# 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

集群容错FailoverClusterInvoker内部首先会调用父类AbstractClusterInvoker的list()方法来获取invoker列表。即从RegistryDirectory管理的RouterChain的route()方法中获取保存的invoker列表。

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

## 6.2 Failfast Cluster策略源码分析

|  |
| --- |
| public Result doInvoke(Invocation invocation, List<Invoker<T>> invokers, LoadBalance loadbalance) throws RpcException {  checkInvokers(invokers, invocation);  // 使用负载均衡策略选择一个服务提供者  Invoker<T> invoker = select(loadbalance, invocation, invokers, null);  try {  return invoker.invoke(invocation);  } catch (Throwable e) {  // 出错则抛出异常  if (e instanceof RpcException && ((RpcException) e).isBiz()) { // biz exception.  throw (RpcException) e;  }  throw new RpcException(e instanceof RpcException ? ((RpcException) e).getCode() : 0,  "Failfast invoke providers " + invoker.getUrl() + " " + loadbalance.getClass().getSimpleName()  + " select from all providers " + invokers + " for service " + getInterface().getName()  + " method " + invocation.getMethodName() + " on consumer " + NetUtils.*getLocalHost*()  + " use dubbo version " + Version.*getVersion*()  + ", but no luck to perform the invocation. Last error is: " + e.getMessage(),  e.getCause() != null ? e.getCause() : e);  } } |

## 6.3 Failsafe Cluster策略源码分析

|  |
| --- |
| public Result doInvoke(Invocation invocation, List<Invoker<T>> invokers, LoadBalance loadbalance) throws RpcException {  try {  checkInvokers(invokers, invocation);  // 使用负载均衡策略选择一个服务提供者  Invoker<T> invoker = select(loadbalance, invocation, invokers, null);  // 执行远程调用  return invoker.invoke(invocation);  } catch (Throwable e) {  *logger*.error("Failsafe ignore exception: " + e.getMessage(), e);  return new RpcResult(); // ignore  } } |

## 6.4 Failover Cluster策略源码分析

|  |
| --- |
| public Result doInvoke(Invocation invocation, final List<Invoker<T>> invokers, LoadBalance loadbalance) throws RpcException {  // 1）所有服务提供者  List<Invoker<T>> copyInvokers = invokers;  checkInvokers(copyInvokers, invocation);  // 2）获取重试次数  String methodName = RpcUtils.*getMethodName*(invocation);  int len = getUrl().getMethodParameter(methodName, Constants.*RETRIES\_KEY*, Constants.*DEFAULT\_RETRIES*) + 1;  if (len <= 0) {  len = 1;  }  // retry loop.  // 3）使用循环，失败重试  RpcException le = null; // last exception.  List<Invoker<T>> invoked = new ArrayList<Invoker<T>>(copyInvokers.size()); // invoked invokers.  Set<String> providers = new HashSet<String>(len);  for (int i = 0; i < len; i++) {  //Reselect before retry to avoid a change of candidate `invokers`.  //NOTE: if `invokers` changed, then `invoked` also lose accuracy.  // 重试时，进行重新选择，避免重试时invoker列表发生变化。  // 注意：如果列表发生了变化，那么invoker判断会失效，因为invoker实例已经改变  if (i > 0) {  // 3.1)  checkWhetherDestroyed();  // 3.2）重新获取所有服务提供者  copyInvokers = list(invocation);  // check again  // 3.3)重新检查一下  checkInvokers(copyInvokers, invocation);  }  // 3.4)选择负载均衡策略  Invoker<T> invoker = select(loadbalance, invocation, copyInvokers, invoked);  invoked.add(invoker);  RpcContext.*getContext*().setInvokers((List) invoked);  try {  // 3.5)具体发起远程调用  Result result = invoker.invoke(invocation);  if (le != null && *logger*.isWarnEnabled()) {  *logger*.warn("Although retry the method " + methodName  + " in the service " + getInterface().getName()  + " was successful by the provider " + invoker.getUrl().getAddress()  + ", but there have been failed providers " + providers  + " (" + providers.size() + "/" + copyInvokers.size()  + ") from the registry " + directory.getUrl().getAddress()  + " on the consumer " + NetUtils.*getLocalHost*()  + " using the dubbo version " + Version.*getVersion*() + ". Last error is: "  + le.getMessage(), le);  }  return result;  } catch (RpcException e) {  if (e.isBiz()) { // biz exception.  throw e;  }  le = e;  } catch (Throwable e) {  le = new RpcException(e.getMessage(), e);  } finally {  providers.add(invoker.getUrl().getAddress());  }  }  throw new RpcException(le.getCode(), "Failed to invoke the method "  + methodName + " in the service " + getInterface().getName()  + ". Tried " + len + " times of the providers " + providers  + " (" + providers.size() + "/" + copyInvokers.size()  + ") from the registry " + directory.getUrl().getAddress()  + " on the consumer " + NetUtils.*getLocalHost*() + " using the dubbo version "  + Version.*getVersion*() + ". Last error is: "  + le.getMessage(), le.getCause() != null ? le.getCause() : le); } |

## 6.5 Failback Cluster策略源码分析

|  |
| --- |
| protected Result doInvoke(Invocation invocation, List<Invoker<T>> invokers, LoadBalance loadbalance) throws RpcException {  Invoker<T> invoker = null;  try {  checkInvokers(invokers, invocation);  invoker = select(loadbalance, invocation, invokers, null);  return invoker.invoke(invocation);  } catch (Throwable e) {  *logger*.error("Failback to invoke method " + invocation.getMethodName() + ", wait for retry in background. Ignored exception: "  + e.getMessage() + ", ", e);  // 失败添加到定时器  addFailed(loadbalance, invocation, invokers, invoker);  return new RpcResult(); // ignore  } }  private void addFailed(LoadBalance loadbalance, Invocation invocation, List<Invoker<T>> invokers, Invoker<T> lastInvoker) {  // 4）创建定时器实例  if (failTimer == null) {  synchronized (this) {  if (failTimer == null) {  failTimer = new HashedWheelTimer(  new NamedThreadFactory("failback-cluster-timer", true),  1,  TimeUnit.*SECONDS*, 32, failbackTasks);  }  }  }  // 5）创建重试任务，并启动  RetryTimerTask retryTimerTask = new RetryTimerTask(loadbalance, invocation, invokers, lastInvoker, retries, *RETRY\_FAILED\_PERIOD*);  try {  failTimer.newTimeout(retryTimerTask, *RETRY\_FAILED\_PERIOD*, TimeUnit.*SECONDS*);  } catch (Throwable e) {  *logger*.error("Failback background works error,invocation->" + invocation + ", exception: " + e.getMessage());  } }  public void run(Timeout timeout) {  try {  // 负载均衡选择器  Invoker<T> retryInvoker = select(loadbalance, invocation, invokers, Collections.*singletonList*(lastInvoker));  // 发起远端调用  lastInvoker = retryInvoker;  retryInvoker.invoke(invocation);  } catch (Throwable e) {  *logger*.error("Failed retry to invoke method " + invocation.getMethodName() + ", waiting again.", e);  // 失败次数达到重试次数则不再重试  if ((++retryTimes) >= retries) {  *logger*.error("Failed retry times exceed threshold (" + retries + "), We have to abandon, invocation->" + invocation);  } else {  // 否则再次重试  rePut(timeout);  }  } }  private void rePut(Timeout timeout) {  if (timeout == null) {  return;  }   Timer timer = timeout.timer();  if (timer.isStop() || timeout.isCancelled()) {  return;  }  // 再次重试  timer.newTimeout(timeout.task(), tick, TimeUnit.*SECONDS*); } |

## 6.6 Forking Cluster策略源码分析

|  |
| --- |
| // 线程池  private final ExecutorService executor = Executors.*newCachedThreadPool*(  new NamedInternalThreadFactory("forking-cluster-timer", true));  public Result doInvoke(final Invocation invocation, List<Invoker<T>> invokers, LoadBalance loadbalance) throws RpcException {  try {  checkInvokers(invokers, invocation);  final List<Invoker<T>> selected;  // 1）获取配置参数  final int forks = getUrl().getParameter(Constants.*FORKS\_KEY*, Constants.*DEFAULT\_FORKS*);  final int timeout = getUrl().getParameter(Constants.*TIMEOUT\_KEY*, Constants.*DEFAULT\_TIMEOUT*);  // 2）获取并行执行的invoker列表  if (forks <= 0 || forks >= invokers.size()) {  selected = invokers;  } else {  selected = new ArrayList<>();  for (int i = 0; i < forks; i++) {  // TODO. Add some comment here, refer chinese version for more details.  // 在invoker列表（排除selected）后，如果没有选够，则存在重  // 复循环问题  Invoker<T> invoker = select(loadbalance, invocation, invokers, selected);  if (!selected.contains(invoker)) {  // 防止重复添加invoker  //Avoid add the same invoker several times.  selected.add(invoker);  }  }  }  // 3）使用线程池让invoker列表中的invoker并发执行  RpcContext.*getContext*().setInvokers((List) selected);  final AtomicInteger count = new AtomicInteger();  final BlockingQueue<Object> ref = new LinkedBlockingQueue<>();  for (final Invoker<T> invoker : selected) {  executor.execute(new Runnable() {  @Override  public void run() {  try {  Result result = invoker.invoke(invocation);  ref.offer(result);  } catch (Throwable e) {  // 3.1）所有的invoker都调用失败则记录错误  int value = count.incrementAndGet();  if (value >= selected.size()) {  ref.offer(e);  }  }  }  });  }  try {  // 4.1）获取执行结果并返回  Object ret = ref.poll(timeout, TimeUnit.*MILLISECONDS*);  if (ret instanceof Throwable) {  Throwable e = (Throwable) ret;  throw new RpcException(e instanceof RpcException ? ((RpcException) e).getCode() : 0, "Failed to forking invoke provider " + selected + ", but no luck to perform the invocation. Last error is: " + e.getMessage(), e.getCause() != null ? e.getCause() : e);  }  return (Result) ret;  } catch (InterruptedException e) {  throw new RpcException("Failed to forking invoke provider " + selected + ", but no luck to perform the invocation. Last error is: " + e.getMessage(), e);  }  } finally {  // clear attachments which is binding to current thread.  RpcContext.*getContext*().clearAttachments();  } } |

## 6.7 Broadcast Cluster策略源码分析

实现广播调用的是BroadcastClusterInvoker类。

|  |
| --- |
| public Result doInvoke(final Invocation invocation, List<Invoker<T>> invokers, LoadBalance loadbalance) throws RpcException {  checkInvokers(invokers, invocation);  RpcContext.*getContext*().setInvokers((List) invokers);  RpcException exception = null;  Result result = null;  for (Invoker<T> invoker : invokers) {  // 使用循环，轮询每个机器进行调用，其中result为最后一个机器的结果  try {  result = invoker.invoke(invocation);  } catch (RpcException e) {  exception = e;  *logger*.warn(e.getMessage(), e);  } catch (Throwable e) {  exception = new RpcException(e.getMessage(), e);  *logger*.warn(e.getMessage(), e);  }  }  if (exception != null) {  throw exception;  }  return result; } |

## 6.8 扩展接口自定义集群容错策略

|  |
| --- |
| package com.books.dubbo.cluster;  import org.apache.dubbo.rpc.\*; import org.apache.dubbo.rpc.cluster.Directory; import org.apache.dubbo.rpc.cluster.LoadBalance; import org.apache.dubbo.rpc.cluster.support.AbstractClusterInvoker;  import java.util.List;  public class MyClusterInvoker<T> extends AbstractClusterInvoker<T> {    public MyClusterInvoker(Directory<T> directory) {  super(directory);  }   @Override  protected Result doInvoke(Invocation invocation, List<Invoker<T>> invokers, LoadBalance loadbalance) throws RpcException {   checkInvokers(invokers, invocation);  RpcContext.*getContext*().setInvokers((List) invokers);  RpcException exception = null;  Result result = null;   // 做自己的集群容错策略     return null;  } }  package com.books.dubbo.cluster;  import org.apache.dubbo.rpc.Invoker; import org.apache.dubbo.rpc.RpcException; import org.apache.dubbo.rpc.cluster.Cluster; import org.apache.dubbo.rpc.cluster.Directory;  public class MyCluster implements Cluster {   @Override  public <T> Invoker<T> join(Directory<T> directory) throws RpcException {    return new MyClusterInvoker(directory);  } } |

## 6.9 Dubbo负载均衡策略概述

Dubbo提供了多种均衡策略，默认为random，也就是每次随机调用一台服务提供者的服务。

Dubbo提供的负载均衡策略有如下几种。

Random LoadBalance：随机策略，按照改路设置权重，比较均匀，并且可以动态调节提供者的权重。

RoundRobin LoadBalance：轮询策略。轮询，按公约后的权重设置轮询比率。会存在执行比较慢的服务提供者堆积请求的情况。

LeastActive LoadBalance：最少活跃调用数。如果每个提供者的活跃数相同，则随机选择一个。在每个服务提供者里维护着一个活跃的计数器，用来记录当前同时处理请求的个数，也就是并发处理任务的个数。

ConsistentHash LoadBalance：一致性Hash策略。一致性Hash，可以保证相同参数的请求总是发到同一个提供者，当某一台提供者机器宕机时，原本发往该提供者的请求，将基于虚拟节点平摊给其他提供者。

|  |  |
| --- | --- |
| APIConsumer | 1. main |
| proxy0 | 2.sayHello |
| InvokerInvocationHandler | 3.invoke |
| MockClusterInvoker | 4.invoke |
| FailoverClusterInvoker | 1. invoke 2. select 3. doSelect |
| RandomLoadBalance | 1. list 2. select 3. doSelect |
| RegistryDirectory | 7.doList |
| DubboInvoker | 12.invoke |

## 6.10 Random LoadBalance 策略源码分析

|  |
| --- |
| package org.apache.dubbo.rpc.cluster.loadbalance;  import org.apache.dubbo.common.URL; import org.apache.dubbo.rpc.Invocation; import org.apache.dubbo.rpc.Invoker;  import java.util.List; import java.util.concurrent.ThreadLocalRandom;  */\*\*  \* random load balance.  \*/* public class RandomLoadBalance extends AbstractLoadBalance {   public static final String *NAME* = "random";   @Override  protected <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {  // Number of invokers  // 1）服务提供者方总个数  int length = invokers.size();  // Every invoker has the same weight?  // 2）所有服务提供者的权重是否都一样  boolean sameWeight = true;  // the weight of every invokers  // 3）存放所有服务提供者设置的权重  int[] weights = new int[length];  // the first invoker's weight  // 4）第一个服务提供者的权重  int firstWeight = getWeight(invokers.get(0), invocation);  weights[0] = firstWeight;  // The sum of weights  // 5）所有服务提供者权重之和  int totalWeight = firstWeight;  for (int i = 1; i < length; i++) {  int weight = getWeight(invokers.get(i), invocation);  // save for later use  // 保存  weights[i] = weight;  // Sum  // 求和  totalWeight += weight;  if (sameWeight && weight != firstWeight) {  sameWeight = false;  }  }  if (totalWeight > 0 && !sameWeight) {  // If (not every invoker has the same weight & at least one invoker's weight>0), select randomly based on totalWeight.  // 7）如果所有服务提供者权重并不是都一样，并且至少有一个提供者  // 的权重大于0，则基于总权重随机选择一个。  int offset = ThreadLocalRandom.*current*().nextInt(totalWeight);  // Return a invoker based on the random value.  // 随机选择一个  for (int i = 0; i < length; i++) {  offset -= weights[i];  if (offset < 0) {  return invokers.get(i);  }  }  }  // If all invokers have the same weight value or totalWeight=0, return evenly.  // 8）如果所有服务提供者权重都一样  // Random在高并发下会导致大量线程竞争同一个原子变量，导致大量线程  // 原地自旋，从而浪费CPU资源  return invokers.get(ThreadLocalRandom.*current*().nextInt(length));  }  }  正常情况下应该选择权重最大的提供者来提供服务，但是Dubbo还考虑另外一个因素，就是服务预热时间。如果服务提供者A的权重比服务提供者B的权重大，但服务提供者A是刚启动的，而服务提供者B已经服务了一些时间，则这时候Dubbo会选择服务提供者B而不是提供者A。  protected int getWeight(Invoker<?> invoker, Invocation invocation) {  // 6.1)获取用户对该服务提供者设置的权重，默认情况下权重都是100  int weight = invoker.getUrl().getMethodParameter(invocation.getMethodName(), Constants.*WEIGHT\_KEY*, Constants.*DEFAULT\_WEIGHT*);  if (weight > 0) {  // 6.2 获取该服务提供者 发布服务的时间timestamp  long timestamp = invoker.getUrl().getParameter(Constants.*REMOTE\_TIMESTAMP\_KEY*, 0L);  if (timestamp > 0L) {  // 6.3）计算该服务已经发布了多少时间  int uptime = (int) (System.*currentTimeMillis*() - timestamp);  // 6.4）获取用户设置的该服务的预热时间，默认是10min  int warmup = invoker.getUrl().getParameter(Constants.*WARMUP\_KEY*, Constants.*DEFAULT\_WARMUP*);  // 6.5）如果该服务提供者还没过预热期，则让该服务提供者的  // 预热时间参与计算权重calculateWarmupWeight.  if (uptime > 0 && uptime < warmup) {  weight = *calculateWarmupWeight*(uptime, warmup, weight);  }  }  }  return weight >= 0 ? weight : 0; } |

## 6.11 RoundRobin LoadBalance 策略源码分析

实现轮询选择策略的是RoundRobinLoadBalance类，

|  |
| --- |
| public static final String *NAME* = "roundrobin"; // RECYCLE\_PERIOD是清理周期 private static int *RECYCLE\_PERIOD* = 60000;  private ConcurrentMap<String, ConcurrentMap<String, WeightedRoundRobin>> methodWeightMap = new ConcurrentHashMap<String, ConcurrentMap<String, WeightedRoundRobin>>(); private AtomicBoolean updateLock = new AtomicBoolean();  protected <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {  // 1）获取调用方法的key  String key = invokers.get(0).getUrl().getServiceKey() + "." + invocation.getMethodName();  // 2）获取该调用方法对应的每个服务提供者的WeightedRoundRobin  // WeightedRoundRobin记录了当前提供者的权重和最后一次更新的时间  ConcurrentMap<String, WeightedRoundRobin> map = methodWeightMap.get(key);  if (map == null) {  methodWeightMap.putIfAbsent(key, new ConcurrentHashMap<String, WeightedRoundRobin>());  map = methodWeightMap.get(key);  }  // 3）遍历所有提供者，计算总权重和权重最大的提供者  int totalWeight = 0;  long maxCurrent = Long.*MIN\_VALUE*;  long now = System.*currentTimeMillis*();  Invoker<T> selectedInvoker = null;  WeightedRoundRobin selectedWRR = null;  // 计算所有提供者对于该方法的总权重和和最大权重，  for (Invoker<T> invoker : invokers) {  String identifyString = invoker.getUrl().toIdentityString();  WeightedRoundRobin weightedRoundRobin = map.get(identifyString);  int weight = getWeight(invoker, invocation);   if (weightedRoundRobin == null) {  weightedRoundRobin = new WeightedRoundRobin();  weightedRoundRobin.setWeight(weight);  map.putIfAbsent(identifyString, weightedRoundRobin);  }  if (weight != weightedRoundRobin.getWeight()) {  //weight changed  // 权重变化  weightedRoundRobin.setWeight(weight);  }  long cur = weightedRoundRobin.increaseCurrent();  weightedRoundRobin.setLastUpdate(now);  if (cur > maxCurrent) {  maxCurrent = cur;  selectedInvoker = invoker;  selectedWRR = weightedRoundRobin;  }  totalWeight += weight;  }  // 4）更新Map  if (!updateLock.get() && invokers.size() != map.size()) {  if (updateLock.compareAndSet(false, true)) {  try {  // copy -> modify -> update reference  // 4.1）拷贝新值  ConcurrentMap<String, WeightedRoundRobin> newMap = new ConcurrentHashMap<String, WeightedRoundRobin>();  newMap.putAll(map);  // 4.2）更新Map，移除过期的  Iterator<Entry<String, WeightedRoundRobin>> it = newMap.entrySet().iterator();  while (it.hasNext()) {  Entry<String, WeightedRoundRobin> item = it.next();  if (now - item.getValue().getLastUpdate() > *RECYCLE\_PERIOD*) {  it.remove();  }  }  // 4.3）更新  methodWeightMap.put(key, newMap);  } finally {  updateLock.set(false);  }  }  }  // 5）返回选择的提供者的invoker对象  // 返回选择的提供者的invoker对象，并把当前提供者对应的方法的方法的  // 权重设置为当前值减去总权重，以便实现轮询对外提供服务。  if (selectedInvoker != null) {  selectedWRR.sel(totalWeight);  return selectedInvoker;  }  // should not happen here  // 返回第一个，不会走到这里  return invokers.get(0); } |

## 6.12 LeastActive LoadBalance 策略源码分析

实现最少活跃调用策略的是LeastActiveLoadBalance类。

|  |
| --- |
| protected <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {  // Number of invokers  // 1）服务提供者个数  int length = invokers.size();  // The least active value of all invokers  // 2）临时变量，用来暂存被调用的最少次数  int leastActive = -1;  // The number of invokers having the same least active value (leastActive)  // 3）记录调用次数等于最小调用次数的服务提供者个数  int leastCount = 0;  // The index of invokers having the same least active value (leastActive)  // 4）记录调用次数等于最小调用次数的服务提供者的下标  int[] leastIndexes = new int[length];  // the weight of every invokers  // 5）记录每个服务提供者的权重  int[] weights = new int[length];  // The sum of the warmup weights of all the least active invokes  // 6）记录调用次数等于最小调用次数的服务提供者的权重和  int totalWeight = 0;  // The weight of the first least active invoke  // 7）记录第一个调用次数等于最小调用次数的服务提供者的权重  int firstWeight = 0;  // Every least active invoker has the same weight value?  // 8）所有的调用次数等于最小调用次数的服务提供者的权重是否一样  boolean sameWeight = true;    // Filter out all the least active invokers  // 9）过滤出所有的调用次数等于最小调用次数的服务提供者  for (int i = 0; i < length; i++) {  Invoker<T> invoker = invokers.get(i);  // Get the active number of the invoke  // 9.1）获取当前服务提供和者被调用次数  int active = RpcStatus.*getStatus*(invoker.getUrl(), invocation.getMethodName()).getActive();  // Get the weight of the invoke configuration. The default value is 100.  // 9.2）计算当前服务提供者的权重（默认为100）  int afterWarmup = getWeight(invoker, invocation);  // save for later use  // 9.3）保存当前服务提供者的权重  weights[i] = afterWarmup;  // If it is the first invoker or the active number of the invoker is less than the current least active number  // 9.4）如果是第一个服务提供者，或者当前服务提供者的调用次数小于当前的  // 最小调用次数  if (leastActive == -1 || active < leastActive) {  // Reset the active number of the current invoker to the least active number  leastActive = active;  // Reset the number of least active invokers  leastCount = 1;  // Put the first least active invoker first in leastIndexs  leastIndexes[0] = i;  // Reset totalWeight  totalWeight = afterWarmup;  // Record the weight the first least active invoker  firstWeight = afterWarmup;  // Each invoke has the same weight (only one invoker here)  sameWeight = true;  // If current invoker's active value equals with leaseActive, then accumulating.  } else if (active == leastActive) {  // Record the index of the least active invoker in leastIndexs order  // 记录调用次数等于最小调用次数的服务提供者的下标  leastIndexes[leastCount++] = i;  // Accumulate the total weight of the least active invoker  // 9.5.1）所有的invoker是否权重都一样  totalWeight += afterWarmup;  // If every invoker has the same weight?  if (sameWeight && i > 0  && afterWarmup != firstWeight) {  sameWeight = false;  }  }  }  // Choose an invoker from all the least active invokers  // 10）如果只有一个最小调用次数的invoker则直接返回  if (leastCount == 1) {  // If we got exactly one invoker having the least active value, return this invoker directly.  return invokers.get(leastIndexes[0]);  }  // 11）如果最小调用次数的invoker有多个并且权重不一样  if (!sameWeight && totalWeight > 0) {  // If (not every invoker has the same weight & at least one invoker's weight>0), select randomly based on   // totalWeight.  int offsetWeight = ThreadLocalRandom.*current*().nextInt(totalWeight);  // Return a invoker based on the random value.  for (int i = 0; i < leastCount; i++) {  int leastIndex = leastIndexes[i];  offsetWeight -= weights[leastIndex];  if (offsetWeight < 0) {  return invokers.get(leastIndex);  }  }  }  // If all invokers have the same weight value or totalWeight=0, return evenly.  // 12）如果最小调用次数的invoker有多个并且权重不一样  return invokers.get(leastIndexes[ThreadLocalRandom.*current*().nextInt(leastCount)]); } |

## 6.13 ConsistentHash LoadBalance 策略源码分析

### 6.13.1 一致性Hash负载均衡策略原理

一致性Hash的特性：

1. 单调性（Monotonicity）:单调性是指如果已经有一些请求通过Hash分派到了相应的服务器进行处理，当又有新的服务器加入到系统中时，应保证原有的请求可以被映射到原有的或者新增的服务器上。
2. 分散性：不能保证同一个用户的请求落到同一台服务器。分散性是指上述情况发生的严重程度。
3. 平衡性：平衡性也就是只负载均衡，指客户端Hash后请求应该能够分散到不同的服务器上。

虚拟节点：

均匀一致性：如果服务器有N台，客户端的Hash值有M个，那么每台服务器应该处理大概M/N个用户的请求，也就是说每台服务器负载尽量均衡。

### 6.13.2 源码分析

ConsistentHashLoadBalance

|  |
| --- |
| protected <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {  String methodName = RpcUtils.*getMethodName*(invocation);  String key = invokers.get(0).getUrl().getServiceKey() + "." + methodName;  int identityHashCode = System.*identityHashCode*(invokers);  ConsistentHashSelector<T> selector = (ConsistentHashSelector<T>) selectors.get(key);  if (selector == null || selector.identityHashCode != identityHashCode) {  selectors.put(key, new ConsistentHashSelector<T>(invokers, methodName, identityHashCode));  selector = (ConsistentHashSelector<T>) selectors.get(key);  }  return selector.select(invocation); }  private static final class ConsistentHashSelector<T> {   private final TreeMap<Long, Invoker<T>> virtualInvokers;   private final int replicaNumber;   private final int identityHashCode;   private final int[] argumentIndex;   ConsistentHashSelector(List<Invoker<T>> invokers, String methodName, int identityHashCode) {  this.virtualInvokers = new TreeMap<Long, Invoker<T>>();  this.identityHashCode = identityHashCode;  URL url = invokers.get(0).getUrl();  // 获取设置的虚拟接地那个数，默认为160个  this.replicaNumber = url.getMethodParameter(methodName, "hash.nodes", 160);  String[] index = Constants.*COMMA\_SPLIT\_PATTERN*.split(url.getMethodParameter(methodName, "hash.arguments", "0"));  argumentIndex = new int[index.length];  for (int i = 0; i < index.length; i++) {  argumentIndex[i] = Integer.*parseInt*(index[i]);  }  for (Invoker<T> invoker : invokers) {  String address = invoker.getUrl().getAddress();  for (int i = 0; i < replicaNumber / 4; i++) {  byte[] digest = md5(address + i);  for (int h = 0; h < 4; h++) {  long m = hash(digest, h);  virtualInvokers.put(m, invoker);  }  }  }  }   public Invoker<T> select(Invocation invocation) {  // 1）获取参与一致性Hash算法的key，默认是第一个参数  String key = toKey(invocation.getArguments());  // 2）根据具体算法计算该key对应md5值  byte[] digest = md5(key);  // 3）计算该key对应Hash环上哪一个点，并选择该点对应的服务提供者。  return selectForKey(hash(digest, 0));  }   private String toKey(Object[] args) {  StringBuilder buf = new StringBuilder();  for (int i : argumentIndex) {  if (i >= 0 && i < args.length) {  buf.append(args[i]);  }  }  return buf.toString();  }   private Invoker<T> selectForKey(long hash) {  Map.Entry<Long, Invoker<T>> entry = virtualInvokers.ceilingEntry(hash);  if (entry == null) {  entry = virtualInvokers.firstEntry();  }  return entry.getValue();  }   private long hash(byte[] digest, int number) {  return (((long) (digest[3 + number \* 4] & 0xFF) << 24)  | ((long) (digest[2 + number \* 4] & 0xFF) << 16)  | ((long) (digest[1 + number \* 4] & 0xFF) << 8)  | (digest[number \* 4] & 0xFF))  & 0xFFFFFFFFL;  }   private byte[] md5(String value) {  MessageDigest md5;  try {  md5 = MessageDigest.*getInstance*("MD5");  } catch (NoSuchAlgorithmException e) {  throw new IllegalStateException(e.getMessage(), e);  }  md5.reset();  byte[] bytes = value.getBytes(StandardCharsets.*UTF\_8*);  md5.update(bytes);  return md5.digest();  }  } |

# dubbo线程模型与线程池策略

## 7.1 Dubbo的线程模型概述

默认的底层网络通信使用的是Netty，服务提供方NettyServer使用两级线程池，其中EventLoopGroup（boss）主要用来接收客户端的连接请求，并把完成TCP三次握手的连接分发给EventLoopGroup（worker）来处理，把boss和worker线程组称为IO线程。

dubbo提供了下面几种线程模型。

1. all（AllDispatcher类）：所有消息都派发到业务线程池，这些消息包括请求，响应、连接事件、断开事件、心跳事件等。
2. direct（DirectDispatcher类）:所有消息都不派发到业务线程池，全部在IO线程上直接执行。
3. message（MessageOnlyDispatcher类）：只有请求响应消息派发到业务线程池，其他消息如连接事件、断开事件、心跳事件等，直接在IO线程上执行。
4. execution（ExecutionDispatcher类）：只把请求消息派发到业务线程池处理，但是响应、连接时间、断开事件、心跳事件等消息直接在IO线程上执行。
5. connection（ConnectionOrderedDispatcher类）：在IO线程上将连接事件、断开事件放入对列，有序地逐个执行，其他消息派发

## 7.2 AllDispatcher源码剖析

|  |
| --- |
| public class AllDispatcher implements Dispatcher {   public static final String *NAME* = "all";   @Override  public ChannelHandler dispatch(ChannelHandler handler, URL url) {  return new AllChannelHandler(handler, url);  }  }  public class AllChannelHandler extends WrappedChannelHandler {   public AllChannelHandler(ChannelHandler handler, URL url) {  super(handler, url);  }  // 连接完成事件，交给业务线程池处理  @Override  public void connected(Channel channel) throws RemotingException {  ExecutorService cexecutor = getExecutorService();  try {  cexecutor.execute(new ChannelEventRunnable(channel, handler, ChannelState.*CONNECTED*));  } catch (Throwable t) {  throw new ExecutionException("connect event", channel, getClass() + " error when process connected event .", t);  }  }  // 链接断开事件，交给业务线程池处理  @Override  public void disconnected(Channel channel) throws RemotingException {  ExecutorService cexecutor = getExecutorService();  try {  cexecutor.execute(new ChannelEventRunnable(channel, handler, ChannelState.*DISCONNECTED*));  } catch (Throwable t) {  throw new ExecutionException("disconnect event", channel, getClass() + " error when process disconnected event .", t);  }  }  // 请求响应事件，交给业务线程池处理  @Override  public void received(Channel channel, Object message) throws RemotingException {  ExecutorService cexecutor = getExecutorService();  try {  cexecutor.execute(new ChannelEventRunnable(channel, handler, ChannelState.*RECEIVED*, message));  } catch (Throwable t) {  //TODO A temporary solution to the problem that the exception information can not be sent to the opposite end after the thread pool is full. Need a refactoring  //fix The thread pool is full, refuses to call, does not return, and causes the consumer to wait for time out  // 临时解决线程池满后异常信息无法发送到对端的问题，待重构  // fix线程池满了拒绝调用不反回，导致消费者一致等待超时  if(message instanceof Request && t instanceof RejectedExecutionException){  Request request = (Request)message;  if(request.isTwoWay()){  String msg = "Server side(" + url.getIp() + "," + url.getPort() + ") threadpool is exhausted ,detail msg:" + t.getMessage();  Response response = new Response(request.getId(), request.getVersion());  response.setStatus(Response.*SERVER\_THREADPOOL\_EXHAUSTED\_ERROR*);  response.setErrorMessage(msg);  channel.send(response);  return;  }  }  throw new ExecutionException(message, channel, getClass() + " error when process received event .", t);  }  }  // 异常处理事件，交给业务线程池处理  @Override  public void caught(Channel channel, Throwable exception) throws RemotingException {  ExecutorService cexecutor = getExecutorService();  try {  cexecutor.execute(new ChannelEventRunnable(channel, handler, ChannelState.*CAUGHT*, exception));  } catch (Throwable t) {  throw new ExecutionException("caught event", channel, getClass() + " error when process caught event .", t);  }  } } |

## 7.3 DireectDispatcher源码剖析

direct线程模型的dispatch()方法直接返回了参数handler，所以其所有事件的处理是在IO线程上进行的

|  |
| --- |
| public class DirectDispatcher implements Dispatcher {   public static final String *NAME* = "direct";   @Override  public ChannelHandler dispatch(ChannelHandler handler, URL url) {  return handler;  }  } |

## 7.4 MessageOnlyDispatcher源码剖析

|  |
| --- |
| public class MessageOnlyDispatcher implements Dispatcher {   public static final String *NAME* = "message";   @Override  public ChannelHandler dispatch(ChannelHandler handler, URL url) {  return new MessageOnlyChannelHandler(handler, url);  }  }  只有请求响应消息派发到业务线程池，其他耗时比较短的连接事件、断开事件、心跳事件等消息则直接在IO线程上执行  public class MessageOnlyChannelHandler extends WrappedChannelHandler {   public MessageOnlyChannelHandler(ChannelHandler handler, URL url) {  super(handler, url);  }   @Override  public void received(Channel channel, Object message) throws RemotingException {  ExecutorService cexecutor = getExecutorService();  try {  cexecutor.execute(new ChannelEventRunnable(channel, handler, ChannelState.*RECEIVED*, message));  } catch (Throwable t) {  throw new ExecutionException(message, channel, getClass() + " error when process received event .", t);  }  }  } |

## 7.5 ExecutionDispatcher源码剖析

|  |
| --- |
| public class ExecutionDispatcher implements Dispatcher {   public static final String *NAME* = "execution";   @Override  public ChannelHandler dispatch(ChannelHandler handler, URL url) {  return new ExecutionChannelHandler(handler, url);  }  }  public class ExecutionChannelHandler extends WrappedChannelHandler {   public ExecutionChannelHandler(ChannelHandler handler, URL url) {  super(handler, url);  }   @Override  public void received(Channel channel, Object message) throws RemotingException {  ExecutorService cexecutor = getExecutorService();  // 只把请求类消息派发到业务线程池处理  if (message instanceof Request) {  try {  cexecutor.execute(new ChannelEventRunnable(channel, handler, ChannelState.*RECEIVED*, message));  } catch (Throwable t) {  // FIXME: when the thread pool is full, SERVER\_THREADPOOL\_EXHAUSTED\_ERROR cannot return properly,  // therefore the consumer side has to wait until gets timeout. This is a temporary solution to prevent  // this scenario from happening, but a better solution should be considered later.  if (t instanceof RejectedExecutionException) {  Request request = (Request) message;  if (request.isTwoWay()) {  String msg = "Server side(" + url.getIp() + "," + url.getPort()  + ") thread pool is exhausted, detail msg:" + t.getMessage();  Response response = new Response(request.getId(), request.getVersion());  response.setStatus(Response.*SERVER\_THREADPOOL\_EXHAUSTED\_ERROR*);  response.setErrorMessage(msg);  channel.send(response);  return;  }  }  throw new ExecutionException(message, channel, getClass() + " error when process received event.", t);  }  } else {  // 响应和其他连接事件、断开事件、心跳事件等消息直接在IO线程上执行  handler.received(channel, message);  }  } } |

## 7.6 ConnectionOrderedDispatcher源码剖析

|  |
| --- |
| public class ConnectionOrderedDispatcher implements Dispatcher {   public static final String *NAME* = "connection";   @Override  public ChannelHandler dispatch(ChannelHandler handler, URL url) {  return new ConnectionOrderedChannelHandler(handler, url);  }  }  public class ConnectionOrderedChannelHandler extends WrappedChannelHandler {   protected final ThreadPoolExecutor connectionExecutor;  private final int queuewarninglimit;  // 1）构造函数  public ConnectionOrderedChannelHandler(ChannelHandler handler, URL url) {  super(handler, url);  // 1.1）创建线程池  String threadName = url.getParameter(Constants.*THREAD\_NAME\_KEY*, Constants.*DEFAULT\_THREAD\_NAME*);  connectionExecutor = new ThreadPoolExecutor(1, 1,  0L, TimeUnit.*MILLISECONDS*,  new LinkedBlockingQueue<Runnable>(url.getPositiveParameter(Constants.*CONNECT\_QUEUE\_CAPACITY*, Integer.*MAX\_VALUE*)),  new NamedThreadFactory(threadName, true),  new AbortPolicyWithReport(threadName, url)  ); // FIXME There's no place to release connectionExecutor!  // 1.2）线程池对列元素限制警告  queuewarninglimit = url.getParameter(Constants.*CONNECT\_QUEUE\_WARNING\_SIZE*, Constants.*DEFAULT\_CONNECT\_QUEUE\_WARNING\_SIZE*);  }  // 2）链接建立事件  @Override  public void connected(Channel channel) throws RemotingException {  try {  checkQueueLength();  connectionExecutor.execute(new ChannelEventRunnable(channel, handler, ChannelState.*CONNECTED*));  } catch (Throwable t) {  throw new ExecutionException("connect event", channel, getClass() + " error when process connected event .", t);  }  }  // 链接断开事件  @Override  public void disconnected(Channel channel) throws RemotingException {  try {  checkQueueLength();  connectionExecutor.execute(new ChannelEventRunnable(channel, handler, ChannelState.*DISCONNECTED*));  } catch (Throwable t) {  throw new ExecutionException("disconnected event", channel, getClass() + " error when process disconnected event .", t);  }  }  // 请求响应事件  @Override  public void received(Channel channel, Object message) throws RemotingException {  ExecutorService cexecutor = getExecutorService();  try {  cexecutor.execute(new ChannelEventRunnable(channel, handler, ChannelState.*RECEIVED*, message));  } catch (Throwable t) {  //fix, reject exception can not be sent to consumer because thread pool is full, resulting in consumers waiting till timeout.  if (message instanceof Request && t instanceof RejectedExecutionException) {  Request request = (Request) message;  if (request.isTwoWay()) {  String msg = "Server side(" + url.getIp() + "," + url.getPort() + ") threadpool is exhausted ,detail msg:" + t.getMessage();  Response response = new Response(request.getId(), request.getVersion());  response.setStatus(Response.*SERVER\_THREADPOOL\_EXHAUSTED\_ERROR*);  response.setErrorMessage(msg);  channel.send(response);  return;  }  }  throw new ExecutionException(message, channel, getClass() + " error when process received event .", t);  }  }  // 5）异常事件  @Override  public void caught(Channel channel, Throwable exception) throws RemotingException {  ExecutorService cexecutor = getExecutorService();  try {  cexecutor.execute(new ChannelEventRunnable(channel, handler, ChannelState.*CAUGHT*, exception));  } catch (Throwable t) {  throw new ExecutionException("caught event", channel, getClass() + " error when process caught event .", t);  }  }  // 6）检查线程池对列元素个数  private void checkQueueLength() {  if (connectionExecutor.getQueue().size() > queuewarninglimit) {  *logger*.warn(new IllegalThreadStateException("connectionordered channel handler `queue size: " + connectionExecutor.getQueue().size() + " exceed the warning limit number :" + queuewarninglimit));  }  } } |

## 7.7 线程模型的确定时机

|  |
| --- |
| public NettyServer(URL url, ChannelHandler handler) throws RemotingException {  // 加载具体的线程模型  super(url, ChannelHandlers.*wrap*(handler, ExecutorUtil.*setThreadName*(url, *SERVER\_THREAD\_POOL\_NAME*))); }  public class ChannelHandlers {   private static ChannelHandlers *INSTANCE* = new ChannelHandlers();   protected ChannelHandlers() {  }   public static ChannelHandler wrap(ChannelHandler handler, URL url) {  return ChannelHandlers.*getInstance*().wrapInternal(handler, url);  }   protected static ChannelHandlers getInstance() {  return *INSTANCE*;  }   static void setTestingChannelHandlers(ChannelHandlers instance) {  *INSTANCE* = instance;  }  //   protected ChannelHandler wrapInternal(ChannelHandler handler, URL url) {  return new MultiMessageHandler(new HeartbeatHandler(ExtensionLoader.*getExtensionLoader*(Dispatcher.class)  .getAdaptiveExtension().dispatch(handler, url)));  } } |

## 7.8 如何扩展自定义的线程模型

|  |
| --- |
| package com.books.dubbo.cluster;  import org.apache.dubbo.common.URL; import org.apache.dubbo.remoting.ChannelHandler; import org.apache.dubbo.remoting.Dispatcher;  public class MyDispatcher implements Dispatcher {    public static final String *NAME* = "mydispatcher";   @Override  public ChannelHandler dispatch(ChannelHandler handler, URL url) {  return new MyChannelHandler(handler,url);  } }  package com.books.dubbo.cluster;  import org.apache.dubbo.common.URL; import org.apache.dubbo.remoting.ChannelHandler; import org.apache.dubbo.remoting.transport.dispatcher.WrappedChannelHandler;  public class MyChannelHandler extends WrappedChannelHandler {   public MyChannelHandler(ChannelHandler handler, URL url) {  super(handler, url);  } } |

## 7.9 Dubbo线程池策略

为了尽量早地释放netty的IO线程，某些线程模型会把请求投递到线程池进行异步处理，那么这里所谓的线程池是什么样的线程池？其实这里的线程池ThreadPool也是一个扩展接口的SPI，Dubbo提供了该扩展接口的一些实现，具体如下：

1. FixedThreadPool：创建一个具有固定个数线程的线程池
2. LimitedThreadPool：创建一个线程池，这个线程池中的线程个数随着需要量动态增加，但是数量不超过配置的阈值。另外，空闲的线程不会被回收，会一直存在。
3. EagerThreadPool：创建一个线程池，在这个线程池，所有的核心线程都处于忙碌状态时，将创建新的线程来执行新任务，而不是把任务放入线程池阻塞对列。
4. CacheThreadPool：创建一个自适应线程池，当线程空闲1分钟时，线程会被回收；当有新请求到来时，会创建新线程。

## 7.10 FixedThreadPool源码分析

|  |
| --- |
| public class FixedThreadPool implements ThreadPool {   @Override  public Executor getExecutor(URL url) {  // 1）获取线程名称  String name = url.getParameter(Constants.*THREAD\_NAME\_KEY*, Constants.*DEFAULT\_THREAD\_NAME*);  // 2）获取线程池线程个数  int threads = url.getParameter(Constants.*THREADS\_KEY*, Constants.*DEFAULT\_THREADS*);  // 3）获取线程池队列大小  int queues = url.getParameter(Constants.*QUEUES\_KEY*, Constants.*DEFAULT\_QUEUES*);  // 4）获取线程池对列大小  return new ThreadPoolExecutor(threads, threads, 0, TimeUnit.*MILLISECONDS*,  queues == 0 ? new SynchronousQueue<Runnable>() :  (queues < 0 ? new LinkedBlockingQueue<Runnable>()  : new LinkedBlockingQueue<Runnable>(queues)),  new NamedInternalThreadFactory(name, true), new AbortPolicyWithReport(name, url));  }  } |

## 7.11 LimitedThreadPool源码分析

|  |
| --- |
| public class LimitedThreadPool implements ThreadPool {   @Override  public Executor getExecutor(URL url) {  // 1）获取线程名称  String name = url.getParameter(Constants.*THREAD\_NAME\_KEY*, Constants.*DEFAULT\_THREAD\_NAME*);  // 2）获取线程池核心线程个数  int cores = url.getParameter(Constants.*CORE\_THREADS\_KEY*, Constants.*DEFAULT\_CORE\_THREADS*);  // 3）获取线程池核最大线程个数  int threads = url.getParameter(Constants.*THREADS\_KEY*, Constants.*DEFAULT\_THREADS*);  // 4）获取线程池对列大小  int queues = url.getParameter(Constants.*QUEUES\_KEY*, Constants.*DEFAULT\_QUEUES*);  // 5）使用JUC包的ThreadPoolExecutor创建线程池  return new ThreadPoolExecutor(cores, threads, Long.*MAX\_VALUE*, TimeUnit.*MILLISECONDS*,  queues == 0 ? new SynchronousQueue<Runnable>() :  (queues < 0 ? new LinkedBlockingQueue<Runnable>()  : new LinkedBlockingQueue<Runnable>(queues)),  new NamedInternalThreadFactory(name, true), new AbortPolicyWithReport(name, url));  }  } |

## 7.12 EagerThreadPool源码分析

|  |
| --- |
| public class EagerThreadPool implements ThreadPool {   @Override  public Executor getExecutor(URL url) {  // 1）获取线程名称  String name = url.getParameter(Constants.*THREAD\_NAME\_KEY*, Constants.*DEFAULT\_THREAD\_NAME*);  // 2）获取线程池核心线程个数  int cores = url.getParameter(Constants.*CORE\_THREADS\_KEY*, Constants.*DEFAULT\_CORE\_THREADS*);  // 3）获取线程池核最大线程个数  int threads = url.getParameter(Constants.*THREADS\_KEY*, Integer.*MAX\_VALUE*);  // 4）获取线程池对列大小  int queues = url.getParameter(Constants.*QUEUES\_KEY*, Constants.*DEFAULT\_QUEUES*);  // 5）获取线程池对列线程空闲时间被回收  int alive = url.getParameter(Constants.*ALIVE\_KEY*, Constants.*DEFAULT\_ALIVE*);   // init queue and executor  // 6）初始化自定义线程池核和对列  TaskQueue<Runnable> taskQueue = new TaskQueue<Runnable>(queues <= 0 ? 1 : queues);  EagerThreadPoolExecutor executor = new EagerThreadPoolExecutor(cores,  threads,  alive,  TimeUnit.*MILLISECONDS*,  taskQueue,  new NamedInternalThreadFactory(name, true),  new AbortPolicyWithReport(name, url));  taskQueue.setExecutor(executor);  return executor;  } }  EagerThreadPoolExecutor与JUC包中的ThreadPoolExecutor不同之处在于，对于后者来说，当线程池核心线程个数达到设置的阈值时，新来的任务会被放入线程池对列，等对列满了以后，才会开启新的线程来处理任务（前提是当前线程个数没有超过线程池最大线程数）； |

## 7.13 CachedThreadPool源码分析

|  |
| --- |
| @Override public Executor getExecutor(URL url) {  // 1)获取线程名称  String name = url.getParameter(Constants.*THREAD\_NAME\_KEY*, Constants.*DEFAULT\_THREAD\_NAME*);  // 2)获取线程池核心线程个数  int cores = url.getParameter(Constants.*CORE\_THREADS\_KEY*, Constants.*DEFAULT\_CORE\_THREADS*);  // 3)获取线程池核最大线程个数  int threads = url.getParameter(Constants.*THREADS\_KEY*, Integer.*MAX\_VALUE*);  // 4)获取线程池对列大小  int queues = url.getParameter(Constants.*QUEUES\_KEY*, Constants.*DEFAULT\_QUEUES*);  // 5)获取线程池对列线程空闲多少时间被回收  int alive = url.getParameter(Constants.*ALIVE\_KEY*, Constants.*DEFAULT\_ALIVE*);  return new ThreadPoolExecutor(cores, threads, alive, TimeUnit.*MILLISECONDS*,  queues == 0 ? new SynchronousQueue<Runnable>() :  (queues < 0 ? new LinkedBlockingQueue<Runnable>()  : new LinkedBlockingQueue<Runnable>(queues)),  new NamedInternalThreadFactory(name, true), new AbortPolicyWithReport(name, url)); } |

## 7.14 线程池的确定时机

这里以线程模型AllDispatcher为例。线程模型AllDispatcher对应的处理器是AllChannelHandler。

|  |
| --- |
| public AllChannelHandler(ChannelHandler handler, URL url) {  // 调用父类  super(handler, url); }  public WrappedChannelHandler(ChannelHandler handler, URL url) {  this.handler = handler;  this.url = url;  executor = (ExecutorService) ExtensionLoader.*getExtensionLoader*(ThreadPool.class).getAdaptiveExtension().getExecutor(url);   String componentKey = Constants.*EXECUTOR\_SERVICE\_COMPONENT\_KEY*;  if (Constants.*CONSUMER\_SIDE*.equalsIgnoreCase(url.getParameter(Constants.*SIDE\_KEY*))) {  componentKey = Constants.*CONSUMER\_SIDE*;  }  DataStore dataStore = ExtensionLoader.*getExtensionLoader*(DataStore.class).getDefaultExtension();  dataStore.put(componentKey, Integer.*toString*(url.getPort()), executor); } |

# dubbo如何实现泛化引用

泛化接口调用方式主要在服务消费端没有API接口类及模型类元(比如入参和出参的POJO类)的情况下，其参数及返回值中没有对应的POJO类，所以全部的POJO均转换为Map表示。

主要内容：服务消费端如何使用GenericImplFilter拦截泛化调用，把泛化参数进行校验并发起远程调用；服务提供方如何使用GenericFilter拦截请求，并把泛化参数进行反序列化处理，然后把请求转发给具体的服务进行执行。

## 8.1 服务消费端GenericImplFilter源码分析

|  |  |
| --- | --- |
| APICenericConsumer | 1.main |
| proxy0 | 2.sayHello |
| InvokerInvocationHandler | 3.invoke |
| MockClusterInvoker | 4.invoke |
| FailoverClusterInvoker | 5.invoke |
| ProtocolFilterWrapper | 6.invoke |
| GenericImplFilter | 7.invoke |
| DubboInvoker | 8.invoke |
|  |  |

GenericImplFilter

|  |
| --- |
| public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {  String generic = invoker.getUrl().getParameter(Constants.*GENERIC\_KEY*);  // 1)判断是否是泛化调用  if (ProtocolUtils.*isGeneric*(generic)  && !Constants.*$INVOKE*.equals(invocation.getMethodName())  && invocation instanceof RpcInvocation) {  RpcInvocation invocation2 = (RpcInvocation) invocation;  String methodName = invocation2.getMethodName();  // 1.1)获取泛化参数  Class<?>[] parameterTypes = invocation2.getParameterTypes();  Object[] arguments = invocation2.getArguments();   String[] types = new String[parameterTypes.length];  for (int i = 0; i < parameterTypes.length; i++) {  types[i] = ReflectUtils.*getName*(parameterTypes[i]);  }    Object[] args;  //  if (ProtocolUtils.*isBeanGenericSerialization*(generic)) {  args = new Object[arguments.length];  for (int i = 0; i < arguments.length; i++) {  args[i] = JavaBeanSerializeUtil.*serialize*(arguments[i], JavaBeanAccessor.*METHOD*);  }  } else {  args = PojoUtils.*generalize*(arguments);  }   invocation2.setMethodName(Constants.*$INVOKE*);  invocation2.setParameterTypes(*GENERIC\_PARAMETER\_TYPES*);  invocation2.setArguments(new Object[]{methodName, types, args});  Result result = invoker.invoke(invocation2);   if (!result.hasException()) {  Object value = result.getValue();  try {  Method method = invoker.getInterface().getMethod(methodName, parameterTypes);  if (ProtocolUtils.*isBeanGenericSerialization*(generic)) {  if (value == null) {  return new RpcResult(value);  } else if (value instanceof JavaBeanDescriptor) {  return new RpcResult(JavaBeanSerializeUtil.*deserialize*((JavaBeanDescriptor) value));  } else {  throw new RpcException(  "The type of result value is " +  value.getClass().getName() +  " other than " +  JavaBeanDescriptor.class.getName() +  ", and the result is " +  value);  }  } else {  return new RpcResult(PojoUtils.*realize*(value, method.getReturnType(), method.getGenericReturnType()));  }  } catch (NoSuchMethodException e) {  throw new RpcException(e.getMessage(), e);  }  } else if (result.getException() instanceof GenericException) {  GenericException exception = (GenericException) result.getException();  try {  String className = exception.getExceptionClass();  Class<?> clazz = ReflectUtils.*forName*(className);  Throwable targetException = null;  Throwable lastException = null;  try {  targetException = (Throwable) clazz.newInstance();  } catch (Throwable e) {  lastException = e;  for (Constructor<?> constructor : clazz.getConstructors()) {  try {  targetException = (Throwable) constructor.newInstance(new Object[constructor.getParameterTypes().length]);  break;  } catch (Throwable e1) {  lastException = e1;  }  }  }  if (targetException != null) {  try {  Field field = Throwable.class.getDeclaredField("detailMessage");  if (!field.isAccessible()) {  field.setAccessible(true);  }  field.set(targetException, exception.getExceptionMessage());  } catch (Throwable e) {  *logger*.warn(e.getMessage(), e);  }  result = new RpcResult(targetException);  } else if (lastException != null) {  throw lastException;  }  } catch (Throwable e) {  throw new RpcException("Can not deserialize exception " + exception.getExceptionClass() + ", message: " + exception.getExceptionMessage(), e);  }  }  return result;  }   if (invocation.getMethodName().equals(Constants.*$INVOKE*)  && invocation.getArguments() != null  && invocation.getArguments().length == 3  && ProtocolUtils.*isGeneric*(generic)) {   Object[] args = (Object[]) invocation.getArguments()[2];  if (ProtocolUtils.*isJavaGenericSerialization*(generic)) {   for (Object arg : args) {  if (!(byte[].class == arg.getClass())) {  error(generic, byte[].class.getName(), arg.getClass().getName());  }  }  } else if (ProtocolUtils.*isBeanGenericSerialization*(generic)) {  for (Object arg : args) {  if (!(arg instanceof JavaBeanDescriptor)) {  error(generic, JavaBeanDescriptor.class.getName(), arg.getClass().getName());  }  }  }  // 1.4)设置泛化调用方式  ((RpcInvocation) invocation).setAttachment(  Constants.*GENERIC\_KEY*, invoker.getUrl().getParameter(Constants.*GENERIC\_KEY*));  }  return invoker.invoke(invocation); } |

## 8.2 服务提供端GenericFilter源码分析

|  |  |
| --- | --- |
| ChannelEventRunnable | 1.run |
| DecodeHandler | 2.received |
| HeaderExchangeHandler | 1. received 2. handleRequest |
| ExchangerHandlerAdapter | 5.replay |
| GenericFilter | 6.invoke |
| AbstractProxyInvoker | 1. invoke |
|  |  |

|  |
| --- |
| @Override public Result invoke(Invoker<?> invoker, Invocation inv) throws RpcException {  // 1)判断是否为泛化请求  if (inv.getMethodName().equals(Constants.*$INVOKE*)  && inv.getArguments() != null  && inv.getArguments().length == 3  && !GenericService.class.isAssignableFrom(invoker.getInterface())) {  // 1.1）获取参数名称、参数类型、参数值  String name = ((String) inv.getArguments()[0]).trim();  String[] types = (String[]) inv.getArguments()[1];  Object[] args = (Object[]) inv.getArguments()[2];  try {  // 1.2）使用反射获取调用方法  Method method = ReflectUtils.*findMethodByMethodSignature*(invoker.getInterface(), name, types);  Class<?>[] params = method.getParameterTypes();  if (args == null) {  args = new Object[params.length];  }  // 1.3）获取泛化引用方使用的泛化类型  String generic = inv.getAttachment(Constants.*GENERIC\_KEY*);  // 1.4)泛化类型为空，则使用generic=true的泛化方式  if (StringUtils.*isBlank*(generic)) {  generic = RpcContext.*getContext*().getAttachment(Constants.*GENERIC\_KEY*);  }  // 1.5)generic=native的方式  if (StringUtils.*isEmpty*(generic)  || ProtocolUtils.*isDefaultGenericSerialization*(generic)) {  args = PojoUtils.*realize*(args, params, method.getGenericParameterTypes());  } else if (ProtocolUtils.*isJavaGenericSerialization*(generic)) {  for (int i = 0; i < args.length; i++) {  if (byte[].class == args[i].getClass()) {  try(UnsafeByteArrayInputStream is = new UnsafeByteArrayInputStream((byte[]) args[i])) {  args[i] = ExtensionLoader.*getExtensionLoader*(Serialization.class)  .getExtension(Constants.*GENERIC\_SERIALIZATION\_NATIVE\_JAVA*)  .deserialize(null, is).readObject();  } catch (Exception e) {  throw new RpcException("Deserialize argument [" + (i + 1) + "] failed.", e);  }  } else {  throw new RpcException(  "Generic serialization [" +  Constants.*GENERIC\_SERIALIZATION\_NATIVE\_JAVA* +  "] only support message type " +  byte[].class +  " and your message type is " +  args[i].getClass());  }  }  } else if (ProtocolUtils.*isBeanGenericSerialization*(generic)) {  // 1.6)generic=bean的方式  for (int i = 0; i < args.length; i++) {  if (args[i] instanceof JavaBeanDescriptor) {  args[i] = JavaBeanSerializeUtil.*deserialize*((JavaBeanDescriptor) args[i]);  } else {  throw new RpcException(  "Generic serialization [" +  Constants.*GENERIC\_SERIALIZATION\_BEAN* +  "] only support message type " +  JavaBeanDescriptor.class.getName() +  " and your message type is " +  args[i].getClass().getName());  }  }  }  //1.7）传递请求道filter链的下一个filter，最后执行具体服务。  Result result = invoker.invoke(new RpcInvocation(method, args, inv.getAttachments()));  if (result.hasException()  && !(result.getException() instanceof GenericException)) {  return new RpcResult(new GenericException(result.getException()));  }  // 1.8)如果泛化类型巍峨nativejava则使用Java序列化方式对结果进行序列化处理  if (ProtocolUtils.*isJavaGenericSerialization*(generic)) {  try {  UnsafeByteArrayOutputStream os = new UnsafeByteArrayOutputStream(512);  ExtensionLoader.*getExtensionLoader*(Serialization.class)  .getExtension(Constants.*GENERIC\_SERIALIZATION\_NATIVE\_JAVA*)  .serialize(null, os).writeObject(result.getValue());  return new RpcResult(os.toByteArray());  } catch (IOException e) {  throw new RpcException("Serialize result failed.", e);  }  } else if (ProtocolUtils.*isBeanGenericSerialization*(generic)) {  // 1.9)如果泛化雷类型为bean，则使用bean序列化方式对结果进行序列化处理。  return new RpcResult(JavaBeanSerializeUtil.*serialize*(result.getValue(), JavaBeanAccessor.*METHOD*));  } else {  // 1.10）如果泛化类型为true，则使用POJO序列化方式对结果进行序列化处理  return new RpcResult(PojoUtils.*generalize*(result.getValue()));  }  } catch (NoSuchMethodException e) {  throw new RpcException(e.getMessage(), e);  } catch (ClassNotFoundException e) {  throw new RpcException(e.getMessage(), e);  }  }  return invoker.invoke(inv); } |

# dubbo并发控制

## 9.1服务消费端并发控制

|  |  |
| --- | --- |
| *InvokerInvocationHandler* | 1. invoke |
| MockClusterInvoker | 2.invoke |
| FailoverClusterInvoker | 1. invoke 2. doInvoke 3. select 4. doSelect |
| InvokerDelegete | 7.invoke |
| ProtocolFIlterWrapper | 8.invoke |
| ActiveLimitFilter | 9.invoke |
| DubboInvoker | 1. invoke 2. doInvoke |

|  |
| --- |
| public class ActiveLimitFilter implements Filter {   @Override  public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {  // 1)获取URL和调用的方法名称  URL url = invoker.getUrl();  String methodName = invocation.getMethodName();  // 2）获取设置的actives的值（默认为0）和最大可用并发数  int max = invoker.getUrl().getMethodParameter(methodName, Constants.*ACTIVES\_KEY*, 0);  // 3) 根据URL和方法名获取对应的状态对象  RpcStatus count = RpcStatus.*getStatus*(invoker.getUrl(), invocation.getMethodName());  // 4）判断是不是超过并发限制  if (!count.*beginCount*(url, methodName, max)) {  long timeout = invoker.getUrl().getMethodParameter(invocation.getMethodName(), Constants.*TIMEOUT\_KEY*, 0);  long start = System.*currentTimeMillis*();  long remain = timeout;  // 4.1)超过并发限制则阻塞当前线程timeout时间  synchronized (count) {  while (!count.*beginCount*(url, methodName, max)) {  try {  count.wait(remain);  } catch (InterruptedException e) {  // ignore  }  long elapsed = System.*currentTimeMillis*() - start;  remain = timeout - elapsed;  // 4.2)超时了还没有被唤醒则抛出异常  if (remain <= 0) {  throw new RpcException("Waiting concurrent invoke timeout in client-side for service: "  + invoker.getInterface().getName() + ", method: "  + invocation.getMethodName() + ", elapsed: " + elapsed  + ", timeout: " + timeout + ". concurrent invokes: " + count.getActive()  + ". max concurrent invoke limit: " + max);  }  }  }  }  // 5)到这里说明计划并发数没达到限制，则继续Filter链的调用，正常发起远程调用  boolean isSuccess = true;  long begin = System.*currentTimeMillis*();  try {  return invoker.invoke(invocation);  } catch (RuntimeException t) {  isSuccess = false;  throw t;  } finally {  // 6)远程调用完成后，当前激活并发数减去1，听过notifyAll()方法  // 激活所有挂起的线程。  count.*endCount*(url, methodName, System.*currentTimeMillis*() - begin, isSuccess);  if (max > 0) {  synchronized (count) {  count.notifyAll();  }  }  }  } } |

RpcStatus

|  |
| --- |
| public class RpcStatus {   private static final ConcurrentMap<String, RpcStatus> *SERVICE\_STATISTICS* = new ConcurrentHashMap<String, RpcStatus>();  // 7）缓存key为接口类，value为map  private static final ConcurrentMap<String, ConcurrentMap<String, RpcStatus>> *METHOD\_STATISTICS* = new ConcurrentHashMap<String, ConcurrentMap<String, RpcStatus>>();  private final ConcurrentMap<String, Object> values = new ConcurrentHashMap<String, Object>();  // 8)当前激活并发数  private final AtomicInteger active = new AtomicInteger();  private final AtomicLong total = new AtomicLong();  private final AtomicInteger failed = new AtomicInteger();  private final AtomicLong totalElapsed = new AtomicLong();  private final AtomicLong failedElapsed = new AtomicLong();  private final AtomicLong maxElapsed = new AtomicLong();  private final AtomicLong failedMaxElapsed = new AtomicLong();  private final AtomicLong succeededMaxElapsed = new AtomicLong();   private RpcStatus() {  }   */\*\*  \** ***@param*** *url  \** ***@return*** *status  \*/* public static RpcStatus getStatus(URL url) {  String uri = url.toIdentityString();  RpcStatus status = *SERVICE\_STATISTICS*.get(uri);  if (status == null) {  *SERVICE\_STATISTICS*.putIfAbsent(uri, new RpcStatus());  status = *SERVICE\_STATISTICS*.get(uri);  }  return status;  }   */\*\*  \** ***@param*** *url  \*/* public static void removeStatus(URL url) {  String uri = url.toIdentityString();  *SERVICE\_STATISTICS*.remove(uri);  }   */\*\*  \** ***@param*** *url  \** ***@param*** *methodName  \** ***@return*** *status  \*/*  *// 9）获取方法对应的RpcStatus* public static RpcStatus getStatus(URL url, String methodName) {  String uri = url.toIdentityString();  ConcurrentMap<String, RpcStatus> map = *METHOD\_STATISTICS*.get(uri);  if (map == null) {  *METHOD\_STATISTICS*.putIfAbsent(uri, new ConcurrentHashMap<String, RpcStatus>());  map = *METHOD\_STATISTICS*.get(uri);  }  RpcStatus status = map.get(methodName);  if (status == null) {  map.putIfAbsent(methodName, new RpcStatus());  status = map.get(methodName);  }  return status;  }   */\*\*  \** ***@param*** *url  \*/* public static void removeStatus(URL url, String methodName) {  String uri = url.toIdentityString();  ConcurrentMap<String, RpcStatus> map = *METHOD\_STATISTICS*.get(uri);  if (map != null) {  map.remove(methodName);  }  }  // 10）递增方法对应的激活并发数  public static void beginCount(URL url, String methodName) {  *beginCount*(url, methodName, Integer.*MAX\_VALUE*);  }   */\*\*  \** ***@param*** *url  \*/* public static boolean beginCount(URL url, String methodName, int max) {  max = (max <= 0) ? Integer.*MAX\_VALUE* : max;  RpcStatus appStatus = *getStatus*(url);  RpcStatus methodStatus = *getStatus*(url, methodName);  if (methodStatus.active.incrementAndGet() > max) {  methodStatus.active.decrementAndGet();  return false;  } else {  appStatus.active.incrementAndGet();  return true;  }  }   */\*\*  \** ***@param*** *url  \** ***@param*** *elapsed  \** ***@param*** *succeeded  \*/*  *// 原子性的递减方法对应的激活并发数* public static void endCount(URL url, String methodName, long elapsed, boolean succeeded) {  *endCount*(*getStatus*(url), elapsed, succeeded);  *endCount*(*getStatus*(url, methodName), elapsed, succeeded);  }   private static void endCount(RpcStatus status, long elapsed, boolean succeeded) {  status.active.decrementAndGet();  status.total.incrementAndGet();  status.totalElapsed.addAndGet(elapsed);  if (status.maxElapsed.get() < elapsed) {  status.maxElapsed.set(elapsed);  }  if (succeeded) {  if (status.succeededMaxElapsed.get() < elapsed) {  status.succeededMaxElapsed.set(elapsed);  }  } else {  status.failed.incrementAndGet();  status.failedElapsed.addAndGet(elapsed);  if (status.failedMaxElapsed.get() < elapsed) {  status.failedMaxElapsed.set(elapsed);  }  }  }   */\*\*  \* set value.  \*  \** ***@param*** *key  \** ***@param*** *value  \*/* public void set(String key, Object value) {  values.put(key, value);  }   */\*\*  \* get value.  \*  \** ***@param*** *key  \** ***@return*** *value  \*/* public Object get(String key) {  return values.get(key);  }   */\*\*  \* get active.  \*  \** ***@return*** *active  \*/* public int getActive() {  return active.get();  }   */\*\*  \* get total.  \*  \** ***@return*** *total  \*/* public long getTotal() {  return total.longValue();  }   */\*\*  \* get total elapsed.  \*  \** ***@return*** *total elapsed  \*/* public long getTotalElapsed() {  return totalElapsed.get();  }   */\*\*  \* get average elapsed.  \*  \** ***@return*** *average elapsed  \*/* public long getAverageElapsed() {  long total = getTotal();  if (total == 0) {  return 0;  }  return getTotalElapsed() / total;  }   */\*\*  \* get max elapsed.  \*  \** ***@return*** *max elapsed  \*/* public long getMaxElapsed() {  return maxElapsed.get();  }   */\*\*  \* get failed.  \*  \** ***@return*** *failed  \*/* public int getFailed() {  return failed.get();  }   */\*\*  \* get failed elapsed.  \*  \** ***@return*** *failed elapsed  \*/* public long getFailedElapsed() {  return failedElapsed.get();  }   */\*\*  \* get failed average elapsed.  \*  \** ***@return*** *failed average elapsed  \*/* public long getFailedAverageElapsed() {  long failed = getFailed();  if (failed == 0) {  return 0;  }  return getFailedElapsed() / failed;  }   */\*\*  \* get failed max elapsed.  \*  \** ***@return*** *failed max elapsed  \*/* public long getFailedMaxElapsed() {  return failedMaxElapsed.get();  }   */\*\*  \* get succeeded.  \*  \** ***@return*** *succeeded  \*/* public long getSucceeded() {  return getTotal() - getFailed();  }   */\*\*  \* get succeeded elapsed.  \*  \** ***@return*** *succeeded elapsed  \*/* public long getSucceededElapsed() {  return getTotalElapsed() - getFailedElapsed();  }   */\*\*  \* get succeeded average elapsed.  \*  \** ***@return*** *succeeded average elapsed  \*/* public long getSucceededAverageElapsed() {  long succeeded = getSucceeded();  if (succeeded == 0) {  return 0;  }  return getSucceededElapsed() / succeeded;  }   */\*\*  \* get succeeded max elapsed.  \*  \** ***@return*** *succeeded max elapsed.  \*/* public long getSucceededMaxElapsed() {  return succeededMaxElapsed.get();  }   */\*\*  \* Calculate average TPS (Transaction per second).  \*  \** ***@return*** *tps  \*/* public long getAverageTps() {  if (getTotalElapsed() >= 1000L) {  return getTotal() / (getTotalElapsed() / 1000L);  }  return getTotal();  }   } |

## 9.2服务提供端并发控制

|  |  |
| --- | --- |
| ChannelEventRunnnable | 1.run |
| DecodeHandler | 2.received |
| HeaderExchangeHandler | 1. received 2. handleRequest |
| ExchangeHandlerAdapter | 5.reply |
| ExecuteLimitFilter | 6.invoke |
| AbstractProxyInvoker | 7.invoke |

|  |
| --- |
| public class ExecuteLimitFilter implements Filter {   @Override  public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {  // 1)获取URL和调用的方法名称  URL url = invoker.getUrl();  String methodName = invocation.getMethodName();  // 2)获取设置的executes的值和最大可用并发数  int max = url.getMethodParameter(methodName, Constants.*EXECUTES\_KEY*, 0);  // 3)判断是不是超过并发设置  if (!RpcStatus.*beginCount*(url, methodName, max)) {  throw new RpcException("Failed to invoke method " + invocation.getMethodName() + " in provider " +  url + ", cause: The service using threads greater than <dubbo:service executes=\"" + max +  "\" /> limited.");  }  // 4）到这里说明激活并发数没达到限制，继续Filter链的处理，  // 正常执行服务处理  long begin = System.*currentTimeMillis*();  boolean isSuccess = true;  try {  return invoker.invoke(invocation);  } catch (Throwable t) {  isSuccess = false;  if (t instanceof RuntimeException) {  throw (RuntimeException) t;  } else {  throw new RpcException("unexpected exception when ExecuteLimitFilter", t);  }  } finally {  // 5)当前方法激活并发数减去1  RpcStatus.*endCount*(url, methodName, System.*currentTimeMillis*() - begin, isSuccess);  }  }  } |

# dubbo隐式参数传递

## 10.1 服务消费端AbstractClusterInvoker原理剖析

|  |  |
| --- | --- |
| APIConsumer | 1. main |
| proxy0 | 2.sayHello |
| InvoerInvocationHandler | 3.invoke |
| MockClusterInvoker | 4.invoke |
| FailoverClusterInvoker | 5.invoke |
| ConsumerContextFilter | 6.invoke |
| DubboInvoker | 7.invoke |
| RpcContext | 8.clearAttachments |

|  |
| --- |
| public Result invoke(final Invocation invocation) throws RpcException {  checkWhetherDestroyed();   // binding attachments into invocation.  // 从上下文中获取附加属性并设置到RpcInvocation对象中  Map<String, String> contextAttachments = RpcContext.*getContext*().getAttachments();  if (contextAttachments != null && contextAttachments.size() != 0) {  ((RpcInvocation) invocation).addAttachments(contextAttachments);  }   List<Invoker<T>> invokers = list(invocation);  LoadBalance loadbalance = initLoadBalance(invokers, invocation);  RpcUtils.*attachInvocationIdIfAsync*(getUrl(), invocation);  return doInvoke(invocation, invokers, loadbalance); } |

服务消费端清除隐式参数

|  |
| --- |
| public class ConsumerContextFilter implements Filter {   @Override  public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {  RpcContext.*getContext*()  .setInvoker(invoker)  .setInvocation(invocation)  .setLocalAddress(NetUtils.*getLocalHost*(), 0)  .setRemoteAddress(invoker.getUrl().getHost(),  invoker.getUrl().getPort());  if (invocation instanceof RpcInvocation) {  ((RpcInvocation) invocation).setInvoker(invoker);  }  try {  // TODO should we clear server context?  RpcContext.*removeServerContext*();  return invoker.invoke(invocation);  } finally {  // TODO removeContext? but we need to save future for RpcContext.getFuture() API. If clear attachments here, attachments will not available when postProcessResult is invoked.  RpcContext.*getContext*().clearAttachments();  }  }   @Override  public Result onResponse(Result result, Invoker<?> invoker, Invocation invocation) {  RpcContext.*getServerContext*().setAttachments(result.getAttachments());  return result;  } } |

## 10.2 服务提供方ContextFilter原理剖析

|  |
| --- |
| @Activate(group = Constants.*PROVIDER*, order = -10000) public class ContextFilter implements Filter {   @Override  public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {  // 1）从invocation对象获取附加属性map  Map<String, String> attachments = invocation.getAttachments();  // 2)不为null则设置到上下文对象中  if (attachments != null) {  attachments = new HashMap<>(attachments);  attachments.remove(Constants.*PATH\_KEY*);  attachments.remove(Constants.*INTERFACE\_KEY*);  attachments.remove(Constants.*GROUP\_KEY*);  attachments.remove(Constants.*VERSION\_KEY*);  attachments.remove(Constants.*DUBBO\_VERSION\_KEY*);  attachments.remove(Constants.*TOKEN\_KEY*);  attachments.remove(Constants.*TIMEOUT\_KEY*);  // Remove async property to avoid being passed to the following invoke chain.  attachments.remove(Constants.*ASYNC\_KEY*);  attachments.remove(Constants.*TAG\_KEY*);  attachments.remove(Constants.*FORCE\_USE\_TAG*);  }  RpcContext.*getContext*()  .setInvoker(invoker)  .setInvocation(invocation) // .setAttachments(attachments) // merged from dubbox  .setLocalAddress(invoker.getUrl().getHost(),  invoker.getUrl().getPort());   // merged from dubbox  // we may already added some attachments into RpcContext before this filter (e.g. in rest protocol)  if (attachments != null) {  if (RpcContext.*getContext*().getAttachments() != null) {  RpcContext.*getContext*().getAttachments().putAll(attachments);  } else {  RpcContext.*getContext*().setAttachments(attachments);  }  }   if (invocation instanceof RpcInvocation) {  ((RpcInvocation) invocation).setInvoker(invoker);  }  try {  return invoker.invoke(invocation);  } finally {  // IMPORTANT! For async scenario, we must remove context from current thread, so we always create a new RpcContext for the next invoke for the same thread.  // 3)清除上下文对象，则附加属性也被回收  RpcContext.*removeContext*();  RpcContext.*removeServerContext*();  }  }   @Override  public Result onResponse(Result result, Invoker<?> invoker, Invocation invocation) {  // pass attachments to result  result.addAttachments(RpcContext.*getServerContext*().getAttachments());  return result;  } } |

# dubbo全链路异步

## 11.1 服务消费端异步调用

UserThread

request

I/O Thread

Server

Provider

RpcContext

Future

1.sayHello

8.set/notify

2.create future

3.request

7.response

4.setFuture

5.getFuture

6.get/wait

Consumer

### 11.1.1 2.7.0版本前的异步调用实现

|  |
| --- |
| public class ApiAsyncConsumer {   public static void main(String[] args) throws ExecutionException, InterruptedException {   // 1.创建引用实例，并设置属性  ReferenceConfig<GreetingService> referenceConfig = new ReferenceConfig<GreetingService>();  referenceConfig.setApplication(new ApplicationConfig("first-dubbo-consumer"));  referenceConfig.setRegistry(new RegistryConfig("zookeeper://127.0.0.1:2181"));  referenceConfig.setInterface(GreetingService.class);  referenceConfig.setVersion("1.0.0");  referenceConfig.setGroup("dubbo");   // 2.设置为异步  referenceConfig.setAsync(true);  // 3.直接返回null  GreetingService greetingService = referenceConfig.get();  System.*out*.println(greetingService.sayHello("world"));   // 4.异步执行回调  Future<String> future = RpcContext.*getContext*().getFuture();  System.*out*.println(future.get());    }  } |

DubboInvoker#doInvoke

|  |
| --- |
| @Override protected Result doInvoke(final Invocation invocation) throws Throwable {  RpcInvocation inv = (RpcInvocation) invocation;  final String methodName = RpcUtils.*getMethodName*(invocation);  inv.setAttachment(Constants.*PATH\_KEY*, getUrl().getPath());  inv.setAttachment(Constants.*VERSION\_KEY*, version);   ExchangeClient currentClient;  if (clients.length == 1) {  currentClient = clients[0];  } else {  currentClient = clients[index.getAndIncrement() % clients.length];  }  try {  // 1.1）是否为异步调用  boolean isAsync = RpcUtils.*isAsync*(getUrl(), invocation);  // 1.2）是否为future异步调用  boolean isAsyncFuture = RpcUtils.*isReturnTypeFuture*(inv);  // 1.3）是否为oneWay，也就是不需要响应结果的请求  boolean isOneway = RpcUtils.*isOneway*(getUrl(), invocation);  // 1.4）超时等待时间  int timeout = getUrl().getMethodParameter(methodName, Constants.*TIMEOUT\_KEY*, Constants.*DEFAULT\_TIMEOUT*);  // 1.5）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) {  // 1.6）异步请求  // 1.6.1）  ResponseFuture future = currentClient.request(inv, timeout);  // For compatibility  // 为了兼容，返回适配器对象  FutureAdapter<Object> futureAdapter = new FutureAdapter<>(future);  RpcContext.*getContext*().setFuture(futureAdapter);  // 1.6.2）异步future  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 {  // 1.7）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);  } } |

|  |  |
| --- | --- |
| DubboInvoker | 1.doInvoke |
| ReferenceCountExchangeClient | 2.request |
| HeaderExchangeClient | 3.request |
| HeaderExchangeChannel | 4.request |
|  |  |
|  |  |

HeaderExchangeChannel#request

|  |
| --- |
| public ResponseFuture request(Object request, int timeout) throws RemotingException {  if (closed) {  throw new RemotingException(this.getLocalAddress(), null, "Failed to send request " + request + ", cause: The channel " + this + " is closed!");  }  // create request.  // 创建request对象  Request req = new Request();  req.setVersion(Version.*getProtocolVersion*());  req.setTwoWay(true);  req.setData(request);  // 创建DefaultFuture对象  DefaultFuture future = DefaultFuture.*newFuture*(channel, req, timeout);  try {  // 发送请求到远端，并返回future  channel.send(req);  } catch (RemotingException e) {  future.cancel();  throw e;  }  return future; } |

DefaultFuture

|  |
| --- |
| // 通道缓存  private static final Map<Long, Channel> *CHANNELS* = new ConcurrentHashMap<>();  // future缓存 private static final Map<Long, DefaultFuture> *FUTURES* = new ConcurrentHashMap<>(); // 超时检查定时器 public static final Timer *TIME\_OUT\_TIMER* = new HashedWheelTimer(  new NamedThreadFactory("dubbo-future-timeout", true),  30,  TimeUnit.*MILLISECONDS*);  lock独占锁，done是该锁的一个条件变量  private final Lock lock = new ReentrantLock(); private final Condition done = lock.newCondition(); |

|  |
| --- |
| public static DefaultFuture newFuture(Channel channel, Request request, int timeout) {  // 5)创建DefaultFuture实例  final DefaultFuture future = new DefaultFuture(channel, request, timeout);  // timeout check  // 6)超时检查  *timeoutCheck*(future);  return future; }  private DefaultFuture(Channel channel, Request request, int timeout) {  // 5.1)保存通信通道  this.channel = channel;  // 5.2)保存请求内容  this.request = request;  // 5.3)保存请求id  this.id = request.getId();  // 5.4)保存请超时时间  this.timeout = timeout > 0 ? timeout : channel.getUrl().getPositiveParameter(Constants.*TIMEOUT\_KEY*, Constants.*DEFAULT\_TIMEOUT*);  // put into waiting map.  // 5.5)把当前对象保存到缓存FUTURES中  *FUTURES*.put(id, this);  // 5.6)把当前通道保存到缓存CHANNELS中  *CHANNELS*.put(id, channel); }  private static void timeoutCheck(DefaultFuture future) {  // 6.1）创建一个任务  TimeoutCheckTask task = new TimeoutCheckTask(future);  // 6.2）定时timeout执行任务  *TIME\_OUT\_TIMER*.newTimeout(task, future.getTimeout(), TimeUnit.*MILLISECONDS*); } |

|  |
| --- |
| private static class TimeoutCheckTask implements TimerTask {   private DefaultFuture future;   TimeoutCheckTask(DefaultFuture future) {  this.future = future;  }   @Override  public void run(Timeout timeout) {  // 6.3)如果future为null或者future任务已经完成，则返回  if (future == null || future.isDone()) {  return;  }  // create exception response.  // 6.4）创建响应对象  Response timeoutResponse = new Response(future.getId());  // set timeout status.  // 6.5)设置超时状态  timeoutResponse.setStatus(future.isSent() ? Response.*SERVER\_TIMEOUT* : Response.*CLIENT\_TIMEOUT*);  timeoutResponse.setErrorMessage(future.getTimeoutMessage(true));  // handle response.  // 6.5)把超时响应信息设置到future内的通道  DefaultFuture.*received*(future.getChannel(), timeoutResponse);   } } |

newFuture()方法的作用，具体来说就是创建一个DefualtFuture对象，并启动一个定时器，然后再超时时间后检查是否已经有响应椰果，如果有则直接返回，否则返会超时信息

|  |
| --- |
| public static void received(Channel channel, Response response) {  try {  // 7）从FUTURES中移除ID对应的future对象  DefaultFuture future = *FUTURES*.remove(response.getId());  // 8）如果future存在，则调用doReceived  if (future != null) {  future.doReceived(response);  } else {  *logger*.warn("The timeout response finally returned at "  + (new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS").format(new Date()))  + ", response " + response  + (channel == null ? "" : ", channel: " + channel.getLocalAddress()  + " -> " + channel.getRemoteAddress()));  }  } finally {  // 9)从缓存CHANNELS中移除请求ID对应的通道  *CHANNELS*.remove(response.getId());  } }  private void doReceived(Response res) {  // 10）获取独占锁  lock.lock();  try {  // 10.1）激活由于调用done的wait()方法被阻塞的一个线程  response = res;  if (done != null) {  done.signal();  }  } finally {  // 10.2）释放锁  lock.unlock();  }  // 11）如何设置了回调，则调用回调  if (callback != null) {  invokeCallback(callback);  } } |

获取执行结果的get()方法如下：

|  |
| --- |
| @Override public Object get() throws RemotingException {  return get(timeout); }  @Override public Object get(int timeout) throws RemotingException {  // 12）如果超时时间<=0，则使用默认值1000  if (timeout <= 0) {  timeout = Constants.*DEFAULT\_TIMEOUT*;  }  // 13）如果future任务还没有完成  if (!isDone()) {  long start = System.*currentTimeMillis*();  lock.lock();  try {  // 任务没有完成则挂起线程timeoutms  while (!isDone()) {  done.await(timeout, TimeUnit.*MILLISECONDS*);  if (isDone() || System.*currentTimeMillis*() - start > timeout) {  break;  }  }  } catch (InterruptedException e) {  throw new RuntimeException(e);  } finally {  lock.unlock();  }  if (!isDone()) {  throw new TimeoutException(sent > 0, channel, getTimeoutMessage(false));  }  }  // 14)任务完成，则直接返回响应结果  return returnFromResponse(); } |

本节一开始讲解了Dubbo异步调用链路流程图，当服务消费端业务线程发起请求后，会创建一个DefaultFuture对象并设置到RpcContext中，然后再启动IO线程发起请求后调用线程就返回了null的结果；当前线程会调用条件变量done的await()方法而挂起；当服务提供端把结果协会调用方之后，调用方线程模型中线程池里的线程会把结果写入DefaultFuture对象内的结果变量中，接着调用条件变量的signal()方法来激活业务线程，然后业务线程就会从get()方法返回响应结果。

|  |
| --- |
| public class ApiAsyncConsumerForCallBack {    public static void main(String[] args) throws InterruptedException {   // 1.创建引用实例  ReferenceConfig<GreetingService> referenceConfig = new ReferenceConfig<>();  referenceConfig.setApplication(new ApplicationConfig("first-dubbo-consumer"));  referenceConfig.setRegistry(new RegistryConfig("zookeeper://127.0.0.1:2181"));  referenceConfig.setInterface(GreetingService.class);   referenceConfig.setTimeout(5000);  referenceConfig.setVersion("1.0.0");  referenceConfig.setGroup("dubbo");   // 2. 设置异步  referenceConfig.setAsync(true);   // 3. 直接返回null  GreetingService greetingService = referenceConfig.get();  System.*out*.println(greetingService.sayHello("world"));   // 4. 异步执行回调   ((FutureAdapter) RpcContext.*getContext*().getFuture()).getFuture().setCallback(new ResponseCallback() {  @Override  public void done(Object response) {  System.*out*.println("result:" + response);  }   @Override  public void caught(Throwable exception) {  System.*out*.println("error:" + exception.getLocalizedMessage());  }  });   Thread.*currentThread*().join();   }   } |

|  |
| --- |
| public void setCallback(ResponseCallback callback) {  // 18）如果响应结果已经有了，则直接调用回调callback  if (isDone()) {  invokeCallback(callback);  } else {  // 19）设置callback  boolean isdone = false;  lock.lock();  try {  // 如果没有结果，则设置callback  if (!isDone()) {  this.callback = callback;  } else {  isdone = true;  }  } finally {  lock.unlock();  }  if (isdone) {  invokeCallback(callback);  }  } }  private void invokeCallback(ResponseCallback c) {  ResponseCallback callbackCopy = c;  if (callbackCopy == null) {  throw new NullPointerException("callback cannot be null.");  }  Response res = response;  if (res == null) {  throw new IllegalStateException("response cannot be null. url:" + channel.getUrl());  }  // 20）状态码为20，则调用回调的done()方法  if (res.getStatus() == Response.*OK*) {  try {  callbackCopy.done(res.getResult());  } catch (Exception e) {  *logger*.error("callback invoke error .result:" + res.getResult() + ",url:" + channel.getUrl(), e);  }  } else if (res.getStatus() == Response.*CLIENT\_TIMEOUT* || res.getStatus() == Response.*SERVER\_TIMEOUT*) {  // 21）超时则调用caught方法  try {  TimeoutException te = new TimeoutException(res.getStatus() == Response.*SERVER\_TIMEOUT*, channel, res.getErrorMessage());  callbackCopy.caught(te);  } catch (Exception e) {  *logger*.error("callback invoke error ,url:" + channel.getUrl(), e);  }  } else {  try {  RuntimeException re = new RuntimeException(res.getErrorMessage());  callbackCopy.caught(re);  } catch (Exception e) {  *logger*.error("callback invoke error ,url:" + channel.getUrl(), e);  }  } } |

### 11.1.2 2.7.0版本提供的异步调用实现

|  |
| --- |
| public class ApiAsyncConsumerForCompletableFuture2 {    public static void main(String[] args) throws InterruptedException {    // 1.创建服务引用对象，并设置数据  ReferenceConfig<GreetingService> referenceConfig = new ReferenceConfig<>();  referenceConfig.setApplication(new ApplicationConfig("first-dubbo-consumer"));   referenceConfig.setRegistry(new RegistryConfig("zookeeper://127.0.0.1:2181"));  referenceConfig.setInterface(GreetingService.class);  referenceConfig.setTimeout(30000);  referenceConfig.setVersion("1.0.0");  referenceConfig.setGroup("dubbo");   // 2.设置为异步  referenceConfig.setAsync(true);   // 3.直接返回null  GreetingService greetingService = referenceConfig.get();  System.*out*.println(greetingService.sayHello("world"));   // 4. 异步执行回调  CompletableFuture<String> future = RpcContext.*getContext*().getCompletableFuture();  future.whenComplete((v,t)->{  if(t != null){  t.printStackTrace();  }else{  System.*out*.println(v);  }  });   System.*out*.println("over");  Thread.*currentThread*().join();   }  } |

|  |
| --- |
| public class FutureAdapter<V> extends CompletableFuture<V> {   private final ResponseFuture future;  private CompletableFuture<Result> resultFuture;   public FutureAdapter(ResponseFuture future) {  // 5)赋值与初始化  this.future = future;  this.resultFuture = new CompletableFuture<>();  // 6)设置DefaultFuture实例回调  future.setCallback(new ResponseCallback() {  @Override  public void done(Object response) {  // 6.1) 把响应结果设置到CompletableFuture  Result result = (Result) response;  FutureAdapter.this.resultFuture.complete(result);  V value = null;  try {  value = (V) result.recreate();  } catch (Throwable t) {  FutureAdapter.this.completeExceptionally(t);  }  FutureAdapter.this.complete(value);  }  // 6.2) 把异常设置到CompletableFuture中  @Override  public void caught(Throwable exception) {  FutureAdapter.this.completeExceptionally(exception);  }  });  }   public ResponseFuture getFuture() {  return future;  }   @Override  public boolean cancel(boolean mayInterruptIfRunning) {  return false;  }   @Override  public boolean isCancelled() {  return false;  }   @Override  public boolean isDone() {  return super.isDone();  }   @Override  @SuppressWarnings("unchecked")  public V get() throws InterruptedException, ExecutionException {  try {  return super.get();  } catch (ExecutionException | InterruptedException e) {  throw e;  } catch (Throwable e) {  throw new RpcException(e);  }  }   @Override  @SuppressWarnings("unchecked")  public V get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException, TimeoutException {  try {  return super.get(timeout, unit);  } catch (TimeoutException | ExecutionException | InterruptedException e) {  throw e;  } catch (Throwable e) {  throw new RpcException(e);  }  }   */\*\*  \* FIXME  \* This method has no need open to the the end user.  \* Mostly user use RpcContext.getFuture() to refer the instance of this class, so the user will get a CompletableFuture, this method will rarely be noticed.  \*  \** ***@return*** *\*/* public CompletableFuture<Result> getResultFuture() {  return resultFuture;  }  } |

## 11.2 服务提供端异步调用

UserThread

request

IO Thread

RpcContext

Future

1.sayHello

5.getFuture

4.setFuture

2.create future

6.get/wait

8.set/notify

IO 线程

Dubbo ThreadPool

BizThread

Consumer

Provider

3.request

7.response

### 11.2.1 基于定义CompletableFuture签名的接口实现异步执行

|  |
| --- |
| public class GrettingServiceAsyncImpl implements GrettingServiceAsync {    // 1.创建业务自定义线程池  private final ThreadPoolExecutor bizThreadpool = new ThreadPoolExecutor(8, 16, 1, TimeUnit.*MINUTES*,  new SynchronousQueue<Runnable>(), new NamedThreadFactory("biz-thread-pool"), new ThreadPoolExecutor.CallerRunsPolicy());    // 2. 创建服务处理接口，返回值为CompletableFuture  public CompletableFuture<String> sayHello(final String name) {  // 2.1 为supplyAsync提供自定义线程池bizThreadpool，避免使用JDK共用线程池(ForkJoinPool.commonPool())  // 使用CompletableFuture.supplyAsync让服务处理异步进行处理  // 保存当前线程的上下文   RpcContext context = RpcContext.*getContext*();   return CompletableFuture.*supplyAsync*(()->{  try {  Thread.*sleep*(2000);  } catch (InterruptedException e) {  e.printStackTrace();  }  System.*out*.println("async return ");  return "Hello "+name+" "+ context.getAttachment("company");  },bizThreadpool);  }  } |

HeaderExchangeHandler

|  |
| --- |
| void handleRequest(final ExchangeChannel channel, Request req) throws RemotingException {  //   Response res = new Response(req.getId(), req.getVersion());  if (req.isBroken()) {  Object data = req.getData();   String msg;  if (data == null) {  msg = null;  } else if (data instanceof Throwable) {  msg = StringUtils.*toString*((Throwable) data);  } else {  msg = data.toString();  }  res.setErrorMessage("Fail to decode request due to: " + msg);  res.setStatus(Response.*BAD\_REQUEST*);   channel.send(res);  return;  }  // find handler by message class.  Object msg = req.getData();  try {  // handle data.  // 3）调用DubboProtocol的reply方法  CompletableFuture<Object> future = handler.reply(channel, msg);  // 4）  if (future.isDone()) {  res.setStatus(Response.*OK*);  res.setResult(future.get());  channel.send(res);  return;  }  // 5）否则等返回结果后异步调用回调  future.whenComplete((result, t) -> {  try {  if (t == null) {  res.setStatus(Response.*OK*);  res.setResult(result);  } else {  res.setStatus(Response.*SERVICE\_ERROR*);  res.setErrorMessage(StringUtils.*toString*(t));  }  channel.send(res);  } catch (RemotingException e) {  *logger*.warn("Send result to consumer failed, channel is " + channel + ", msg is " + e);  } finally {  // HeaderExchangeChannel.removeChannelIfDisconnected(channel);  }  });  } catch (Throwable e) {  res.setStatus(Response.*SERVICE\_ERROR*);  res.setErrorMessage(StringUtils.*toString*(e));  channel.send(res);  } } |

DubboProtocol

|  |
| --- |
| private ExchangeHandler requestHandler = new ExchangeHandlerAdapter() {   @Override  public CompletableFuture<Object> reply(ExchangeChannel channel, Object message) throws RemotingException {   if (!(message instanceof Invocation)) {  throw new RemotingException(channel, "Unsupported request: "  + (message == null ? null : (message.getClass().getName() + ": " + message))  + ", channel: consumer: " + channel.getRemoteAddress() + " --> provider: " + channel.getLocalAddress());  }   Invocation inv = (Invocation) message;  Invoker<?> invoker = getInvoker(channel, inv);  // need to consider backward-compatibility if it's a callback  if (Boolean.*TRUE*.toString().equals(inv.getAttachments().get(*IS\_CALLBACK\_SERVICE\_INVOKE*))) {  String methodsStr = invoker.getUrl().getParameters().get("methods");  boolean hasMethod = false;  if (methodsStr == null || !methodsStr.contains(",")) {  hasMethod = inv.getMethodName().equals(methodsStr);  } else {  String[] methods = methodsStr.split(",");  for (String method : methods) {  if (inv.getMethodName().equals(method)) {  hasMethod = true;  break;  }  }  }  if (!hasMethod) {  logger.warn(new IllegalStateException("The methodName " + inv.getMethodName()  + " not found in callback service interface ,invoke will be ignored."  + " please update the api interface. url is:"  + invoker.getUrl()) + " ,invocation is :" + inv);  return null;  }  }  RpcContext rpcContext = RpcContext.*getContext*();  rpcContext.setRemoteAddress(channel.getRemoteAddress());  // 3.1）执行invoker调用链  Result result = invoker.invoke(inv);  // 3.2）写回结果  // 如果结果为AsyncRpcResult，说明是服务提供方的异步执行  if (result instanceof AsyncRpcResult) {  return ((AsyncRpcResult) result).getResultFuture().thenApply(r -> (Object) r);   } else {  // 否则为同步执行，则转换结果为CompletableFuture  return CompletableFuture.*completedFuture*(result);  }  }   @Override  public void received(Channel channel, Object message) throws RemotingException {  if (message instanceof Invocation) {  reply((ExchangeChannel) channel, message);   } else {  super.received(channel, message);  }  }   @Override  public void connected(Channel channel) throws RemotingException {  invoke(channel, Constants.*ON\_CONNECT\_KEY*);  }   @Override  public void disconnected(Channel channel) throws RemotingException {  if (logger.isDebugEnabled()) {  logger.debug("disconnected from " + channel.getRemoteAddress() + ",url:" + channel.getUrl());  }  invoke(channel, Constants.*ON\_DISCONNECT\_KEY*);  }   private void invoke(Channel channel, String methodKey) {  Invocation invocation = createInvocation(channel, channel.getUrl(), methodKey);  if (invocation != null) {  try {  received(channel, invocation);  } catch (Throwable t) {  logger.warn("Failed to invoke event method " + invocation.getMethodName() + "(), cause: " + t.getMessage(), t);  }  }  }   private Invocation createInvocation(Channel channel, URL url, String methodKey) {  String method = url.getParameter(methodKey);  if (method == null || method.length() == 0) {  return null;  }   RpcInvocation invocation = new RpcInvocation(method, new Class<?>[0], new Object[0]);  invocation.setAttachment(Constants.*PATH\_KEY*, url.getPath());  invocation.setAttachment(Constants.*GROUP\_KEY*, url.getParameter(Constants.*GROUP\_KEY*));  invocation.setAttachment(Constants.*INTERFACE\_KEY*, url.getParameter(Constants.*INTERFACE\_KEY*));  invocation.setAttachment(Constants.*VERSION\_KEY*, url.getParameter(Constants.*VERSION\_KEY*));  if (url.getParameter(Constants.*STUB\_EVENT\_KEY*, false)) {  invocation.setAttachment(Constants.*STUB\_EVENT\_KEY*, Boolean.*TRUE*.toString());  }   return invocation;  } }; |

AbstractProxyInvoker

|  |
| --- |
| public Result invoke(Invocation invocation) throws RpcException {  // 6)获取上下文对象  RpcContext rpcContext = RpcContext.*getContext*();  try {  // 6.1)具体执行本地服务调用  Object obj = doInvoke(proxy, invocation.getMethodName(), invocation.getParameterTypes(), invocation.getArguments());  // 6.2)返回结果  // 如果future\_returntype为true，则返回结果是CompletableFuture  // 类型  if (RpcUtils.*isReturnTypeFuture*(invocation)) {  return new AsyncRpcResult((CompletableFuture<Object>) obj);  } else if (rpcContext.isAsyncStarted()) { // ignore obj in case of RpcContext.startAsync()? always rely on user to write back.  // 使用RpcContext.startAsync()开启异步执行  return new AsyncRpcResult(((AsyncContextImpl)(rpcContext.getAsyncContext())).getInternalFuture());  } else {  // 同步执行时候  return new RpcResult(obj);  }  } catch (InvocationTargetException e) {  // TODO async throw exception before async thread write back, should stop asyncContext  if (rpcContext.isAsyncStarted() && !rpcContext.stopAsync()) {  logger.error("Provider async started, but got an exception from the original method, cannot write the exception back to consumer because an async result may have returned the new thread.", e);  }  return new RpcResult(e.getTargetException());  } catch (Throwable e) {  throw new RpcException("Failed to invoke remote proxy method " + invocation.getMethodName() + " to " + getUrl() + ", cause: " + e.getMessage(), e);  } } |

### 11.2.2 使用AsyncContext实现异步执行

|  |
| --- |
| public class GrettingServiceAsyncContextImpl implements GrettingServiceRpcContext {    // 1.创建业务自定义线程池  private final ThreadPoolExecutor bizThreadPool = new ThreadPoolExecutor(8, 16, 1, TimeUnit.*MINUTES*,  new SynchronousQueue<>(), new NamedThreadFactory("biz-thread-pool"),  new ThreadPoolExecutor.CallerRunsPolicy());    // 2.创建服务处理接口，返回值为CompletableFuture  @Override  public String sayHello(String name) {   // 2.1开启异步  final AsyncContext asyncContext = RpcContext.*startAsync*();  bizThreadPool.execute(() -> {  // 2.2 如果要使用上下文，则必须要放在第一句执行  asyncContext.signalContextSwitch();   try {  Thread.*sleep*(500);  } catch (InterruptedException e) {  e.printStackTrace();  }   // 2.3 写回响应  asyncContext.write("Hello "+name+" "+RpcContext.*getContext*().getAttachment("company"));   });  return null;  } } |

RpcContext.startAsync()方法：

|  |
| --- |
| public static AsyncContext startAsync() throws IllegalStateException {  // 2.1.1)获取当前线程的上下文对象  RpcContext currentContext = *getContext*();  // 2.1.2)为当前线程的上下文创建AsyncContextImpl实现  if (currentContext.asyncContext == null) {  currentContext.asyncContext = new AsyncContextImpl();  }  // 2.1.3）启动异步上下文，并返回  currentContext.asyncContext.start();  return currentContext.asyncContext; } |

## 11.3 异步调用与执行引入的新问题

|  |
| --- |
| public interface Filter {   */\*\*  \* do invoke filter.  \* <p>  \* <code>  \* // before filter  \* Result result = invoker.invoke(invocation);  \* // after filter  \* return result;  \* </code>  \*  \** ***@param*** *invoker service  \** ***@param*** *invocation invocation.  \** ***@return*** *invoke result.  \** ***@throws*** *RpcException  \** ***@see*** *org.apache.dubbo.rpc.Invoker#invoke(Invocation)  \*/* Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException;   default Result onResponse(Result result, Invoker<?> invoker, Invocation invocation) {  return result;  }  } |

# dubbo全链路异步

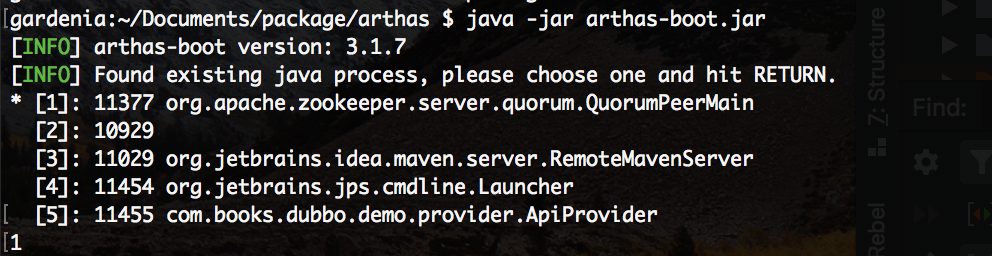
# dubbo全链路异步

# dubbo实战

下载 <https://alibaba.github.io/arthas/arhtas-boot.jar>

java -jar arthas-boot.jar

可以看到java的相关进程列表



输入编号选择进入某个进程中，比如输入5则进入11455进程中

jad org.apache.dubbo.rpc.Protocol$Adaptive

可以输出源码