DubboSerialization、FastJsonSerialization、Hessian2Serialization、JavaSerialization等。

# Dubbo远程服务发布与引用流程剖析

## 3.1 Dubbo服务端发布端启动流程剖析

步骤1：ServiceConfig#export()方法来激活发布服务。

public synchronized void export() {  
 checkAndUpdateSubConfigs();

# 是否需要导出服务  
 if (!shouldExport()) {  
 return;  
 }  
 # 延迟发布  
 if (shouldDelay()) {  
 *delayExportExecutor*.schedule(this::doExport, delay, TimeUnit.*MILLISECONDS*);  
 } else {

# 直接发布  
 doExport();  
 }  
}

步骤2：ServiceConfig#doExport()，主要是对ServiceConfig进行合法性检查，这里主要看doExportUrls

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();  
}

步骤3：ServiceConfig#doExportUrls()

private void doExportUrls() {

# 加载所有的服务注册中心对象  
 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);  
 ApplicationModel.*initProviderModel*(pathKey, providerModel);  
 doExportUrlsFor1Protocol(protocolConfig, registryURLs);  
 }  
}

步骤4：ServiceConfig#loadRegistries()加载所有的注册中心

protected List<URL> loadRegistries(boolean provider) {  
 // check && override if necessary  
 List<URL> registryList = new ArrayList<URL>();  
 if (CollectionUtils.*isNotEmpty*(registries)) {  
 for (RegistryConfig config : registries) {  
 String address = config.getAddress();  
 if (StringUtils.*isEmpty*(address)) {  
 address = Constants.*ANYHOST\_VALUE*;  
 }  
 if (!RegistryConfig.*NO\_AVAILABLE*.equalsIgnoreCase(address)) {  
 Map<String, String> map = new HashMap<String, String>();  
 *appendParameters*(map, application);  
 *appendParameters*(map, config);  
 map.put(Constants.*PATH\_KEY*, RegistryService.class.getName());  
 *appendRuntimeParameters*(map);  
 if (!map.containsKey(Constants.*PROTOCOL\_KEY*)) {  
 map.put(Constants.*PROTOCOL\_KEY*, Constants.*DUBBO\_PROTOCOL*);  
 }  
 List<URL> urls = UrlUtils.*parseURLs*(address, map);  
  
 for (URL url : urls) {  
 url = URLBuilder.*from*(url)  
 .addParameter(Constants.*REGISTRY\_KEY*, url.getProtocol())  
 .setProtocol(Constants.*REGISTRY\_PROTOCOL*)  
 .build();  
 if ((provider && url.getParameter(Constants.*REGISTER\_KEY*, true))  
 || (!provider && url.getParameter(Constants.*SUBSCRIBE\_KEY*, true))) {  
 registryList.add(url);  
 }  
 }  
 }  
 }  
 }  
 return registryList;  
}

步骤5：ServiceConfig#dpExportUrlsFor1Protocol

private void doExportUrlsFor1Protocol(ProtocolConfig protocolConfig, List<URL> registryURLs) {  
 String name = protocolConfig.getName();  
 if (StringUtils.*isEmpty*(name)) {  
 name = Constants.*DUBBO*;  
 }  
  
 Map<String, String> map = new HashMap<String, String>();  
 map.put(Constants.*SIDE\_KEY*, Constants.*PROVIDER\_SIDE*);  
 *appendRuntimeParameters*(map);  
 *appendParameters*(map, application);  
 *appendParameters*(map, module);  
 *appendParameters*(map, provider, Constants.*DEFAULT\_KEY*);  
 *appendParameters*(map, protocolConfig);  
 *appendParameters*(map, this);

# 解析MethodConfig配置  
 if (CollectionUtils.*isNotEmpty*(methods)) {  
 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 (CollectionUtils.*isNotEmpty*(arguments)) {  
 for (ArgumentConfig argument : arguments) {  
 // convert argument type  
 if (argument.getType() != null && argument.getType().length() > 0) {  
 Method[] methods = interfaceClass.getMethods();  
 // visit all methods  
 if (methods != null && methods.length > 0) {  
 for (int i = 0; i < methods.length; i++) {  
 String methodName = methods[i].getName();  
 // target the method, and get its signature  
 if (methodName.equals(method.getName())) {  
 Class<?>[] argtypes = methods[i].getParameterTypes();  
 // one callback in the method  
 if (argument.getIndex() != -1) {  
 if (argtypes[argument.getIndex()].getName().equals(argument.getType())) {  
 *appendParameters*(map, argument, method.getName() + "." + argument.getIndex());  
 } else {  
 throw new IllegalArgumentException("Argument config error : the index attribute and type attribute not match :index :" + argument.getIndex() + ", type:" + argument.getType());  
 }  
 } else {  
 // multiple callbacks in the method  
 for (int j = 0; j < argtypes.length; j++) {  
 Class<?> argclazz = argtypes[j];  
 if (argclazz.getName().equals(argument.getType())) {  
 *appendParameters*(map, argument, method.getName() + "." + j);  
 if (argument.getIndex() != -1 && argument.getIndex() != j) {  
 throw new IllegalArgumentException("Argument config error : the index attribute and type attribute not match :index :" + argument.getIndex() + ", type:" + argument.getType());  
 }  
 }  
 }  
 }  
 }  
 }  
 }  
 } else if (argument.getIndex() != -1) {  
 *appendParameters*(map, argument, method.getName() + "." + argument.getIndex());  
 } else {  
 throw new IllegalArgumentException("Argument config must set index or type attribute.eg: <dubbo:argument index='0' .../> or <dubbo:argument type=xxx .../>");  
 }  
  
 }  
 }  
 } // end of methods for  
 }

# 如果是泛型调用，设置泛型类型  
 if (ProtocolUtils.*isGeneric*(generic)) {  
 map.put(Constants.*GENERIC\_KEY*, generic);  
 map.put(Constants.*METHODS\_KEY*, Constants.*ANY\_VALUE*);  
 } else {

# 正常调用拼接URL的参数  
 String revision = Version.*getVersion*(interfaceClass, version);  
 if (revision != null && revision.length() > 0) {  
 map.put("revision", revision);  
 }  
  
 String[] methods = Wrapper.*getWrapper*(interfaceClass).getMethodNames();  
 if (methods.length == 0) {  
 *logger*.warn("No method found in service interface " + interfaceClass.getName());  
 map.put(Constants.*METHODS\_KEY*, Constants.*ANY\_VALUE*);  
 } else {  
 map.put(Constants.*METHODS\_KEY*, StringUtils.*join*(new HashSet<String>(Arrays.*asList*(methods)), ","));  
 }  
 }  
 if (!ConfigUtils.*isEmpty*(token)) {  
 if (ConfigUtils.*isDefault*(token)) {  
 map.put(Constants.*TOKEN\_KEY*, UUID.*randomUUID*().toString());  
 } else {  
 map.put(Constants.*TOKEN\_KEY*, token);  
 }  
 }

# 拼接URL对象  
 // export service  
 String host = this.findConfigedHosts(protocolConfig, registryURLs, map);  
 Integer port = this.findConfigedPorts(protocolConfig, name, map);  
 URL url = new URL(name, host, port, getContextPath(protocolConfig).map(p -> p + "/" + path).orElse(path), map);  
  
 if (ExtensionLoader.*getExtensionLoader*(ConfiguratorFactory.class)  
 .hasExtension(url.getProtocol())) {  
 url = ExtensionLoader.*getExtensionLoader*(ConfiguratorFactory.class)  
 .getExtension(url.getProtocol()).getConfigurator(url).configure(url);  
 }

# 导出服务，本地服务，远程服务  
 String scope = url.getParameter(Constants.*SCOPE\_KEY*);  
 // don't export when none is configured

# 如果scope为SCOPE\_NONE则不导出服务  
 if (!Constants.*SCOPE\_NONE*.equalsIgnoreCase(scope)) {  
  
 // export to local if the config is not remote (export to remote only when config is remote)

# 5.1 如果scope不是SCOPE\_REMOTE，则导出本地服务  
 if (!Constants.*SCOPE\_REMOTE*.equalsIgnoreCase(scope)) {  
 exportLocal(url);  
 }  
 // export to remote if the config is not local (export to local only when config is local)

# 5.2 如果scope不是SCOPE\_LOCAL，则导出远程服务  
 if (!Constants.*SCOPE\_LOCAL*.equalsIgnoreCase(scope)) {  
 if (*logger*.isInfoEnabled()) {  
 *logger*.info("Export dubbo service " + interfaceClass.getName() + " to url " + url);  
 }

# 5.2.1 如果有服务注册中心地址  
 if (CollectionUtils.*isNotEmpty*(registryURLs)) {  
 for (URL registryURL : registryURLs) {  
 url = url.addParameterIfAbsent(Constants.*DYNAMIC\_KEY*, registryURL.getParameter(Constants.*DYNAMIC\_KEY*));  
 URL monitorUrl = loadMonitor(registryURL);  
 if (monitorUrl != null) {  
 url = url.addParameterAndEncoded(Constants.*MONITOR\_KEY*, monitorUrl.toFullString());  
 }  
 if (*logger*.isInfoEnabled()) {  
 *logger*.info("Register dubbo service " + interfaceClass.getName() + " url " + url + " to registry " + registryURL);  
 }  
  
 // For providers, this is used to enable custom proxy to generate invoker  
 String proxy = url.getParameter(Constants.*PROXY\_KEY*);  
 if (StringUtils.*isNotEmpty*(proxy)) {  
 registryURL = registryURL.addParameter(Constants.*PROXY\_KEY*, proxy);  
 }  
  
 Invoker<?> invoker = *proxyFactory*.getInvoker(ref, (Class) interfaceClass, registryURL.addParameterAndEncoded(Constants.*EXPORT\_KEY*, url.toFullString()));  
 DelegateProviderMetaDataInvoker wrapperInvoker = new DelegateProviderMetaDataInvoker(invoker, this);  
  
 Exporter<?> exporter = *protocol*.export(wrapperInvoker);  
 exporters.add(exporter);  
 }  
 } else {

# 5.2.2 直连方式  
 Invoker<?> invoker = *proxyFactory*.getInvoker(ref, (Class) interfaceClass, url);  
 DelegateProviderMetaDataInvoker wrapperInvoker = new DelegateProviderMetaDataInvoker(invoker, this);  
  
 Exporter<?> exporter = *protocol*.export(wrapperInvoker);  
 exporters.add(exporter);  
 }  
 */\*\*  
 \** ***@since*** *2.7.0  
 \* ServiceData Store  
 \*/*

*# 元数据存储* MetadataReportService metadataReportService = null;  
 if ((metadataReportService = getMetadataReportService()) != null) {  
 metadataReportService.publishProvider(url);  
 }  
 }  
 }  
 this.urls.add(url);  
}

## 3.3 Dubbo服务消费方启动流程剖析

从ReferenceConfig.get()方法开始

|  |
| --- |
| package org.apache.dubbo.demo.consumer;  import com.books.dubbo.demo.api.GreetingService; import org.apache.dubbo.config.ApplicationConfig; import org.apache.dubbo.config.ReferenceConfig; import org.apache.dubbo.config.RegistryConfig; import org.apache.dubbo.rpc.RpcContext;  public class ApiConsumer {    public static void main(String[] args) throws InterruptedException {   // 10.创建服务引用对象实例  ReferenceConfig<GreetingService> referenceConfig = new ReferenceConfig<GreetingService>();  // 11.设置应用程序信息  referenceConfig.setApplication(new ApplicationConfig("first-dubbo-consumer"));  // 12.设置服务注册中心  referenceConfig.setRegistry(new RegistryConfig("zookeeper://127.0.0.1:2181"));  // 直连测试  // referenceConfig.setUrl("dubbo://192.168.0.109:20880");   // 13.设置服务接口和超时时间  referenceConfig.setInterface(GreetingService.class);  referenceConfig.setTimeout(5000);   // 14.设置自定义负载均衡策略与集群错策略 // referenceConfig.setLoadbalance("myroundrobin"); // referenceConfig.setCluster("myCluster"); // RpcContext.getContext().set("ip", "30.10.67.231");    // 15.设置服务分组与版本  referenceConfig.setVersion("1.0.0");  referenceConfig.setGroup("dubbo");   // 16.引用服务  GreetingService greetingService = referenceConfig.get();   // 17.设置隐式参数  RpcContext.*getContext*().setAttachment("company", "alibaba");   // 18.调用服务  System.*out*.println(greetingService.sayHello("world"));  Thread.*currentThread*().join();     }   } |

### ReferenceConfig#get()

|  |
| --- |
| public synchronized T get() {  checkAndUpdateSubConfigs();   if (destroyed) {  throw new IllegalStateException("The invoker of ReferenceConfig(" + url + ") has already destroyed!");  }  if (ref == null) {  init();  }  return ref; } |

### ReferenceConfig#init()

主要关注createProxy方法

|  |
| --- |
| private void init() {  if (initialized) {  return;  }  initialized = true;  checkStubAndLocal(interfaceClass);  checkMock(interfaceClass);  Map<String, String> map = new HashMap<String, String>();   map.put(Constants.*SIDE\_KEY*, Constants.*CONSUMER\_SIDE*);  *appendRuntimeParameters*(map);  if (!isGeneric()) {  String revision = Version.*getVersion*(interfaceClass, version);  if (revision != null && revision.length() > 0) {  map.put("revision", revision);  }   String[] methods = Wrapper.*getWrapper*(interfaceClass).getMethodNames();  if (methods.length == 0) {  *logger*.warn("No method found in service interface " + interfaceClass.getName());  map.put("methods", Constants.*ANY\_VALUE*);  } else {  map.put("methods", StringUtils.*join*(new HashSet<String>(Arrays.*asList*(methods)), ","));  }  }  map.put(Constants.*INTERFACE\_KEY*, interfaceName);  *appendParameters*(map, application);  *appendParameters*(map, module);  *appendParameters*(map, consumer, Constants.*DEFAULT\_KEY*);  *appendParameters*(map, this);  Map<String, Object> attributes = null;  if (CollectionUtils.*isNotEmpty*(methods)) {  attributes = new HashMap<String, Object>();  for (MethodConfig methodConfig : methods) {  *appendParameters*(map, methodConfig, methodConfig.getName());  String retryKey = methodConfig.getName() + ".retry";  if (map.containsKey(retryKey)) {  String retryValue = map.remove(retryKey);  if ("false".equals(retryValue)) {  map.put(methodConfig.getName() + ".retries", "0");  }  }  attributes.put(methodConfig.getName(), *convertMethodConfig2AyncInfo*(methodConfig));  }  }   String hostToRegistry = ConfigUtils.*getSystemProperty*(Constants.*DUBBO\_IP\_TO\_REGISTRY*);  if (StringUtils.*isEmpty*(hostToRegistry)) {  hostToRegistry = NetUtils.*getLocalHost*();  }  map.put(Constants.*REGISTER\_IP\_KEY*, hostToRegistry);  # 创建代理的方法  ref = createProxy(map);   String serviceKey = URL.*buildKey*(interfaceName, group, version);  ApplicationModel.*initConsumerModel*(serviceKey, buildConsumerModel(serviceKey, attributes)); } |

### ReferenceConfig#createProxy(map)

|  |
| --- |
| private T createProxy(Map<String, String> map) {  # 1）是否需要打开本地引用  if (shouldJvmRefer(map)) {  URL url = new URL(Constants.*LOCAL\_PROTOCOL*, Constants.*LOCALHOST\_VALUE*, 0, interfaceClass.getName()).addParameters(map);  invoker = *refprotocol*.refer(interfaceClass, url);  if (*logger*.isInfoEnabled()) {  *logger*.info("Using injvm service " + interfaceClass.getName());  }  } else {  # 2）用户是否指定服务提供地址： 可以是服务提供方IP地址（智联方式）  if (url != null && url.length() > 0) { // user specified URL, could be peer-to-peer address, or register center's address.  String[] us = Constants.*SEMICOLON\_SPLIT\_PATTERN*.split(url);  if (us != null && us.length > 0) {  for (String u : us) {  URL url = URL.*valueOf*(u);  if (StringUtils.*isEmpty*(url.getPath())) {  url = url.setPath(interfaceName);  }  if (Constants.*REGISTRY\_PROTOCOL*.equals(url.getProtocol())) {  urls.add(url.addParameterAndEncoded(Constants.*REFER\_KEY*, StringUtils.*toQueryString*(map)));  } else {  urls.add(ClusterUtils.*mergeUrl*(url, map));  }  }  }  } else { // assemble URL from register center's configuration  # 3）根据服务注册中心信息装配URL对象  checkRegistry();  List<URL> us = loadRegistries(false);  if (CollectionUtils.*isNotEmpty*(us)) {  for (URL u : us) {  URL monitorUrl = loadMonitor(u);  if (monitorUrl != null) {  map.put(Constants.*MONITOR\_KEY*, URL.*encode*(monitorUrl.toFullString()));  }  urls.add(u.addParameterAndEncoded(Constants.*REFER\_KEY*, StringUtils.*toQueryString*(map)));  }  }  if (urls.isEmpty()) {  throw new IllegalStateException("No such any registry to reference " + interfaceName + " on the consumer " + NetUtils.*getLocalHost*() + " use dubbo version " + Version.*getVersion*() + ", please config <dubbo:registry address=\"...\" /> to your spring config.");  }  }  # 4）只有一个服务中心中心时候  if (urls.size() == 1) {  invoker = *refprotocol*.refer(interfaceClass, urls.get(0));  } else {  # 多个服务中心时候  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; // use last registry url  }  }  if (registryURL != null) { // registry url is available  // use RegistryAwareCluster only when register's cluster is available  URL u = registryURL.addParameter(Constants.*CLUSTER\_KEY*, RegistryAwareCluster.*NAME*);  // The invoker wrap relation would be: RegistryAwareClusterInvoker(StaticDirectory) -> FailoverClusterInvoker(RegistryDirectory, will execute route) -> Invoker  invoker = *cluster*.join(new StaticDirectory(u, invokers));  } else { // not a registry url, must be direct invoke.  invoker = *cluster*.join(new StaticDirectory(invokers));  }  }  }  # 5）是否应该在启动时候检查提供方是否可用  if (shouldCheck() && !invoker.isAvailable()) {  // make it possible for consumer to retry later if provider is temporarily unavailable  initialized = false;  throw new IllegalStateException("Failed to check the status of the service " + interfaceName + ". No provider available for the service " + (group == null ? "" : group + "/") + interfaceName + (version == null ? "" : ":" + version) + " from the url " + invoker.getUrl() + " to the consumer " + NetUtils.*getLocalHost*() + " use dubbo version " + Version.*getVersion*());  }  if (*logger*.isInfoEnabled()) {  *logger*.info("Refer dubbo service " + interfaceClass.getName() + " from url " + invoker.getUrl());  }  */\*\*  \** ***@since*** *2.7.0  \* ServiceData Store  \*/* MetadataReportService metadataReportService = null;  if ((metadataReportService = getMetadataReportService()) != null) {  URL consumerURL = new URL(Constants.*CONSUMER\_PROTOCOL*, map.remove(Constants.*REGISTER\_IP\_KEY*), 0, map.get(Constants.*INTERFACE\_KEY*), map);  metadataReportService.publishConsumer(consumerURL);  }  // create service proxy  # 6）创建服务代理  return (T) *proxyFactory*.getProxy(invoker); } |

### RegistryProtocol#refer(Class<T> type,URL url)

|  |
| --- |
| public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {  url = URLBuilder.*from*(url)  .setProtocol(url.getParameter(*REGISTRY\_KEY*, *DEFAULT\_REGISTRY*))  .removeParameter(*REGISTRY\_KEY*)  .build();  Registry registry = registryFactory.getRegistry(url);  if (RegistryService.class.equals(type)) {  return proxyFactory.getInvoker((T) registry, type, url);  }   // group="a,b" or group="\*"  Map<String, String> qs = StringUtils.*parseQueryString*(url.getParameterAndDecoded(*REFER\_KEY*));  String group = qs.get(Constants.*GROUP\_KEY*);  if (group != null && group.length() > 0) {  if ((*COMMA\_SPLIT\_PATTERN*.split(group)).length > 1 || "\*".equals(group)) {  return doRefer(getMergeableCluster(), registry, type, url);  }  }  return doRefer(cluster, registry, type, url); } |

### RegistryProtocol#doRefer(cluster,registry,type,url)

|  |
| --- |
| 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);  // all attributes of REFER\_KEY  Map<String, String> parameters = new HashMap<String, String>(directory.getUrl().getParameters());  URL subscribeUrl = new URL(*CONSUMER\_PROTOCOL*, parameters.remove(*REGISTER\_IP\_KEY*), 0, type.getName(), parameters);  if (!*ANY\_VALUE*.equals(url.getServiceInterface()) && url.getParameter(*REGISTER\_KEY*, true)) {  directory.setRegisteredConsumerUrl(getRegisteredConsumerUrl(subscribeUrl, url));  registry.register(directory.getRegisteredConsumerUrl());  }  # 1）建立路由规则链  directory.buildRouterChain(subscribeUrl);  # 2）订阅服务提供者地址  directory.subscribe(subscribeUrl.addParameter(*CATEGORY\_KEY*,  *PROVIDERS\_CATEGORY* + "," + *CONFIGURATORS\_CATEGORY* + "," + *ROUTERS\_CATEGORY*));  # 3）包装机器容错策略到invoker  Invoker invoker = cluster.join(directory);  ProviderConsumerRegTable.*registerConsumer*(invoker, url, subscribeUrl, directory);  return invoker; } |

1. ：订阅URL创建路由规则链，
2. ：向服务注册中心订阅提供者的服务，
3. ：调用扩展接口Cluster的适配器类的join（）方法。

### RegistryDirectory#subscribe(URL url)

|  |
| --- |
| public void subscribe(URL url) {  setConsumerUrl(url);  *consumerConfigurationListener*.addNotifyListener(this);  serviceConfigurationListener = new ReferenceConfigurationListener(this, url);  registry.subscribe(url, this); } |

### ZookeeperRegistry#doSubscirbe(URL url,NotifyListener listener)

|  |
| --- |
| public void doSubscribe(final URL url, final NotifyListener listener) {  try {  if (Constants.*ANY\_VALUE*.equals(url.getServiceInterface())) {  String root = toRootPath();  ConcurrentMap<NotifyListener, ChildListener> listeners = zkListeners.get(url);  if (listeners == null) {  zkListeners.putIfAbsent(url, new ConcurrentHashMap<>());  listeners = zkListeners.get(url);  }  ChildListener zkListener = listeners.get(listener);  if (zkListener == null) {  listeners.putIfAbsent(listener, (parentPath, currentChilds) -> {  for (String child : currentChilds) {  child = URL.*decode*(child);  if (!anyServices.contains(child)) {  anyServices.add(child);  subscribe(url.setPath(child).addParameters(Constants.*INTERFACE\_KEY*, child,  Constants.*CHECK\_KEY*, String.*valueOf*(false)), listener);  }  }  });  zkListener = listeners.get(listener);  }  zkClient.create(root, false);  List<String> services = zkClient.addChildListener(root, zkListener);  if (CollectionUtils.*isNotEmpty*(services)) {  for (String service : services) {  service = URL.*decode*(service);  anyServices.add(service);  subscribe(url.setPath(service).addParameters(Constants.*INTERFACE\_KEY*, service,  Constants.*CHECK\_KEY*, String.*valueOf*(false)), listener);  }  }  } else {  List<URL> urls = new ArrayList<>();  for (String path : toCategoriesPath(url)) {  ConcurrentMap<NotifyListener, ChildListener> listeners = zkListeners.get(url);  if (listeners == null) {  zkListeners.putIfAbsent(url, new ConcurrentHashMap<>());  listeners = zkListeners.get(url);  }  ChildListener zkListener = listeners.get(listener);  if (zkListener == null) {  listeners.putIfAbsent(listener, (parentPath, currentChilds) -> ZookeeperRegistry.this.notify(url, listener, toUrlsWithEmpty(url, parentPath, currentChilds)));  zkListener = listeners.get(listener);  }  zkClient.create(path, false);  List<String> children = zkClient.addChildListener(path, zkListener);  if (children != null) {  urls.addAll(toUrlsWithEmpty(url, path, children));  }  }  notify(url, listener, urls);  }  } catch (Throwable e) {  throw new RpcException("Failed to subscribe " + url + " to zookeeper " + getUrl() + ", cause: " + e.getMessage(), e);  } } |

### RegistryDirectory#notify()

|  |
| --- |
| public synchronized void notify(List<URL> urls) {  # 对不同的元数据进行分类  Map<String, List<URL>> categoryUrls = urls.stream()  .filter(Objects::*nonNull*)  .filter(this::isValidCategory)  .filter(this::isNotCompatibleFor26x)  .collect(Collectors.*groupingBy*(url -> {  if (UrlUtils.*isConfigurator*(url)) {  return *CONFIGURATORS\_CATEGORY*;  } else if (UrlUtils.*isRoute*(url)) {  return *ROUTERS\_CATEGORY*;  } else if (UrlUtils.*isProvider*(url)) {  return *PROVIDERS\_CATEGORY*;  }  return "";  }));  # 配置信息，比如服务降级信息  List<URL> configuratorURLs = categoryUrls.getOrDefault(*CONFIGURATORS\_CATEGORY*, Collections.*emptyList*());  this.configurators = Configurator.*toConfigurators*(configuratorURLs).orElse(this.configurators);  # 路由信息收集并保存  List<URL> routerURLs = categoryUrls.getOrDefault(*ROUTERS\_CATEGORY*, Collections.*emptyList*());  toRouters(routerURLs).ifPresent(this::addRouters);   // providers  # 服务提供者信息  List<URL> providerURLs = categoryUrls.getOrDefault(*PROVIDERS\_CATEGORY*, Collections.*emptyList*());  refreshOverrideAndInvoker(providerURLs); } |

### RegistryDirectory#refreshOverrideAndInvoker

|  |
| --- |
| private void refreshOverrideAndInvoker(List<URL> urls) {  // mock zookeeper://xxx?mock=return null  overrideDirectoryUrl();  refreshInvoker(urls); }  private void overrideDirectoryUrl() {  // merge override parameters  this.overrideDirectoryUrl = directoryUrl;  List<Configurator> localConfigurators = this.configurators; // local reference  doOverrideUrl(localConfigurators);  List<Configurator> localAppDynamicConfigurators = *consumerConfigurationListener*.getConfigurators(); // local reference  doOverrideUrl(localAppDynamicConfigurators);  if (serviceConfigurationListener != null) {  List<Configurator> localDynamicConfigurators = serviceConfigurationListener.getConfigurators(); // local reference  doOverrideUrl(localDynamicConfigurators);  } } |

### RegistryDirectory#refreshInvoker

|  |
| --- |
| private void refreshInvoker(List<URL> invokerUrls) {  Assert.*notNull*(invokerUrls, "invokerUrls should not be null");   if (invokerUrls.size() == 1  && invokerUrls.get(0) != null  && Constants.*EMPTY\_PROTOCOL*.equals(invokerUrls.get(0).getProtocol())) {  this.forbidden = true; // Forbid to access  this.invokers = Collections.*emptyList*();  routerChain.setInvokers(this.invokers);  destroyAllInvokers(); // Close all invokers  } else {  this.forbidden = false; // Allow to access  Map<String, Invoker<T>> oldUrlInvokerMap = this.urlInvokerMap; // local reference  if (invokerUrls == Collections.<URL>*emptyList*()) {  invokerUrls = new ArrayList<>();  }  if (invokerUrls.isEmpty() && this.cachedInvokerUrls != null) {  invokerUrls.addAll(this.cachedInvokerUrls);  } else {  this.cachedInvokerUrls = new HashSet<>();  this.cachedInvokerUrls.addAll(invokerUrls);//Cached invoker urls, convenient for comparison  }  if (invokerUrls.isEmpty()) {  return;  }  # 将URL转换成Invoker对象  Map<String, Invoker<T>> newUrlInvokerMap = toInvokers(invokerUrls);// Translate url list to Invoker map   */\*\*  \* If the calculation is wrong, it is not processed.  \*  \* 1. The protocol configured by the client is inconsistent with the protocol of the server.  \* eg: consumer protocol = dubbo, provider only has other protocol services(rest).  \* 2. The registration center is not robust and pushes illegal specification data.  \*  \*/* if (CollectionUtils.*isEmptyMap*(newUrlInvokerMap)) {  *logger*.error(new IllegalStateException("urls to invokers error .invokerUrls.size :" + invokerUrls.size() + ", invoker.size :0. urls :" + invokerUrls  .toString()));  return;  }   List<Invoker<T>> newInvokers = Collections.*unmodifiableList*(new ArrayList<>(newUrlInvokerMap.values()));  // pre-route and build cache, notice that route cache should build on original Invoker list.  // toMergeMethodInvokerMap() will wrap some invokers having different groups, those wrapped invokers not should be routed.  routerChain.setInvokers(newInvokers);  this.invokers = multiGroup ? toMergeInvokerList(newInvokers) : newInvokers;  this.urlInvokerMap = newUrlInvokerMap;   try {  destroyUnusedInvokers(oldUrlInvokerMap, newUrlInvokerMap); // Close the unused Invoker  } catch (Exception e) {  *logger*.warn("destroyUnusedInvokers error. ", e);  }  } } |

### RegistryDirectory#toInvokers

|  |
| --- |
| private Map<String, Invoker<T>> toInvokers(List<URL> urls) {  Map<String, Invoker<T>> newUrlInvokerMap = new HashMap<>();  if (urls == null || urls.isEmpty()) {  return newUrlInvokerMap;  }  Set<String> keys = new HashSet<>();  String queryProtocols = this.queryMap.get(Constants.*PROTOCOL\_KEY*);  for (URL providerUrl : urls) {  // If protocol is configured at the reference side, only the matching protocol is selected  if (queryProtocols != null && queryProtocols.length() > 0) {  boolean accept = false;  String[] acceptProtocols = queryProtocols.split(",");  for (String acceptProtocol : acceptProtocols) {  if (providerUrl.getProtocol().equals(acceptProtocol)) {  accept = true;  break;  }  }  if (!accept) {  continue;  }  }  if (Constants.*EMPTY\_PROTOCOL*.equals(providerUrl.getProtocol())) {  continue;  }  if (!ExtensionLoader.*getExtensionLoader*(Protocol.class).hasExtension(providerUrl.getProtocol())) {  *logger*.error(new IllegalStateException("Unsupported protocol " + providerUrl.getProtocol() +  " in notified url: " + providerUrl + " from registry " + getUrl().getAddress() +  " to consumer " + NetUtils.*getLocalHost*() + ", supported protocol: " +  ExtensionLoader.*getExtensionLoader*(Protocol.class).getSupportedExtensions()));  continue;  }  URL url = mergeUrl(providerUrl);   String key = url.toFullString(); // The parameter urls are sorted  if (keys.contains(key)) { // Repeated url  continue;  }  keys.add(key);  // Cache key is url that does not merge with consumer side parameters, regardless of how the consumer combines parameters, if the server url changes, then refer again  Map<String, Invoker<T>> localUrlInvokerMap = this.urlInvokerMap; // local reference  Invoker<T> invoker = localUrlInvokerMap == null ? null : localUrlInvokerMap.get(key);  if (invoker == null) { // Not in the cache, refer again  try {  boolean enabled = true;  if (url.hasParameter(Constants.*DISABLED\_KEY*)) {  enabled = !url.getParameter(Constants.*DISABLED\_KEY*, false);  } else {  enabled = url.getParameter(Constants.*ENABLED\_KEY*, true);  }  if (enabled) {  #调用dubbo转换服务到Invoker中  #protocol#refer实际上时调用适配器Protocol$Adaptive  #的refer()方法  invoker = new InvokerDelegate<>(protocol.refer(serviceType, url), url, providerUrl);  }  } catch (Throwable t) {  *logger*.error("Failed to refer invoker for interface:" + serviceType + ",url:(" + url + ")" + t.getMessage(), t);  }  if (invoker != null) { // Put new invoker in cache  newUrlInvokerMap.put(key, invoker);  }  } else {  newUrlInvokerMap.put(key, invoker);  }  }  keys.clear();  return newUrlInvokerMap; } |

### DubboProtocol#refer

dubbo使用了ProtocolListenerWrapper、ProtocolFilterWrapper等类对DubboProtocol进行了功能增强。

|  |
| --- |
| @Override public <T> Invoker<T> refer(Class<T> serviceType, URL url) throws RpcException {  optimizeSerialization(url);   // create rpc invoker.  DubboInvoker<T> invoker = new DubboInvoker<T>(serviceType, url, getClients(url), invokers);  invokers.add(invoker);   return invoker; }  private ExchangeClient[] getClients(URL url) {  // whether to share connection  // 不同服务是否共享连接  boolean useShareConnect = false;   int connections = url.getParameter(Constants.*CONNECTIONS\_KEY*, 0);  List<ReferenceCountExchangeClient> shareClients = null;  // if not configured, connection is shared, otherwise, one connection for one service  // 1）如果没有配置，则默认连接是共享的，否则每个服务单独有自己的连接  if (connections == 0) {  useShareConnect = true;   */\*\*  \* The xml configuration should have a higher priority than properties.  \*/*  *// 2.1）xml配置优先级高于属性配置* String shareConnectionsStr = url.getParameter(Constants.*SHARE\_CONNECTIONS\_KEY*, (String) null);  connections = Integer.*parseInt*(StringUtils.*isBlank*(shareConnectionsStr) ? ConfigUtils.*getProperty*(Constants.*SHARE\_CONNECTIONS\_KEY*,  Constants.*DEFAULT\_SHARE\_CONNECTIONS*) : shareConnectionsStr);  // 2.2）获取共享NettyClient  shareClients = getSharedClient(url, connections);  }  // 初始化Client  ExchangeClient[] clients = new ExchangeClient[connections];  for (int i = 0; i < clients.length; i++) {  if (useShareConnect) {// 3.1)共享则返回已经存在的  clients[i] = shareClients.get(i);   } else {  // 3.2）否则创建  clients[i] = initClient(url);  }  }   return clients; }  private ExchangeClient initClient(URL url) {   // client type setting.  String str = url.getParameter(Constants.*CLIENT\_KEY*, url.getParameter(Constants.*SERVER\_KEY*, Constants.*DEFAULT\_REMOTING\_CLIENT*));   url = url.addParameter(Constants.*CODEC\_KEY*, DubboCodec.*NAME*);  // enable heartbeat by default  url = url.addParameterIfAbsent(Constants.*HEARTBEAT\_KEY*, String.*valueOf*(Constants.*DEFAULT\_HEARTBEAT*));   // BIO is not allowed since it has severe performance issue.  if (str != null && str.length() > 0 && !ExtensionLoader.*getExtensionLoader*(Transporter.class).hasExtension(str)) {  throw new RpcException("Unsupported client type: " + str + "," +  " supported client type is " + StringUtils.*join*(ExtensionLoader.*getExtensionLoader*(Transporter.class).getSupportedExtensions(), " "));  }   ExchangeClient client;  try {  // connection should be lazy  // 1）惰性连接  if (url.getParameter(Constants.*LAZY\_CONNECT\_KEY*, false)) {  client = new LazyConnectExchangeClient(url, requestHandler);   } else {  // 2）为及时连接  client = Exchangers.*connect*(url, requestHandler);  }   } catch (RemotingException e) {  throw new RpcException("Fail to create remoting client for service(" + url + "): " + e.getMessage(), e);  }   return client; } |

### NettyClient#doOpen

|  |
| --- |
| public NettyClient(final URL url, final ChannelHandler handler) throws RemotingException {  super(url, *wrapChannelHandler*(url, handler)); }  // 加入线程池  protected static ChannelHandler wrapChannelHandler(URL url, ChannelHandler handler) {  url = ExecutorUtil.*setThreadName*(url, *CLIENT\_THREAD\_POOL\_NAME*);  url = url.addParameterIfAbsent(Constants.*THREADPOOL\_KEY*, Constants.*DEFAULT\_CLIENT\_THREADPOOL*);  return ChannelHandlers.*wrap*(handler, url); }  protected void doOpen() throws Throwable {  NettyHelper.*setNettyLoggerFactory*();  bootstrap = new ClientBootstrap(*channelFactory*);  // config  // @see org.jboss.netty.channel.socket.SocketChannelConfig  bootstrap.setOption("keepAlive", true);  bootstrap.setOption("tcpNoDelay", true);  bootstrap.setOption("connectTimeoutMillis", getConnectTimeout());  final NettyHandler nettyHandler = new NettyHandler(getUrl(), this);  bootstrap.setPipelineFactory(new ChannelPipelineFactory() {  @Override  public ChannelPipeline getPipeline() {  NettyCodecAdapter adapter = new NettyCodecAdapter(getCodec(), getUrl(), NettyClient.this);  ChannelPipeline pipeline = Channels.*pipeline*();  pipeline.addLast("decoder", adapter.getDecoder());  pipeline.addLast("encoder", adapter.getEncoder());  pipeline.addLast("handler", nettyHandler);  return pipeline;  }  }); } |

### NettyClient#doConnect

|  |
| --- |
| protected void doConnect() throws Throwable {  long start = System.*currentTimeMillis*();  ChannelFuture future = bootstrap.connect(getConnectAddress());  try {  boolean ret = future.awaitUninterruptibly(getConnectTimeout(), *MILLISECONDS*);   if (ret && future.isSuccess()) {  Channel newChannel = future.channel();  try {  // Close old channel  Channel oldChannel = NettyClient.this.channel; // copy reference  if (oldChannel != null) {  try {  if (*logger*.isInfoEnabled()) {  *logger*.info("Close old netty channel " + oldChannel + " on create new netty channel " + newChannel);  }  oldChannel.close();  } finally {  NettyChannel.*removeChannelIfDisconnected*(oldChannel);  }  }  } finally {  if (NettyClient.this.isClosed()) {  try {  if (*logger*.isInfoEnabled()) {  *logger*.info("Close new netty channel " + newChannel + ", because the client closed.");  }  newChannel.close();  } finally {  NettyClient.this.channel = null;  NettyChannel.*removeChannelIfDisconnected*(newChannel);  }  } else {  NettyClient.this.channel = newChannel;  }  }  } else if (future.cause() != null) {  throw new RemotingException(this, "client(url: " + getUrl() + ") failed to connect to server "  + getRemoteAddress() + ", error message is:" + future.cause().getMessage(), future.cause());  } else {  throw new RemotingException(this, "client(url: " + getUrl() + ") failed to connect to server "  + getRemoteAddress() + " client-side timeout "  + getConnectTimeout() + "ms (elapsed: " + (System.*currentTimeMillis*() - start) + "ms) from netty client "  + NetUtils.*getLocalHost*() + " using dubbo version " + Version.*getVersion*());  }  } finally {  if (!isConnected()) {  //future.cancel(true);  }  } } |

### ChannelHandlers#wrap

|  |
| --- |
| // 确定线程池模型  public static ChannelHandler wrap(ChannelHandler handler, URL url) {  return ChannelHandlers.*getInstance*().wrapInternal(handler, url); }  protected ChannelHandler wrapInternal(ChannelHandler handler, URL url) {  return new MultiMessageHandler(new HeartbeatHandler(ExtensionLoader.*getExtensionLoader*(Dispatcher.class)  .getAdaptiveExtension().dispatch(handler, url))); } |

### AbstractClient

|  |
| --- |
| public AbstractClient(URL url, ChannelHandler handler) throws RemotingException {  // 加载编解码器  super(url, handler);   needReconnect = url.getParameter(Constants.*SEND\_RECONNECT\_KEY*, false);   try {  // 调用子类的doOpen  doOpen();  } catch (Throwable t) {  close();  throw new RemotingException(url.toInetSocketAddress(), null,  "Failed to start " + getClass().getSimpleName() + " " + NetUtils.*getLocalAddress*()  + " connect to the server " + getRemoteAddress() + ", cause: " + t.getMessage(), t);  }  try {  // connect.  // 发送远端连接  connect();  if (*logger*.isInfoEnabled()) {  *logger*.info("Start " + getClass().getSimpleName() + " " + NetUtils.*getLocalAddress*() + " connect to the server " + getRemoteAddress());  }  } catch (RemotingException t) {  if (url.getParameter(Constants.*CHECK\_KEY*, true)) {  close();  throw t;  } else {  *logger*.warn("Failed to start " + getClass().getSimpleName() + " " + NetUtils.*getLocalAddress*()  + " connect to the server " + getRemoteAddress() + " (check == false, ignore and retry later!), cause: " + t.getMessage(), t);  }  } catch (Throwable t) {  close();  throw new RemotingException(url.toInetSocketAddress(), null,  "Failed to start " + getClass().getSimpleName() + " " + NetUtils.*getLocalAddress*()  + " connect to the server " + getRemoteAddress() + ", cause: " + t.getMessage(), t);  }   executor = (ExecutorService) ExtensionLoader.*getExtensionLoader*(DataStore.class)  .getDefaultExtension().get(Constants.*CONSUMER\_SIDE*, Integer.*toString*(url.getPort()));  ExtensionLoader.*getExtensionLoader*(DataStore.class)  .getDefaultExtension().remove(Constants.*CONSUMER\_SIDE*, Integer.*toString*(url.getPort())); } |

### ProtocolFilterFilterWrapper#buildInvokerChain()

|  |
| --- |
| private static <T> Invoker<T> buildInvokerChain(final Invoker<T> invoker, String key, String group) {  Invoker<T> last = invoker;  // 获取所有激活的Filter，然后使用链表方式形成责任链  List<Filter> filters = ExtensionLoader.*getExtensionLoader*(Filter.class).getActivateExtension(invoker.getUrl(), key, group);  if (!filters.isEmpty()) {  for (int i = filters.size() - 1; i >= 0; i--) {  final Filter filter = filters.get(i);  final Invoker<T> next = last;  last = new Invoker<T>() {   @Override  public Class<T> getInterface() {  return invoker.getInterface();  }   @Override  public URL getUrl() {  return invoker.getUrl();  }   @Override  public boolean isAvailable() {  return invoker.isAvailable();  }   @Override  public Result invoke(Invocation invocation) throws RpcException {  Result result = filter.invoke(next, invocation);  // 支持异步调用  if (result instanceof AsyncRpcResult) {  AsyncRpcResult asyncResult = (AsyncRpcResult) result;  asyncResult.thenApplyWithContext(r -> filter.onResponse(r, invoker, invocation));  return asyncResult;  } else {  return filter.onResponse(result, invoker, invocation);  }  }   @Override  public void destroy() {  invoker.destroy();  }   @Override  public String toString() {  return invoker.toString();  }  };  }  }  return last; } |

### FailoverCluster#join

|  |
| --- |
| @Override public <T> Invoker<T> join(Directory<T> directory) throws RpcException {  return new FailoverClusterInvoker<T>(directory); } |

### MockClusterWrapper

|  |
| --- |
| public class MockClusterWrapper implements Cluster {   private Cluster cluster;   public MockClusterWrapper(Cluster cluster) {  this.cluster = cluster;  }   @Override  public <T> Invoker<T> join(Directory<T> directory) throws RpcException {  return new MockClusterInvoker<T>(directory,  this.cluster.join(directory));  }  } |

## 3.4 Dubbo服务消费端调用一次远程调用过程

### InvokerInvocationHandler

|  |
| --- |
| public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  String methodName = method.getName();  Class<?>[] parameterTypes = method.getParameterTypes();  if (method.getDeclaringClass() == Object.class) {  return method.invoke(invoker, args);  }  if ("toString".equals(methodName) && parameterTypes.length == 0) {  return invoker.toString();  }  if ("hashCode".equals(methodName) && parameterTypes.length == 0) {  return invoker.hashCode();  }  if ("equals".equals(methodName) && parameterTypes.length == 1) {  return invoker.equals(args[0]);  }   return invoker.invoke(createInvocation(method, args)).recreate(); }  private RpcInvocation createInvocation(Method method, Object[] args) {  RpcInvocation invocation = new RpcInvocation(method, args);  if (RpcUtils.*hasFutureReturnType*(method)) {  invocation.setAttachment(Constants.*FUTURE\_RETURNTYPE\_KEY*, "true");  invocation.setAttachment(Constants.*ASYNC\_KEY*, "true");  }  return invocation; } |

默认的集群容错策略FailoverClusterInvoker，其内部首先根绝设置的负载均衡策略LoadBalance的扩展实现。

### DubboInvoker

|  |
| --- |
| protected Result doInvoke(final Invocation invocation) throws Throwable {  // 1）设置附加属性  RpcInvocation inv = (RpcInvocation) invocation;  final String methodName = RpcUtils.*getMethodName*(invocation);  inv.setAttachment(Constants.*PATH\_KEY*, getUrl().getPath());  inv.setAttachment(Constants.*VERSION\_KEY*, version);  // 2）获取远程调用Client  ExchangeClient currentClient;  if (clients.length == 1) {  currentClient = clients[0];  } else {  currentClient = clients[index.getAndIncrement() % clients.length];  }  // 3）执行远程调用  try {  // 3.1）是否为异步调用  boolean isAsync = RpcUtils.*isAsync*(getUrl(), invocation);  // 3.2）是否为future方式异步  boolean isAsyncFuture = RpcUtils.*isReturnTypeFuture*(inv);  // 3.3）是否为oneway，也就是不需要响应结果的请求  boolean isOneway = RpcUtils.*isOneway*(getUrl(), invocation);  // 3.4）超时等待时间  int timeout = getUrl().getMethodParameter(methodName, Constants.*TIMEOUT\_KEY*, Constants.*DEFAULT\_TIMEOUT*);  // 3.5）不需要响应的请求  if (isOneway) {  boolean isSent = getUrl().getMethodParameter(methodName, Constants.*SENT\_KEY*, false);  currentClient.send(inv, isSent);  RpcContext.*getContext*().setFuture(null);  return new RpcResult();  } else if (isAsync) {  // 3.6）异步请求  ResponseFuture future = currentClient.request(inv, timeout);  // For compatibility  FutureAdapter<Object> futureAdapter = new FutureAdapter<>(future);  RpcContext.*getContext*().setFuture(futureAdapter);  // 3.6.1）异步请求  Result result;  if (isAsyncFuture) {  // register resultCallback, sometimes we need the async result being processed by the filter chain.  result = new AsyncRpcResult(futureAdapter, futureAdapter.getResultFuture(), false);  } else {  result = new SimpleAsyncRpcResult(futureAdapter, futureAdapter.getResultFuture(), false);  }  return result;  } else {  // 3.7）同步请求  RpcContext.*getContext*().setFuture(null);  return (Result) currentClient.request(inv, timeout).get();  }  } catch (TimeoutException e) {  throw new RpcException(RpcException.*TIMEOUT\_EXCEPTION*, "Invoke remote method timeout. method: " + invocation.getMethodName() + ", provider: " + getUrl() + ", cause: " + e.getMessage(), e);  } catch (RemotingException e) {  throw new RpcException(RpcException.*NETWORK\_EXCEPTION*, "Failed to invoke remote method: " + invocation.getMethodName() + ", provider: " + getUrl() + ", cause: " + e.getMessage(), e);  } } |

# Directory目录与Router路由服务

### 4.1 Directory目录

Directory代表了多个invoker，每个invoker代表了一个服务提供者。

在Directory的实现有RegistryDirectory和StaticDirectory两种，其中前者管理invoker列表是根据服务注册中心的托送变化而变化的。

### 4.2 RegistryDirectory的创建

创建步骤：

|  |  |
| --- | --- |
| ReferenceConfig | 1. get:获得代理 2. init：初始化过程 3. createProxy：创建代理 |
| Procotol$Adaptive | 4.refer |
| QosProtocolWrapper | 5.refer |
| ProtocolListenerWrapper | 6.refer |
| ProtocolFilterWrapper | 7.refer |
| RegistyProtocol | 1. refer 2. doRefer 3. buildRouterChain |
| RegistryDirectory | 11.subscribe |
| Cluster$Adaptive | 1. join 2. join |
| ReferenceConfig | 14.getProxy |
|  |  |

### 4.2 RegistryDirectory中invoker列表的更新

|  |  |
| --- | --- |
| RegistryDirectory | 1.subscribe：调用ZookeeperRegistry#subscribe，会调用ZookeeperRegistry#doSubscribe |
| ZookeeperRegistry | 1. doSubscribe：调用CuratorZookeeperClient#addChildListener添加Zookeeper事件监听器 2. addChildListener：会调用notify，执行doNotify方法 3. notify:没看到调用RegistryDirectory#notify |
| RegistryDirectory | 1. notify 2. addRouters |
| RouterChain | 7.addRouters：将路由规则信息保存在RouterChain |
| RegistryDirectory | 1. refershOveriideAndInvoker:刷新服务提供者列表 2. overrideDirectoryUrl：根据服务降级信息，重写URL 3. refershInvoker：刷新invoker列表 4. toInvokers：转换URL为invoker |
| RouterChain | 12.setInvokers： |
|  |  |
|  |  |
|  |  |
|  |  |

# dubbo消费端服务mock与服务降级策略原理

## 5.1 服务降级原理

### 5.1.1 降级策略注册

当服务消费者启动时，会去订阅com.books.dubbo.demo.api.GreetingService子树中信息，比如Providers（服务提供者列表）、Routes（路由信息）、Configurators（服务降级策略）等信息，

### 5.1.2 服务消费端使用降级策略

|  |  |
| --- | --- |
| APIConsumer | 1. main |
| proxy0 | 2.sayHello |
| InvokerInvocationHandler | 3.invoke |
| MockClusterInvoker | 4.invoke |
| FailoverClusterInvoker | 1. invoke 2. list |
| RegistryDirectory | 1. list 2. doList |
| RouterChain | 9.route |
| DubboInvoker | 10.invoke:在FailoverClusterInvoker调用 |

MockClusterInvoker

|  |
| --- |
| public Result invoke(Invocation invocation) throws RpcException {  Result result = null;  // 1）查看URL里面是否有mock字段  String value = directory.getUrl().getMethodParameter(invocation.getMethodName(), Constants.*MOCK\_KEY*, Boolean.*FALSE*.toString()).trim();  // 2）如果没有，或者值为默认的false，则说明没有设置降级策略  if (value.length() == 0 || value.equalsIgnoreCase("false")) {  //no mock  // 2.1）没有mock，正常发起远程调用  result = this.invoker.invoke(invocation);  } else if (value.startsWith("force")) {  // 3.1)force:direct mock  if (*logger*.isWarnEnabled()) {  *logger*.warn("force-mock: " + invocation.getMethodName() + " force-mock enabled , url : " + directory.getUrl());  }  //force:direct mock  result = doMockInvoke(invocation, null);  } else {  //fail-mock  // 设置fail-mock  try {  result = this.invoker.invoke(invocation);  } catch (RpcException e) {  if (e.isBiz()) {  throw e;  }    if (*logger*.isWarnEnabled()) {  *logger*.warn("fail-mock: " + invocation.getMethodName() + " fail-mock enabled , url : " + directory.getUrl(), e);  }  result = doMockInvoke(invocation, e);  }  }  return result; } |

## 5.2 本地服务mock原理

### 5.2.1 mock合法性检查

ReferenceConfig#checkMock

|  |
| --- |
| void checkMock(Class<?> interfaceClass) {  // 1）没有设置mock，则直接返回  if (ConfigUtils.*isEmpty*(mock)) {  return;  }  // 2）获取格式化的mock方式  String normalizedMock = MockInvoker.*normalizeMock*(mock);  // 3）检查mock值是否合法，不合法则抛出异常  if (normalizedMock.startsWith(Constants.*RETURN\_PREFIX*)) {  normalizedMock = normalizedMock.substring(Constants.*RETURN\_PREFIX*.length()).trim();  try {  //Check whether the mock value is legal, if it is illegal, throw exception  MockInvoker.*parseMockValue*(normalizedMock);  } catch (Exception e) {  throw new IllegalStateException("Illegal mock return in <dubbo:service/reference ... " +  "mock=\"" + mock + "\" />");  }  } else if (normalizedMock.startsWith(Constants.*THROW\_PREFIX*)) {  normalizedMock = normalizedMock.substring(Constants.*THROW\_PREFIX*.length()).trim();  if (ConfigUtils.*isNotEmpty*(normalizedMock)) {  try {  //Check whether the mock value is legal  MockInvoker.*getThrowable*(normalizedMock);  } catch (Exception e) {  throw new IllegalStateException("Illegal mock throw in <dubbo:service/reference ... " +  "mock=\"" + mock + "\" />");  }  }  } else {  //Check whether the mock class is a implementation of the interfaceClass, and if it has a default constructor  // 4）检查mock接口的实现类是否符合规则  MockInvoker.*getMockObject*(normalizedMock, interfaceClass);  } } |

MockInvoker#getMockObject

|  |
| --- |
| public static Object getMockObject(String mockService, Class serviceType) {  // 5）如果mock类型为true或者default  if (ConfigUtils.*isDefault*(mockService)) {  mockService = serviceType.getName() + "Mock";  }  // 6）反射加载字节码创建Class对象  Class<?> mockClass = ReflectUtils.*forName*(mockService);  if (!serviceType.isAssignableFrom(mockClass)) {  throw new IllegalStateException("The mock class " + mockClass.getName() +  " not implement interface " + serviceType.getName());  }  // 7）创建实例  try {  return mockClass.newInstance();  } catch (InstantiationException e) {  throw new IllegalStateException("No default constructor from mock class " + mockClass.getName(), e);  } catch (IllegalAccessException e) {  throw new IllegalStateException(e);  } } |

### 5.2.2 服务消费端使用mock服务

MockClusterInvoker#invoke

|  |
| --- |
| public Result invoke(Invocation invocation) throws RpcException {  Result result = null;   String value = directory.getUrl().getMethodParameter(invocation.getMethodName(), Constants.*MOCK\_KEY*, Boolean.*FALSE*.toString()).trim();  // 1）没有mock  if (value.length() == 0 || value.equalsIgnoreCase("false")) {  //no mock  // 没有mock  result = this.invoker.invoke(invocation);  // 2）force  } else if (value.startsWith("force")) {  if (*logger*.isWarnEnabled()) {  *logger*.warn("force-mock: " + invocation.getMethodName() + " force-mock enabled , url : " + directory.getUrl());  }  //force:direct mock  result = doMockInvoke(invocation, null);  } else {  //fail-mock  //3）fail-mock或mock服务  try {  result = this.invoker.invoke(invocation);  } catch (RpcException e) {  if (e.isBiz()) {  throw e;  }    if (*logger*.isWarnEnabled()) {  *logger*.warn("fail-mock: " + invocation.getMethodName() + " fail-mock enabled , url : " + directory.getUrl(), e);  }  result = doMockInvoke(invocation, e);  }  }  return result; }  // doMockInvoker内部会创建一个MockInvoker对象，然后调用其invoker对象  private Result doMockInvoke(Invocation invocation, RpcException e) {  Result result = null;  Invoker<T> minvoker;   List<Invoker<T>> mockInvokers = selectMockInvoker(invocation);  if (CollectionUtils.*isEmpty*(mockInvokers)) {  minvoker = (Invoker<T>) new MockInvoker(directory.getUrl());  } else {  minvoker = mockInvokers.get(0);  }  try {  result = minvoker.invoke(invocation);  } catch (RpcException me) {  if (me.isBiz()) {  result = new RpcResult(me.getCause());  } else {  throw new RpcException(me.getCode(), getMockExceptionMessage(e, me), me.getCause());  }  } catch (Throwable me) {  throw new RpcException(getMockExceptionMessage(e, me), me.getCause());  }  return result; } |

MockInvoker#invoke

|  |
| --- |
| public Result invoke(Invocation invocation) throws RpcException {  // 4）mock类型  String mock = getUrl().getParameter(invocation.getMethodName() + "." + Constants.*MOCK\_KEY*);  if (invocation instanceof RpcInvocation) {  ((RpcInvocation) invocation).setInvoker(this);  }  if (StringUtils.*isBlank*(mock)) {  mock = getUrl().getParameter(Constants.*MOCK\_KEY*);  }   if (StringUtils.*isBlank*(mock)) {  throw new RpcException(new IllegalAccessException("mock can not be null. url :" + url));  }  // 5）格式化类型  mock = *normalizeMock*(URL.*decode*(mock));  // 6）根据不同类型，返回mock值  if (mock.startsWith(Constants.*RETURN\_PREFIX*)) {  mock = mock.substring(Constants.*RETURN\_PREFIX*.length()).trim();  try {  Type[] returnTypes = RpcUtils.*getReturnTypes*(invocation);  Object value = *parseMockValue*(mock, returnTypes);  return new RpcResult(value);  } catch (Exception ew) {  throw new RpcException("mock return invoke error. method :" + invocation.getMethodName()  + ", mock:" + mock + ", url: " + url, ew);  }  } else if (mock.startsWith(Constants.*THROW\_PREFIX*)) {  mock = mock.substring(Constants.*THROW\_PREFIX*.length()).trim();  if (StringUtils.*isBlank*(mock)) {  throw new RpcException("mocked exception for service degradation.");  } else { // user customized class  Throwable t = *getThrowable*(mock);  throw new RpcException(RpcException.*BIZ\_EXCEPTION*, t);  }  } else { //impl mock  // 7）impl mock  try {  Invoker<T> invoker = getInvoker(mock);  return invoker.invoke(invocation);  } catch (Throwable t) {  throw new RpcException("Failed to create mock implementation class " + mock, t);  }  } }  private Invoker<T> getInvoker(String mockService) {  // 8）缓存存在返回  Invoker<T> invoker = (Invoker<T>) *mocks*.get(mockService);  if (invoker != null) {  return invoker;  }  // 9）不存在则创建代理，并缓存  Class<T> serviceType = (Class<T>) ReflectUtils.*forName*(url.getServiceInterface());  T mockObject = (T) *getMockObject*(mockService, serviceType);  invoker = *proxyFactory*.getInvoker(mockObject, serviceType, url);  if (*mocks*.size() < 10000) {  *mocks*.put(mockService, invoker);  }  return invoker; } |

# dubbo集群容错与负载均衡