**Dubbo**

# Zookeeper

## ZookeeperDynamicConfiguration

|  |
| --- |
| ZookeeperDynamicConfiguration(URL url, ZookeeperTransporter zookeeperTransporter) {  **this**.**url** = url;  **rootPath** = ***PATH\_SEPARATOR*** + url.getParameter(***CONFIG\_NAMESPACE\_KEY***, ***DEFAULT\_GROUP***) + **"/config"**;   **initializedLatch** = **new** CountDownLatch(1);  **this**.**cacheListener** = **new** CacheListener(**rootPath**, **initializedLatch**);  **this**.**executor** = Executors.*newFixedThreadPool*(1, **new** NamedThreadFactory(**this**.getClass().getSimpleName(), **true**));   **// 连接zookeeper**  **zkClient** = zookeeperTransporter.connect(url);  **zkClient**.addDataListener(**rootPath**, **cacheListener**, **executor**);  **try** {  *// Wait for connection* **this**.**initializedLatch**.await();  } **catch** (InterruptedException e) {  ***logger***.warn(**"Failed to build local cache for config center (zookeeper)."** + url);  } } |

## Zookeeper连接

双重检查

|  |
| --- |
| **@Override public ZookeeperClient connect(URL url) {  ZookeeperClient zookeeperClient;  List<String> addressList = getURLBackupAddress(url);  *// The field define the zookeeper server , including protocol, host, port, username, password* if ((zookeeperClient = fetchAndUpdateZookeeperClientCache(addressList)) != null && zookeeperClient.isConnected()) {  *logger*.info("find valid zookeeper client from the cache for address: " + url);  return zookeeperClient;  }  *// avoid creating too many connections， so add lock* synchronized (zookeeperClientMap) {  if ((zookeeperClient = fetchAndUpdateZookeeperClientCache(addressList)) != null && zookeeperClient.isConnected()) {  *logger*.info("find valid zookeeper client from the cache for address: " + url);  return zookeeperClient;  }   zookeeperClient = createZookeeperClient(toClientURL(url));  *logger*.info("No valid zookeeper client found from cache, therefore create a new client for url. " + url);  writeToClientMap(addressList, zookeeperClient);  }  return zookeeperClient; }** |

### ClientCnxn

#### SendThread

用来连接zookeeper的守护线程

|  |
| --- |
| @Override **public void run() {  clientCnxnSocket.introduce(this,sessionId);  clientCnxnSocket.updateNow();  clientCnxnSocket.updateLastSendAndHeard();  int to;  long lastPingRwServer = System.*currentTimeMillis*();  final int MAX\_SEND\_PING\_INTERVAL = 10000; *//10 seconds* while (state.isAlive()) {  try {  if (!clientCnxnSocket.isConnected()) {  if(!isFirstConnect){  try {  Thread.*sleep*(r.nextInt(1000));  } catch (InterruptedException e) {  *LOG*.warn("Unexpected exception", e);  }  }  *// don't re-establish connection if we are closing* if (closing || !state.isAlive()) {  break;  }  startConnect();  clientCnxnSocket.updateLastSendAndHeard();  }   if (state.isConnected()) {  *// determine whether we need to send an AuthFailed event.* if (zooKeeperSaslClient != null) {  boolean sendAuthEvent = false;  if (zooKeeperSaslClient.getSaslState() == ZooKeeperSaslClient.SaslState.*INITIAL*) {  try {  zooKeeperSaslClient.initialize(ClientCnxn.this);  } catch (SaslException e) {  *LOG*.error("SASL authentication with Zookeeper Quorum member failed: " + e);  state = States.*AUTH\_FAILED*;  sendAuthEvent = true;  }  }  KeeperState authState = zooKeeperSaslClient.getKeeperState();  if (authState != null) {  if (authState == KeeperState.*AuthFailed*) {  *// An authentication error occurred during authentication with the Zookeeper Server.* state = States.*AUTH\_FAILED*;  sendAuthEvent = true;  } else {  if (authState == KeeperState.*SaslAuthenticated*) {  sendAuthEvent = true;  }  }  }   if (sendAuthEvent == true) {  eventThread.queueEvent(new WatchedEvent(  Watcher.Event.EventType.*None*,  authState,null));  }  }  to = readTimeout - clientCnxnSocket.getIdleRecv();  } else {  to = connectTimeout - clientCnxnSocket.getIdleRecv();  }    if (to <= 0) {  String warnInfo;  warnInfo = "Client session timed out, have not heard from server in "  + clientCnxnSocket.getIdleRecv()  + "ms"  + " for sessionid 0x"  + Long.*toHexString*(sessionId);  *LOG*.warn(warnInfo);  throw new SessionTimeoutException(warnInfo);  }  if (state.isConnected()) {  *//1000(1 second) is to prevent race condition missing to send the second ping  //also make sure not to send too many pings when readTimeout is small* int timeToNextPing = readTimeout / 2 - clientCnxnSocket.getIdleSend() -   ((clientCnxnSocket.getIdleSend() > 1000) ? 1000 : 0);  *//send a ping request either time is due or no packet sent out within MAX\_SEND\_PING\_INTERVAL* if (timeToNextPing <= 0 || clientCnxnSocket.getIdleSend() > MAX\_SEND\_PING\_INTERVAL) {  sendPing();  clientCnxnSocket.updateLastSend();  } else {  if (timeToNextPing < to) {  to = timeToNextPing;  }  }  }   *// If we are in read-only mode, seek for read/write server* if (state == States.*CONNECTEDREADONLY*) {  long now = System.*currentTimeMillis*();  int idlePingRwServer = (int) (now - lastPingRwServer);  if (idlePingRwServer >= pingRwTimeout) {  lastPingRwServer = now;  idlePingRwServer = 0;  pingRwTimeout =  Math.*min*(2\*pingRwTimeout, *maxPingRwTimeout*);  pingRwServer();  }  to = Math.*min*(to, pingRwTimeout - idlePingRwServer);  }   clientCnxnSocket.doTransport(to, pendingQueue, outgoingQueue, ClientCnxn.this);  } catch (Throwable e) {  if (closing) {  if (*LOG*.isDebugEnabled()) {  *// closing so this is expected  LOG*.debug("An exception was thrown while closing send thread for session 0x"  + Long.*toHexString*(getSessionId())  + " : " + e.getMessage());  }  break;  } else {  *// this is ugly, you have a better way speak up* if (e instanceof SessionExpiredException) {  *LOG*.info(e.getMessage() + ", closing socket connection");  } else if (e instanceof SessionTimeoutException) {  *LOG*.info(e.getMessage() + *RETRY\_CONN\_MSG*);  } else if (e instanceof EndOfStreamException) {  *LOG*.info(e.getMessage() + *RETRY\_CONN\_MSG*);  } else if (e instanceof RWServerFoundException) {  *LOG*.info(e.getMessage());  } else {  *LOG*.warn(  "Session 0x"  + Long.*toHexString*(getSessionId())  + " for server "  + clientCnxnSocket.getRemoteSocketAddress()  + ", unexpected error"  + *RETRY\_CONN\_MSG*, e);  }  cleanup();  if (state.isAlive()) {  eventThread.queueEvent(new WatchedEvent(  Event.EventType.*None*,  Event.KeeperState.*Disconnected*,  null));  }  clientCnxnSocket.updateNow();  clientCnxnSocket.updateLastSendAndHeard();  }  }  }  cleanup();  clientCnxnSocket.close();  if (state.isAlive()) {  eventThread.queueEvent(new WatchedEvent(Event.EventType.*None*,  Event.KeeperState.*Disconnected*, null));  }  ZooTrace.*logTraceMessage*(*LOG*, ZooTrace.*getTextTraceLevel*(),  "SendThread exited loop for session: 0x"  + Long.*toHexString*(getSessionId())); }** |

dubbo://192.168.4.17:20880/com.order.service.OrderService?anyhost=true&application=order-service&bean.name=com.order.service.OrderService&bind.ip=192.168.4.17&bind.port=20880&deprecated=false&dubbo=2.0.2&dynamic=true&generic=false&interface=com.order.service.OrderService&methods=createOrder&pid=149960&register=true&release=2.7.3&revision=1.0-SNAPSHOT&side=provider&timestamp=1579319384152

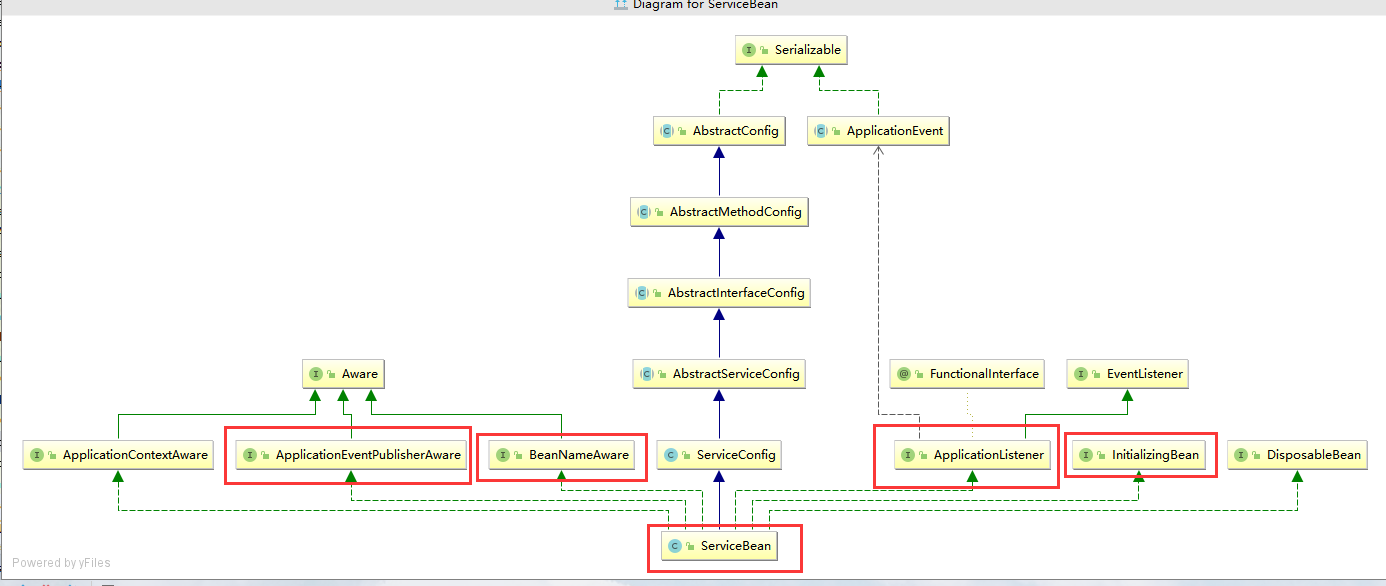
# dubbo服务的暴露

## ServiceBean

### 用途

作为<dubbo:service>标签的bean

### 类图



### 事件发布(用来启动服务)

### 代码

|  |
| --- |
| @Override **public void** onApplicationEvent(ContextRefreshedEvent event) {  **if** (!isExported() && !isUnexported()) {  **if** (***logger***.isInfoEnabled()) {  ***logger***.info(**"The service ready on spring started. service: "** + getInterface());  }  export();  } } |

### 检查配置

|  |
| --- |
| **public void checkAndUpdateSubConfigs() {  *// Use default configs defined explicitly on global configs* completeCompoundConfigs();  *// Config Center should always being started first.* startConfigCenter();  checkDefault();  checkProtocol();  checkApplication();  *// if protocol is not injvm checkRegistry* if (!isOnlyInJvm()) {**  **// 连接注册中心  checkRegistry();  }  this.refresh();  checkMetadataReport();   if (StringUtils.*isEmpty*(interfaceName)) {  throw new IllegalStateException("<dubbo:service interface=\"\" /> interface not allow null!");  }   if (ref instanceof GenericService) {  interfaceClass = GenericService.class;  if (StringUtils.*isEmpty*(generic)) {  generic = Boolean.*TRUE*.toString();  }  } else {  try {  interfaceClass = Class.*forName*(interfaceName, true, Thread.*currentThread*()  .getContextClassLoader());  } catch (ClassNotFoundException e) {  throw new IllegalStateException(e.getMessage(), e);  }  checkInterfaceAndMethods(interfaceClass, methods);  checkRef();  generic = Boolean.*FALSE*.toString();  }  if (local != null) {  if ("true".equals(local)) {  local = interfaceName + "Local";  }  Class<?> localClass;  try {  localClass = ClassUtils.*forNameWithThreadContextClassLoader*(local);  } catch (ClassNotFoundException e) {  throw new IllegalStateException(e.getMessage(), e);  }  if (!interfaceClass.isAssignableFrom(localClass)) {  throw new IllegalStateException("The local implementation class " + localClass.getName() + " not implement interface " + interfaceName);  }  }  if (stub != null) {  if ("true".equals(stub)) {  stub = interfaceName + "Stub";  }  Class<?> stubClass;  try {  stubClass = ClassUtils.*forNameWithThreadContextClassLoader*(stub);  } catch (ClassNotFoundException e) {  throw new IllegalStateException(e.getMessage(), e);  }  if (!interfaceClass.isAssignableFrom(stubClass)) {  throw new IllegalStateException("The stub implementation class " + stubClass.getName() + " not implement interface " + interfaceName);  }  }  checkStubAndLocal(interfaceClass);  checkMock(interfaceClass); }** |

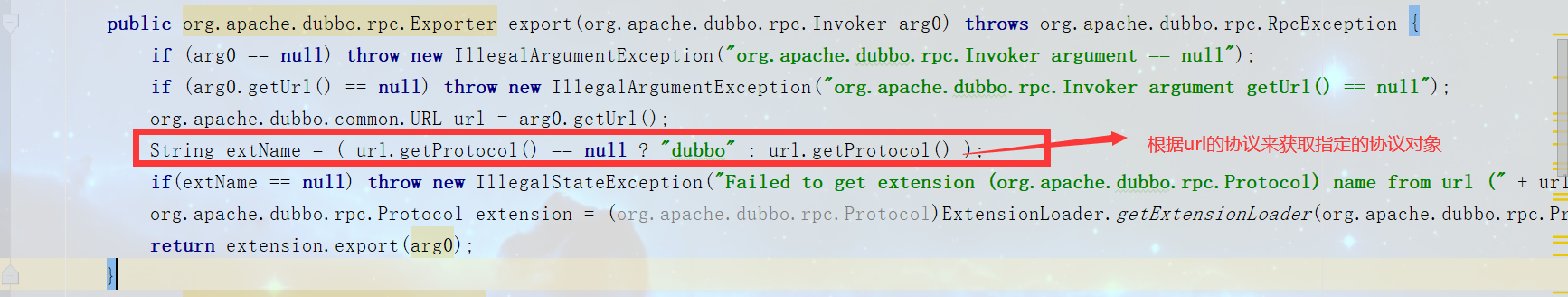
## 服务的注册

### ZookeeperRegistry

|  |
| --- |
| **@Override public void doRegister(URL url) {  try {**  **// 创建节点  zkClient.create(toUrlPath(url), url.getParameter(*DYNAMIC\_KEY*, true));  } catch (Throwable e) {  throw new RpcException("Failed to register " + url + " to zookeeper " + getUrl() + ", cause: " + e.getMessage(), e);  } }** |

## Dubbo服务端口的暴露

在获取协议的时候，使用了动态匹配，这个是动态生成的代码



### DubboProtocol

|  |
| --- |
| **private ExchangeServer createServer(URL url) {  url = URLBuilder.*from*(url)  *// send readonly event when server closes, it's enabled by default* .addParameterIfAbsent(*CHANNEL\_READONLYEVENT\_SENT\_KEY*, Boolean.*TRUE*.toString())  *// enable heartbeat by default* .addParameterIfAbsent(*HEARTBEAT\_KEY*, String.*valueOf*(*DEFAULT\_HEARTBEAT*))  .addParameter(*CODEC\_KEY*, DubboCodec.*NAME*)  .build();  String str = url.getParameter(*SERVER\_KEY*, *DEFAULT\_REMOTING\_SERVER*);   if (str != null && str.length() > 0 && !ExtensionLoader.*getExtensionLoader*(Transporter.class).hasExtension(str)) {  throw new RpcException("Unsupported server type: " + str + ", url: " + url);  }   ExchangeServer server;  try {  server = Exchangers.*bind*(url, requestHandler);  } catch (RemotingException e) {  throw new RpcException("Fail to start server(url: " + url + ") " + e.getMessage(), e);  }   str = url.getParameter(*CLIENT\_KEY*);  if (str != null && str.length() > 0) {  Set<String> supportedTypes = ExtensionLoader.*getExtensionLoader*(Transporter.class).getSupportedExtensions();  if (!supportedTypes.contains(str)) {  throw new RpcException("Unsupported client type: " + str);  }  }   return server; }** |

### org.apache.dubbo.remoting.transport.netty4.NettyServer

|  |
| --- |
| **protected void doOpen() throws Throwable {  bootstrap = new ServerBootstrap();   bossGroup = new NioEventLoopGroup(1, new DefaultThreadFactory("NettyServerBoss", true));  workerGroup = new NioEventLoopGroup(getUrl().getPositiveParameter(*IO\_THREADS\_KEY*, Constants.*DEFAULT\_IO\_THREADS*),  new DefaultThreadFactory("NettyServerWorker", true));   final NettyServerHandler nettyServerHandler = new NettyServerHandler(getUrl(), this);  channels = nettyServerHandler.getChannels();   bootstrap.group(bossGroup, workerGroup)  .channel(NioServerSocketChannel.class)  .childOption(ChannelOption.*TCP\_NODELAY*, Boolean.*TRUE*)  .childOption(ChannelOption.*SO\_REUSEADDR*, Boolean.*TRUE*)  .childOption(ChannelOption.*ALLOCATOR*, PooledByteBufAllocator.*DEFAULT*)  .childHandler(new ChannelInitializer<NioSocketChannel>() {  @Override  protected void initChannel(NioSocketChannel ch) throws Exception {  *// FIXME: should we use getTimeout()?* int idleTimeout = UrlUtils.*getIdleTimeout*(getUrl());  NettyCodecAdapter adapter = new NettyCodecAdapter(getCodec(), getUrl(), NettyServer.this);  ch.pipeline()*//.addLast("logging",new LoggingHandler(LogLevel.INFO))//for debug* .addLast("decoder", adapter.getDecoder())  .addLast("encoder", adapter.getEncoder())  .addLast("server-idle-handler", new IdleStateHandler(0, 0, idleTimeout, *MILLISECONDS*))  .addLast("handler", nettyServerHandler);  }  });  *// bind* ChannelFuture channelFuture = bootstrap.bind(getBindAddress());  channelFuture.syncUninterruptibly();  channel = channelFuture.channel();  }** |

# 服务调用

## FailoverClusterInvoker

### 介绍

集群默认的容错模式:失败自动切换

### 远程调用

|  |
| --- |
| **@Override @SuppressWarnings({"unchecked", "rawtypes"}) public Result doInvoke(Invocation invocation, final List<Invoker<T>> invokers, LoadBalance loadbalance) throws RpcException {  List<Invoker<T>> copyInvokers = invokers;  checkInvokers(copyInvokers, invocation);  String methodName = RpcUtils.*getMethodName*(invocation);**  **// 重试的次数  int len = getUrl().getMethodParameter(methodName, *RETRIES\_KEY*, *DEFAULT\_RETRIES*) + 1;  if (len <= 0) {  len = 1;  }  *// retry loop.* 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.* if (i > 0) {  checkWhetherDestroyed();  copyInvokers = list(invocation);  *// check again* checkInvokers(copyInvokers, invocation);  }  Invoker<T> invoker = select(loadbalance, invocation, copyInvokers, invoked);  invoked.add(invoker);  RpcContext.*getContext*().setInvokers((List) invoked);  try {  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); }** |

### 统一方法的调用

不管是什么方法的调用统一调用invoker中的invoke方法。

## 为服务接口生成代理

入口为ReferenceConfig的getObject()方法

|  |
| --- |
| @SuppressWarnings({**"unchecked"**, **"rawtypes"**, **"deprecation"**}) **private** T createProxy(Map<String, String> map) {  **if** (shouldJvmRefer(map)) {  URL url = **new** URL(***LOCAL\_PROTOCOL***, ***LOCALHOST\_VALUE***, 0, **interfaceClass**.getName()).addParameters(map);  **invoker** = ***REF\_PROTOCOL***.refer(**interfaceClass**, url);  **if** (***logger***.isInfoEnabled()) {  ***logger***.info(**"Using injvm service "** + **interfaceClass**.getName());  }  } **else** {  **urls**.clear(); *// reference retry init will add url to urls, lead to OOM* **if** (**url** != **null** && **url**.length() > 0) { *// user specified URL, could be peer-to-peer address, or register center's address.* String[] us = ***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** (***REGISTRY\_PROTOCOL***.equals(url.getProtocol())) {  **urls**.add(url.addParameterAndEncoded(***REFER\_KEY***, StringUtils.*toQueryString*(map)));  } **else** {  **urls**.add(ClusterUtils.*mergeUrl*(url, map));  }  }  }  } **else** { *// assemble URL from register center's configuration  // if protocols not injvm checkRegistry* **if** (!***LOCAL\_PROTOCOL***.equalsIgnoreCase(getProtocol())){  checkRegistry();  List<URL> us = loadRegistries(**false**);  **if** (CollectionUtils.*isNotEmpty*(us)) {  **for** (URL u : us) {  URL monitorUrl = loadMonitor(u);  **if** (monitorUrl != **null**) {  map.put(***MONITOR\_KEY***, URL.*encode*(monitorUrl.toFullString()));  }  **urls**.add(u.addParameterAndEncoded(***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."**);  }  }  }   **if** (**urls**.size() == 1) {  **invoker** = ***REF\_PROTOCOL***.refer(**interfaceClass**, **urls**.get(0));  } **else** {  List<Invoker<?>> invokers = **new** ArrayList<Invoker<?>>();  URL registryURL = **null**;  **for** (URL url : **urls**) {  invokers.add(***REF\_PROTOCOL***.refer(**interfaceClass**, url));  **if** (***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(***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));  }  }  }   **if** (shouldCheck() && !**invoker**.isAvailable()) {  **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(***CONSUMER\_PROTOCOL***, map.remove(***REGISTER\_IP\_KEY***), 0, map.get(***INTERFACE\_KEY***), map);  metadataReportService.publishConsumer(consumerURL);  }  *// create service proxy* **return** (T) ***PROXY\_FACTORY***.getProxy(**invoker**); } |

# SPI

## ExtensionLoader

### 用途

扩展类的加载

### 代码分析

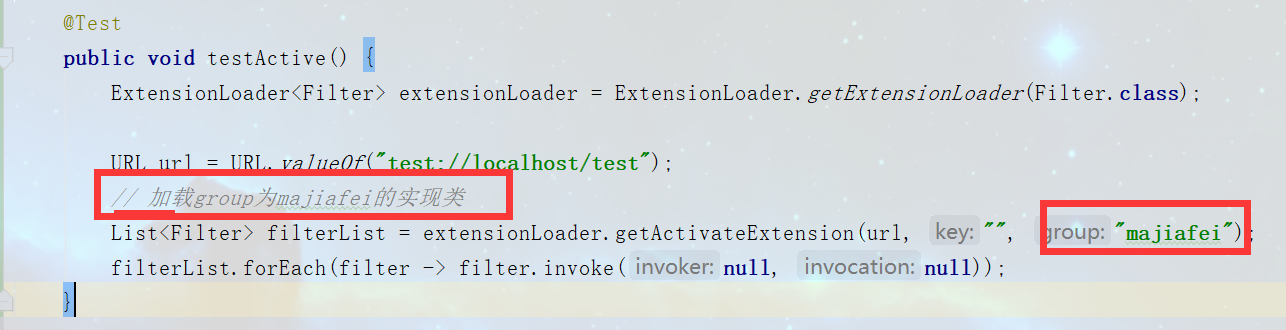
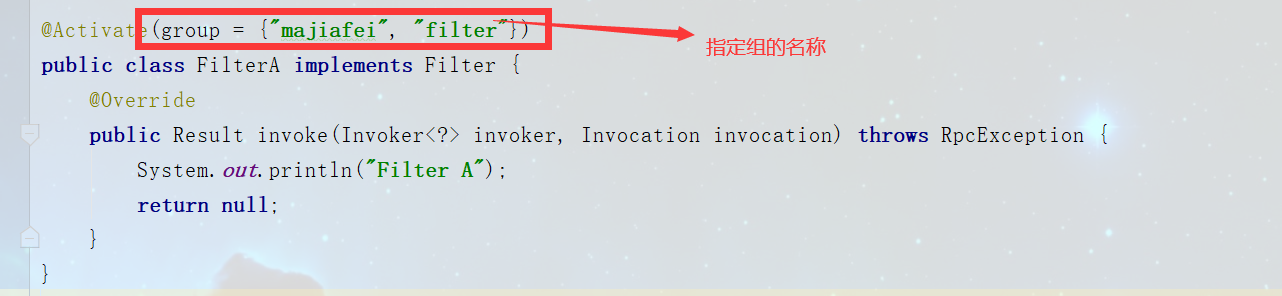
|  |
| --- |
| **public static <T> ExtensionLoader<T> getExtensionLoader(Class<T> type) {  if (type == null) {  throw new IllegalArgumentException("Extension type == null");  }**  **// 判断type是否是接口  if (!type.isInterface()) {  throw new IllegalArgumentException("Extension type (" + type + ") is not an interface!");  }**  **// 判断是否添加@SPI注解  if (!*withExtensionAnnotation*(type)) {  throw new IllegalArgumentException("Extension type (" + type +  ") is not an extension, because it is NOT annotated with @" + SPI.class.getSimpleName() + "!");  }   ExtensionLoader<T> loader = (ExtensionLoader<T>) *EXTENSION\_LOADERS*.get(type);  if (loader == null) {  *EXTENSION\_LOADERS*.putIfAbsent(type, new ExtensionLoader<T>(type));  loader = (ExtensionLoader<T>) *EXTENSION\_LOADERS*.get(type);  }  return loader; }** |

### 从文件中加载内容

|  |
| --- |
| **private Map<String, Class<?>> loadExtensionClasses() {  cacheDefaultExtensionName();   Map<String, Class<?>> extensionClasses = new HashMap<>();  loadDirectory(extensionClasses, *DUBBO\_INTERNAL\_DIRECTORY*, type.getName());  loadDirectory(extensionClasses, *DUBBO\_INTERNAL\_DIRECTORY*, type.getName().replace("org.apache", "com.alibaba"));  loadDirectory(extensionClasses, *DUBBO\_DIRECTORY*, type.getName());  loadDirectory(extensionClasses, *DUBBO\_DIRECTORY*, type.getName().replace("org.apache", "com.alibaba"));  loadDirectory(extensionClasses, *SERVICES\_DIRECTORY*, type.getName());  loadDirectory(extensionClasses, *SERVICES\_DIRECTORY*, type.getName().replace("org.apache", "com.alibaba"));  return extensionClasses; }** |

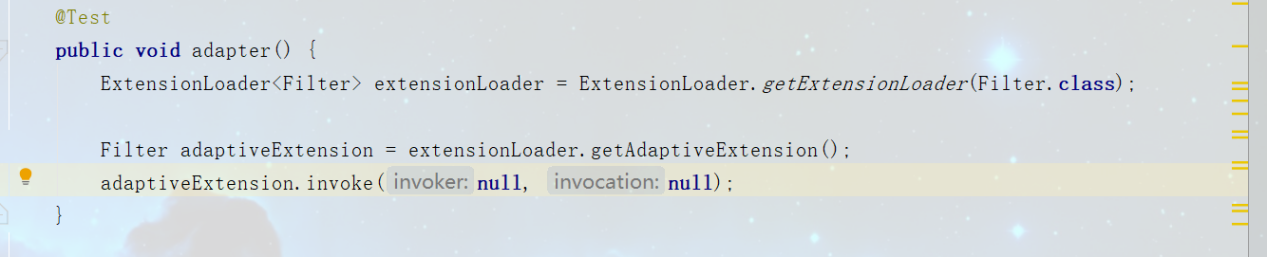
## 使用多个实现类

### @Activate



### @Adaptive

为接口找一个实现类

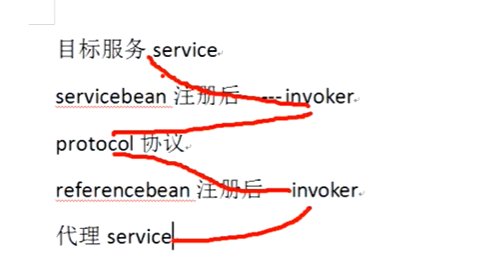


## 接口实现类依赖其他的接口，怎么做？

加载适配的实现类

### ExtensionFactory

# 消费端和服务端的大致流程



# 消费端

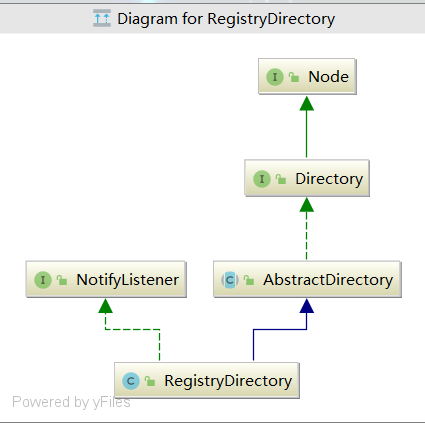
## InvokerInvocationHandler

由于引入的服务是个代理，所以调用服务的时候会调用该类的invoke方法

## 通知

### NotifyListener

当zk发现url更改之后，会调用notify方法通知消费端



# 组件

## 关系



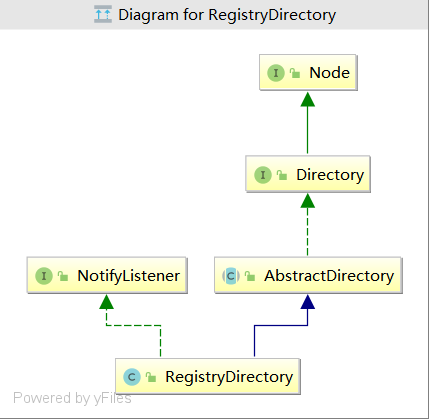
## Invoker

统一方法的调用.

invoker ---> filter-----> invoker

## Protocol

### RegistryProtocol

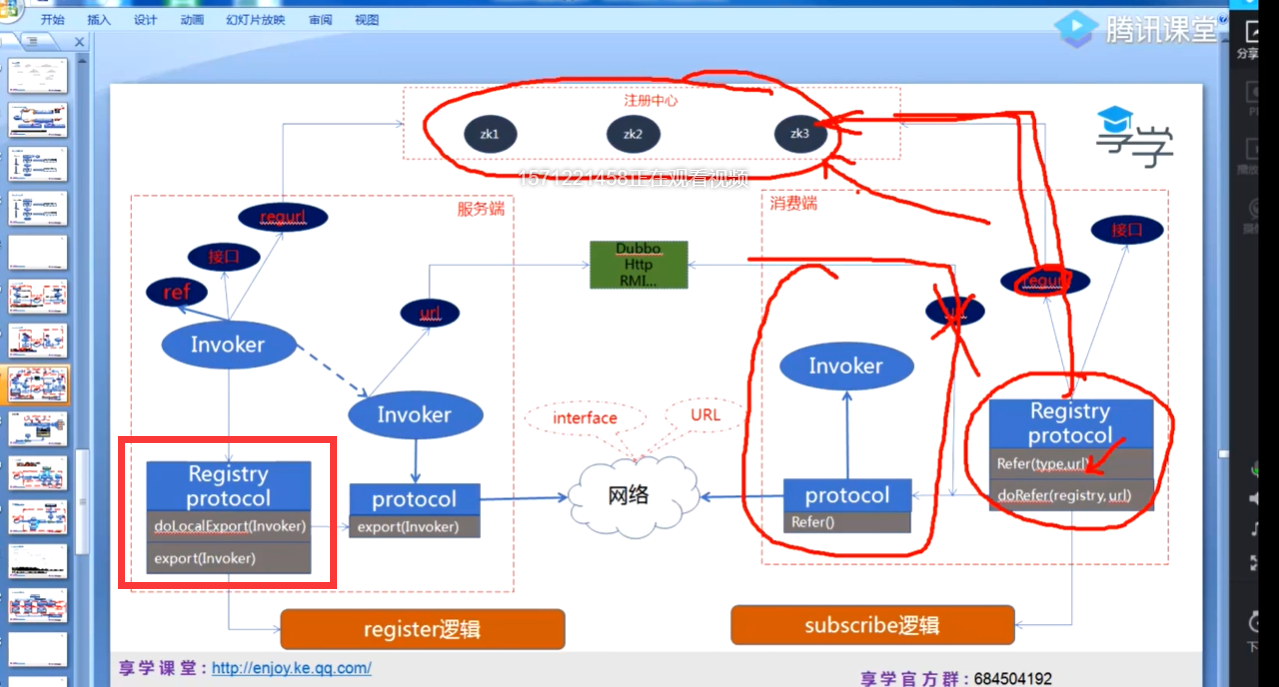


|  |
| --- |
| **private** <T> Invoker<T> doRefer(Cluster cluster, Registry registry, Class<T> type, URL url) {  RegistryDirectory<T> directory = **new** RegistryDirectory<T>(type, url);  directory.setRegistry(registry);  directory.setProtocol(**protocol**);  *// all attributes of REFER\_KEY* Map<String, 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());  }  directory.buildRouterChain(subscribeUrl);  directory.subscribe(subscribeUrl.addParameter(***CATEGORY\_KEY***,  ***PROVIDERS\_CATEGORY*** + **","** + ***CONFIGURATORS\_CATEGORY*** + **","** + ***ROUTERS\_CATEGORY***));   Invoker invoker = cluster.join(directory);  ProviderConsumerRegTable.*registerConsumer*(invoker, url, subscribeUrl, directory);  **return** invoker; } |

### 暴露服务和注册服务

|  |
| --- |
| @Override **public** <T> Exporter<T> export(**final** Invoker<T> originInvoker) **throws** RpcException {  URL registryUrl = getRegistryUrl(originInvoker);  *// url to export locally* URL providerUrl = getProviderUrl(originInvoker);   *// Subscribe the override data  // FIXME When the provider subscribes, it will affect the scene : a certain JVM exposes the service and call  // the same service. Because the subscribed is cached key with the name of the service, it causes the  // subscription information to cover.* **final** URL overrideSubscribeUrl = getSubscribedOverrideUrl(providerUrl);  **final** OverrideListener overrideSubscribeListener = **new** OverrideListener(overrideSubscribeUrl, originInvoker);  **overrideListeners**.put(overrideSubscribeUrl, overrideSubscribeListener);   providerUrl = overrideUrlWithConfig(providerUrl, overrideSubscribeListener);  *//export invoker* **final** ExporterChangeableWrapper<T> exporter = doLocalExport(originInvoker, providerUrl);   *// url to registry* **final** Registry registry = getRegistry(originInvoker);  **final** URL registeredProviderUrl = getRegisteredProviderUrl(providerUrl, registryUrl);  ProviderInvokerWrapper<T> providerInvokerWrapper = ProviderConsumerRegTable.*registerProvider*(originInvoker,  registryUrl, registeredProviderUrl);  *//to judge if we need to delay publish* **boolean** register = registeredProviderUrl.getParameter(**"register"**, **true**);  **if** (register) {  register(registryUrl, registeredProviderUrl);  providerInvokerWrapper.setReg(**true**);  }   *// Deprecated! Subscribe to override rules in 2.6.x or before.* registry.subscribe(overrideSubscribeUrl, overrideSubscribeListener);   exporter.setRegisterUrl(registeredProviderUrl);  exporter.setSubscribeUrl(overrideSubscribeUrl);  *//Ensure that a new exporter instance is returned every time export* **return new** DestroyableExporter<>(exporter); } |

### 流程

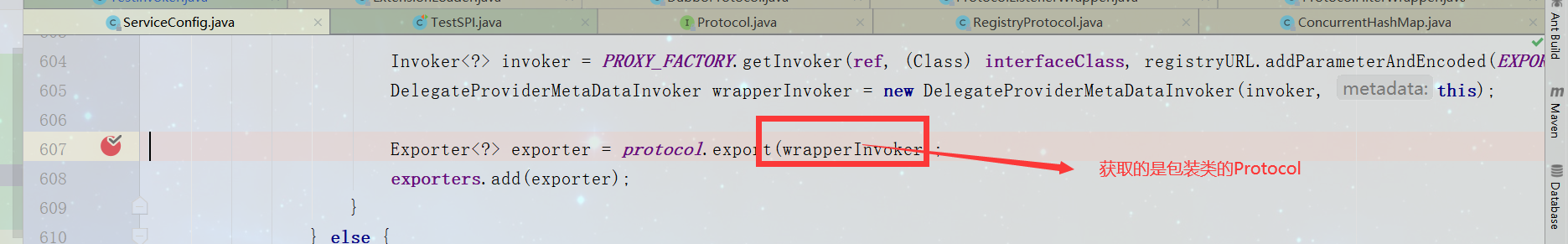


### Export方法

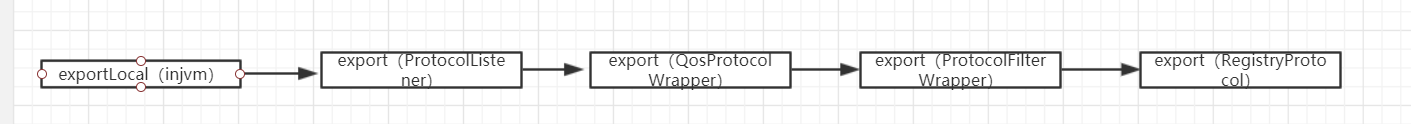
ServiceConfig类中doExportUrlsFor1Protocol 方法中的这行代码:

Exporter<?> exporter = ***protocol***.export(wrapperInvoker);为啥会走到ProtocolListenerWrapper中的export方法，不是根据url中protocol去匹配的呢吗，

wrapperInvoker的url中protocol=registry,但是调试的时候直接走向了ProtocolListenerWrapper



ProtocolListenerWrapper包装了QosProtocolWrapper包装了ProtocolFilterWrapper包装了RegistryProtocol

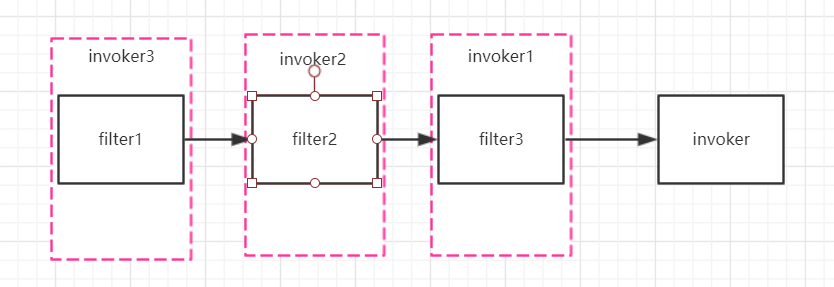


### QosProtocolWrapper

startQosServer

|  |
| --- |
| **private void** startQosServer(URL url) {  **try** {  **if** (!*hasStarted*.compareAndSet(**false**, **true**)) {  **return**;  }   **boolean** qosEnable = url.getParameter(***QOS\_ENABLE***, **true**);  **if** (!qosEnable) {  **logger**.info(**"qos won't be started because it is disabled. "** +  **"Please check dubbo.application.qos.enable is configured either in system property, "** +  **"dubbo.properties or XML/spring-boot configuration."**);  **return**;  }  **// 端口默认为2222**  **int** port = url.getParameter(***QOS\_PORT***, QosConstants.***DEFAULT\_PORT***);  **boolean** acceptForeignIp = Boolean.*parseBoolean*(url.getParameter(***ACCEPT\_FOREIGN\_IP***, **"false"**));  Server server = Server.*getInstance*();  server.setPort(port);  server.setAcceptForeignIp(acceptForeignIp);  server.start();   } **catch** (Throwable throwable) {  **logger**.warn(**"Fail to start qos server: "**, throwable);  } } |

### ProtocolFilterWrapper



|  |
| --- |
| **private static** <T> Invoker<T> buildInvokerChain(**final** Invoker<T> invoker, String key, String group) {  Invoker<T> last = invoker;  **// 获取filter list**  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 asyncResult;  **try** {  asyncResult = filter.invoke(next, invocation);  } **catch** (Exception e) {  *// onError callback* **if** (filter **instanceof** ListenableFilter) {  Filter.Listener listener = ((ListenableFilter) filter).listener();  **if** (listener != **null**) {  listener.onError(e, invoker, invocation);  }  }  **throw** e;  }  **return** asyncResult;  }   @Override  **public void** destroy() {  invoker.destroy();  }   @Override  **public** String toString() {  **return** invoker.toString();  }  };  }  }   **return new** CallbackRegistrationInvoker<>(last, filters); } |

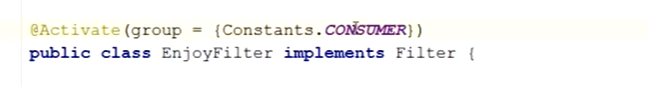
## Registry

## Filter

### MonitorFilter

监控服务

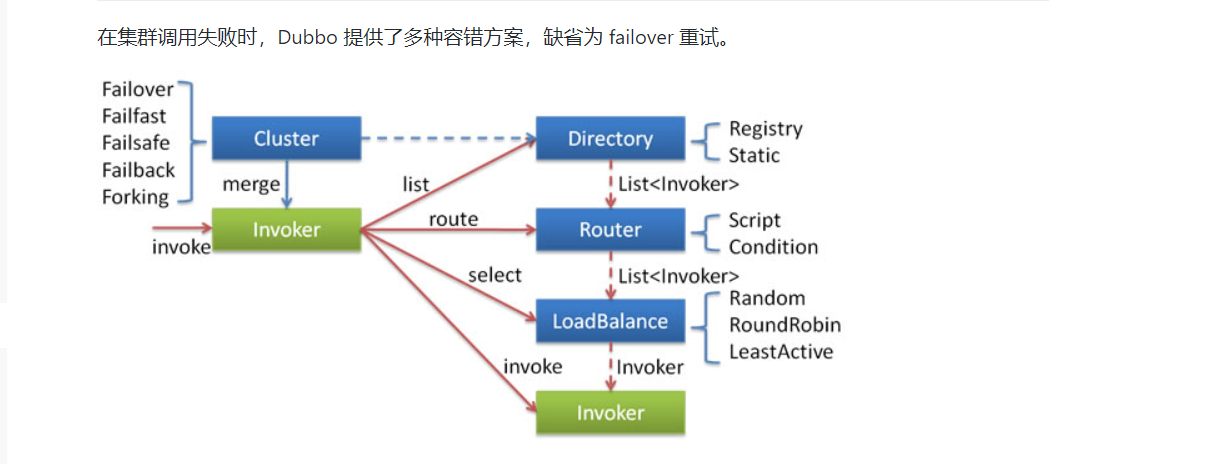
### Consumer是什么时候加入到dubbo中的



ProtocolFilterWrapper.buildInvokerChain

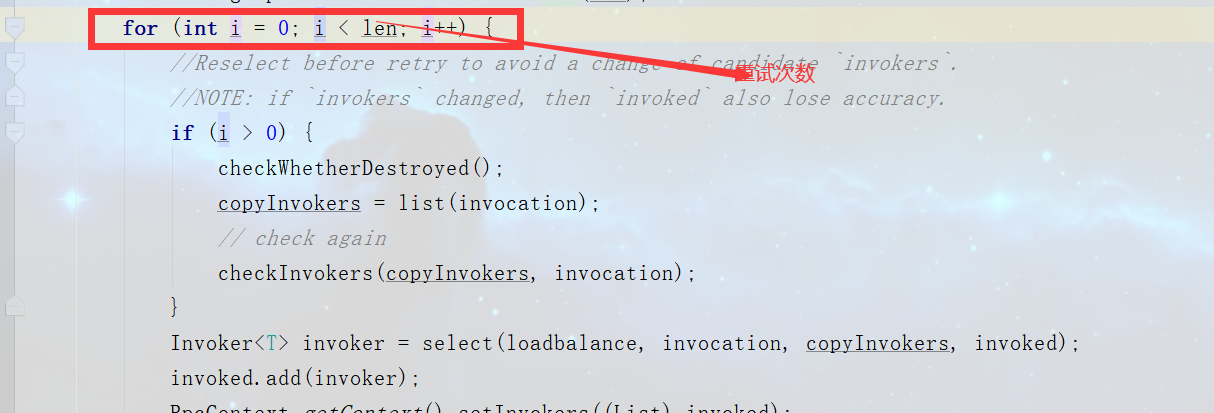
## Cluster

### 集群容错



### Failover

根据重试次数循环



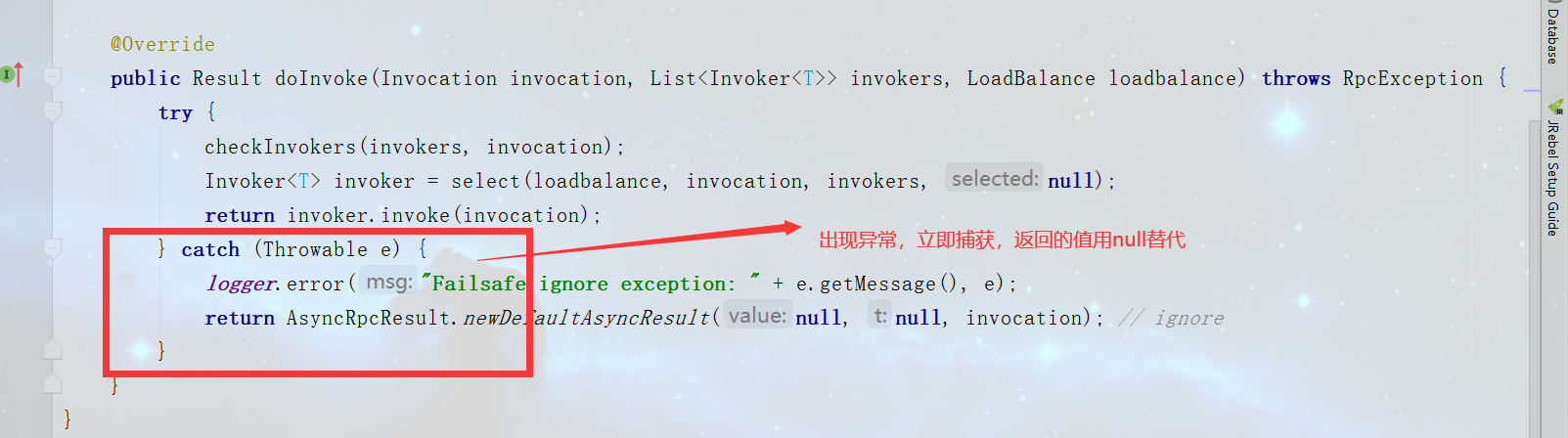
### Failfast

快速失败

只调用一次，失败即失败

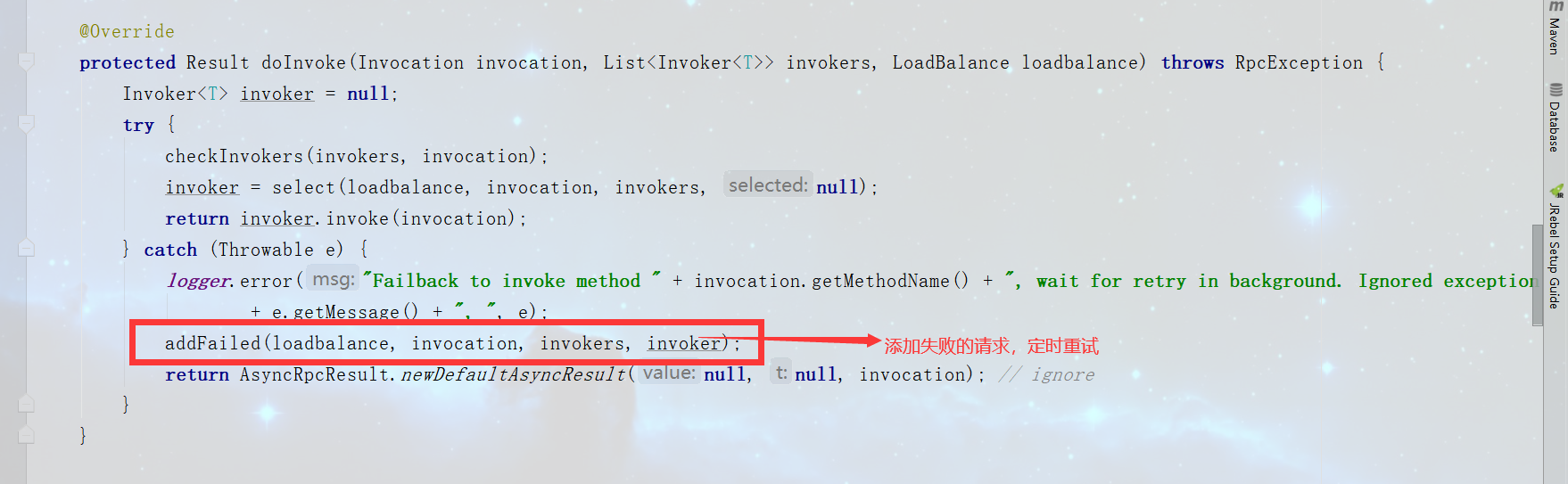
### Failsafe

失败安全



### Failback

失败自动恢复，后台记录失败请求，定时重发。通常用于消息通知操作。



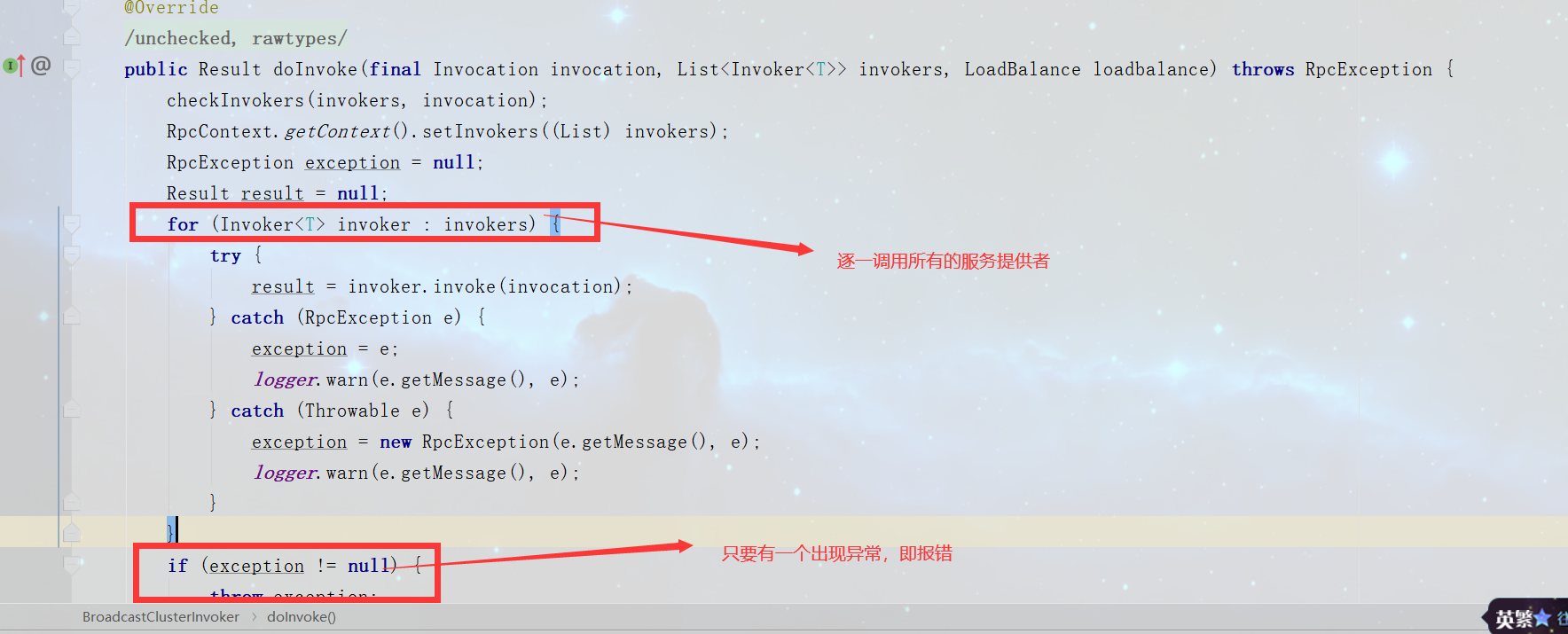
### Forking

并行调用多个服务器，只要一个成功即返回。通常用于实时性要求较高的读操作，但需要浪费更多服务资源。可通过 forks="2" 来设置最大并行数。



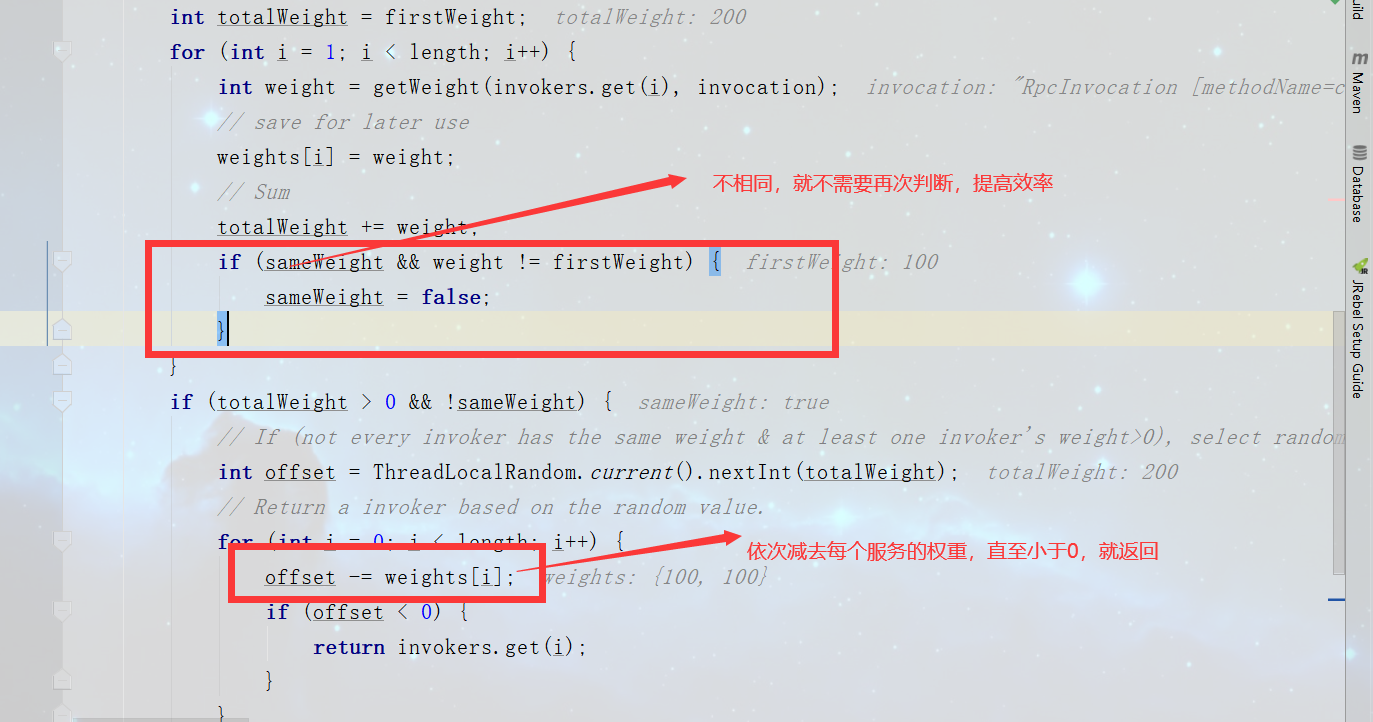
### Broadcast

广播调用所有提供者，逐个调用，任意一台报错则报错 [2]。通常用于通知所有提供者更新缓存或日志等本地资源信息。



## LoadBalance

## RandomLoadBalance



### RoundRobinLoadBalance

轮询算法

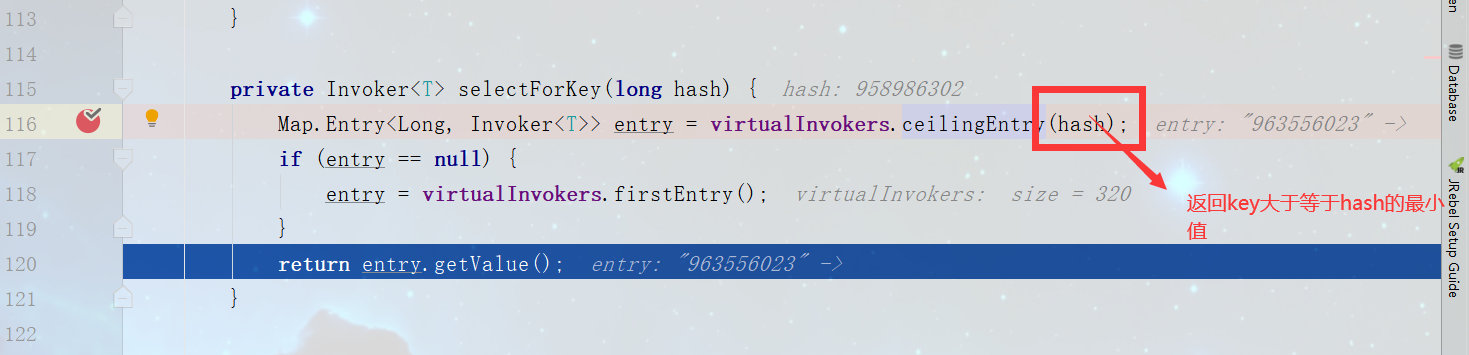
|  |
| --- |
| @Override **protected** <T> Invoker<T> doSelect(List<Invoker<T>> invokers, URL url, Invocation invocation) {  String key = invokers.get(0).getUrl().getServiceKey() + **"."** + invocation.getMethodName();  ConcurrentMap<String, WeightedRoundRobin> map = **methodWeightMap**.get(key);  **if** (map == **null**) {  **methodWeightMap**.putIfAbsent(key, **new** ConcurrentHashMap<String, WeightedRoundRobin>());  map = **methodWeightMap**.get(key);  }  **int** totalWeight = 0;  **long** maxCurrent = Long.***MIN\_VALUE***;  **long** now = System.*currentTimeMillis*();  Invoker<T> selectedInvoker = **null**;  WeightedRoundRobin selectedWRR = **null**;  **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* weightedRoundRobin.setWeight(weight);  }  **long** cur = weightedRoundRobin.increaseCurrent();  weightedRoundRobin.setLastUpdate(now);  **if** (cur > maxCurrent) **{// 谁的权重大，选谁**  maxCurrent = cur;  selectedInvoker = invoker;  selectedWRR = weightedRoundRobin;  }  totalWeight += weight;  }  **if** (!**updateLock**.get() && invokers.size() != map.size()) {  **if** (**updateLock**.compareAndSet(**false**, **true**)) {  **try** {  *// copy -> modify -> update reference* ConcurrentMap<String, WeightedRoundRobin> newMap = **new** ConcurrentHashMap<String, WeightedRoundRobin>();  newMap.putAll(map);  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();  }  }  **methodWeightMap**.put(key, newMap);  } **finally** {  **updateLock**.set(**false**);  }  }  }  **if** (selectedInvoker != **null**) {  selectedWRR.sel(totalWeight);  **return** selectedInvoker;  }  *// should not happen here* **return** invokers.get(0); } |

### LeastActive

最小活跃数

### ConsistentHashLoadBalance

一致性hash,相同参数的请求总是发到同一提供者。



### 作用

容错，当选择第一台机器出错，那就去调用其他机器。

### 容错机制

1. **Failfast Cluster：快速失败**

当服务消费方调用服务提供者失败后，立即报错，也就是只调用一次。通常这种模式用于非幂等性的写操作。

1. **Failsafe Cluster：失败安全**

当服务消费者调用服务出现异常时，直接忽略异常。这种模式通常用于写入审计日志等操作。

1. **Failback Cluster：失败自动恢复**

当服务消费端用服务出现异常后，在后台记录失败的请求，并按照一定的策略后期再进行重试。这种模式通常用于消息通知操作。

1. **Forking Cluster：并行调用**

当消费方调用一个接口方法后，Dubbo Client会并行调用多个服务提供者的服务，只要一个成功即返回。这种模式通常用于实时性要求较高的读操作，但需要浪费更多服务资源。可通过 forks="2" 来设置最大并行数。

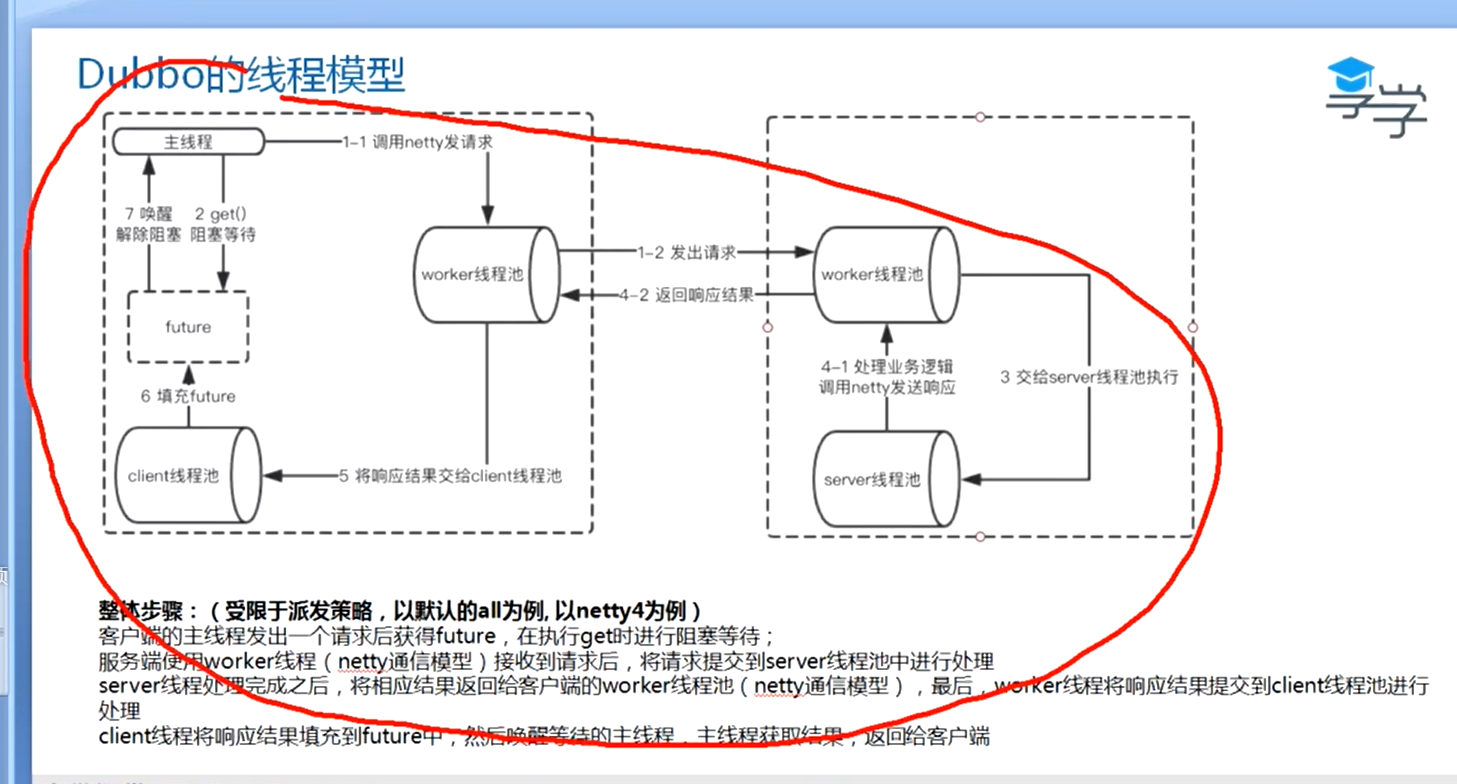
1. **Broadcast Cluster：广播调用**

当消费者调用一个接口方法后，Dubbo Client会逐个调用所有服务提供者，任意一台调用异常则这次调用就标志失败。这种模式通常用于通知所有提供者更新缓存或日志等本地资源信息。

如上，Dubbo本身提供了丰富的集群容错模式，但是如果您有定制化需求，可以根据Dubbo提供的扩展接口Cluster进行定制。在后面的消费方启动流程章节会讲解何时/如何使用的集群容错。

## LoadBlance

# 线程模型

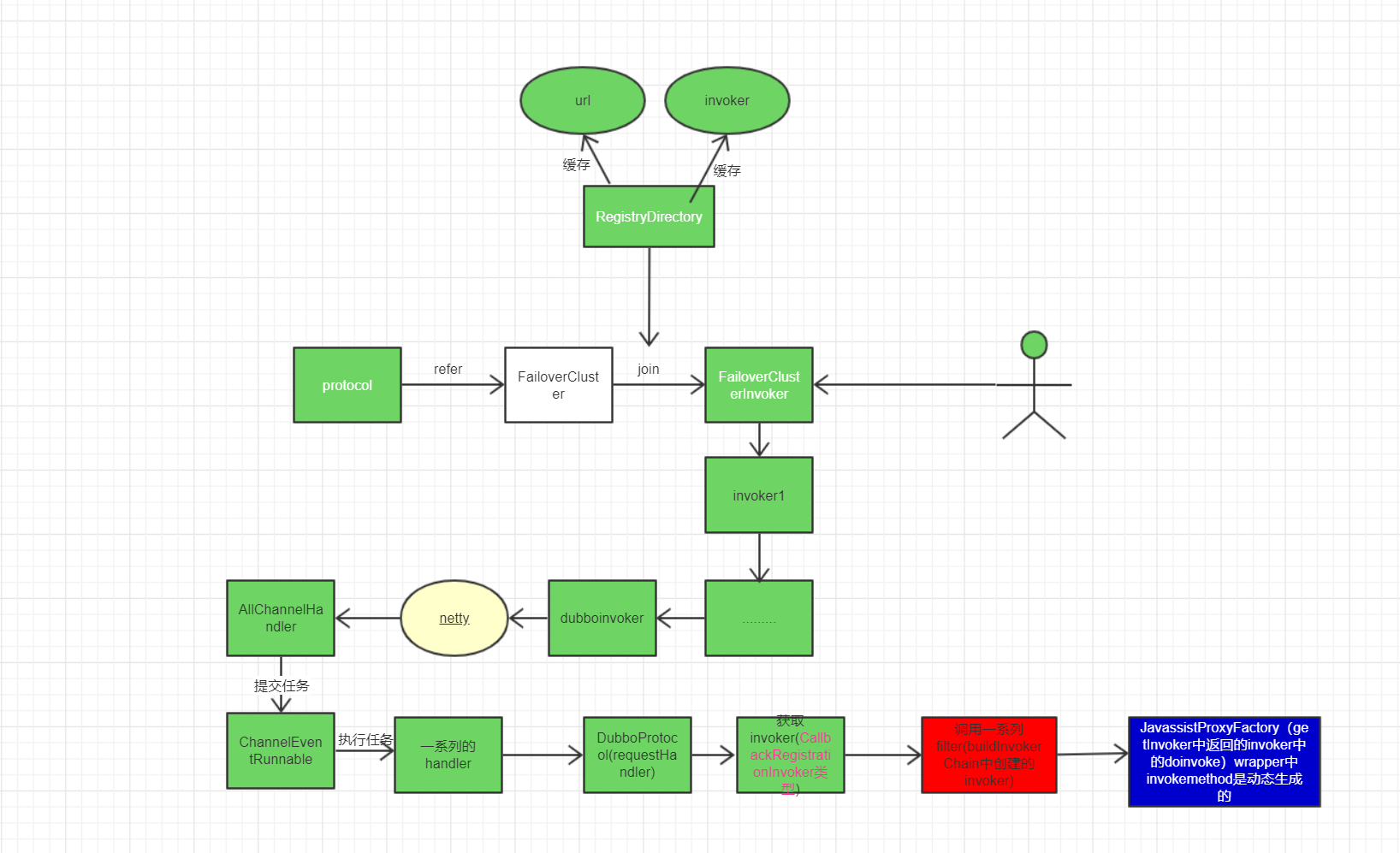


## AllChannelHandler

接收信息

|  |
| --- |
| @Override **public void** received(Channel channel, Object message) **throws** RemotingException {  ExecutorService executor = getExecutorService();  **try** {  **// 创建任务**  executor.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* **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);  } } |

# 消费端到服务端调用的整个流程



# 属性

## 只订阅

<dubbo:registry address="10.20.153.10:9090" register="false" />

## 只注册

<dubbo:registry id="qdRegistry" address="10.20.141.150:9090" subscribe="false" />

## 直接提供者

如果是线上需求需要点对点，可在 <dubbo:reference> 中配置 url 指向提供者，将绕过注册中心，多个地址用分号隔开，配置如下 [1]：

<dubbo:reference id="xxxService" interface="com.alibaba.xxx.XxxService" url="dubbo://localhost:20890" />

## 服务分组

当一个接口有多种实现时，可以用group区分

**服务**

<dubbo:service group="feedback" interface="com.xxx.IndexService" /> <dubbo:service group="member" interface="com.xxx.IndexService" />

**引用**  
<dubbo:reference id="feedbackIndexService" group="feedback" interface="com.xxx.IndexService" /> <dubbo:reference id="memberIndexService" group="member" interface="com.xxx.IndexService" />

**任意组**

<dubbo:reference id="barService" interface="com.foo.BarService" group="\*" />

## 隐式传参

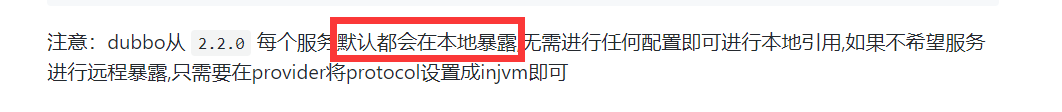


## 异步调用

CompletableFuture<String> asyncHello();

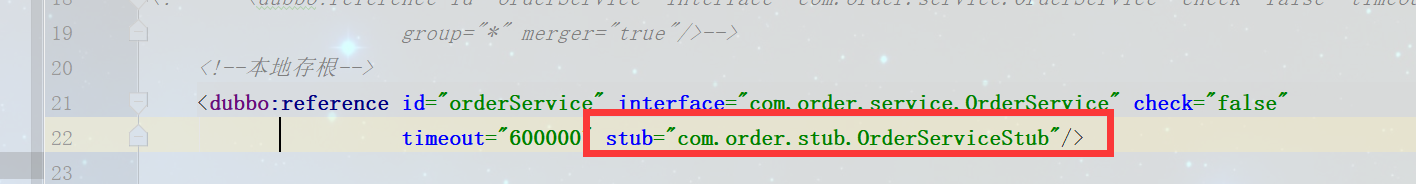
## 本地调用

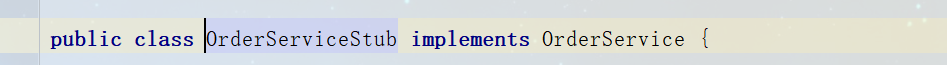
本地调用使用了injvm协议，是一个伪协议，它不开启端口，不发起远程调用，只在jvm内直接关联，但执行dubbo的filter链



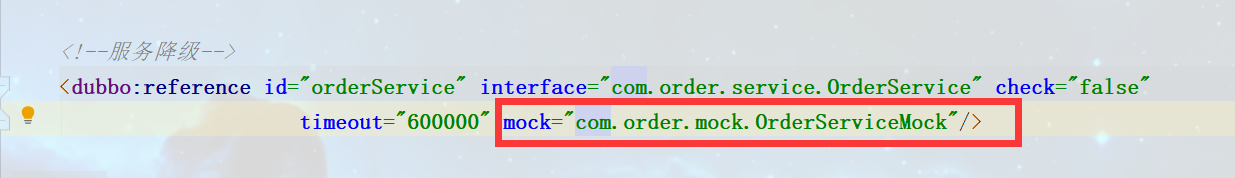
## 本地存根

Stub





## 本地伪装





## 优雅停机

Dubbo 是通过 JDK 的 ShutdownHook 来完成优雅停机的，所以如果用户使用 kill -9 PID 等强制关闭指令，是不会执行优雅停机的，只有通过 kill PID 时，才会执行。