

Dubbo源码解析（三十七）集群——directory

[dubbo](#)  阅读约 60 分钟

集群——directory

目标：介绍dubbo中集群的目录，介绍dubbo-cluster下directory包的源码。

前言

我在前面的文章中也提到了Directory可以看成是多个Invoker的集合，Directory 的用途是保存 Invoker，其实现类 RegistryDirectory 是一个动态服务目录，可感知注册中心配置的变化，它所持有的 Invoker 列表会随着注册中心内容的变化而变化。每次变化后，RegistryDirectory 会动态增删 Invoker，那在之前文章中我忽略了RegistryDirectory的源码分析，在本文中来补充。

源码分析

（一）AbstractDirectory

该类实现了Directory接口，

1. 属性

```
// Logger
private static final Logger logger = LoggerFactory.getLogger(AbstractDirectory.class);

/**
 * url 对象
 */
private final URL url;

/**
 * 是否销毁
 */
private volatile boolean destroyed = false;

/**
 * 消费者端url
 */
private volatile URL consumerUrl;

/**
 * 路由集合
 */
private volatile List<Router> routers;
```

2. list

```

@Override
public List<Invoker<T>> list(Invocation invocation) throws RpcException {
    // 如果销毁，则抛出异常
    if (destroyed) {
        throw new RpcException("Directory already destroyed .url: " + getUrl());
    }
    // 调用doList来获得Invoker集合
    List<Invoker<T>> invokers = doList(invocation);
    // 获得路由集合
    List<Router> localRouters = this.routers; // Local reference
    if (localRouters != null && !localRouters.isEmpty()) {
        // 遍历路由
        for (Router router : localRouters) {
            try {
                if (router.getUrl() == null || router.getUrl().getParameter(Constants.RUNTIME_KEY, false)) {
                    // 根据路由规则选择符合规则的invoker集合
                    invokers = router.route(invokers, getConsumerUrl(), invocation);
                }
            } catch (Throwable t) {
                logger.error("Failed to execute router: " + getUrl() + ", cause: " + t.getMessage(), t);
            }
        }
    }
    return invokers;
}

```

该方法是生成invoker集合的逻辑实现。其中doList是抽象方法，交由子类来实现。

3.setRouters

```

protected void setRouters(List<Router> routers) {
    // copy list
    // 复制路由集合
    routers = routers == null ? new ArrayList<Router>() : new ArrayList<Router>(routers);
    // append url router
    // 获得路由的配置
    String routerkey = url.getParameter(Constants.ROUTER_KEY);
    if (routerkey != null && routerkey.length() > 0) {
        // 加载路由工厂
        RouterFactory routerFactory = ExtensionLoader.getExtensionLoader(RouterFactory.class).getExtension(routerkey);
        // 加入集合
        routers.add(routerFactory.getRouter(url));
    }
    // append mock invoker selector
    // 加入服务降级路由
    routers.add(new MockInvokersSelector());
    // 排序
    Collections.sort(routers);
    this.routers = routers;
}

```

(二) StaticDirectory

静态 Directory 实现类，将传入的 invokers 集合，封装成静态的 Directory 对象。

```

public class StaticDirectory<T> extends AbstractDirectory<T> {

    private final List<Invoker<T>> invokers;

    public StaticDirectory(List<Invoker<T>> invokers) {
        this(null, invokers, null);
    }

    public StaticDirectory(List<Invoker<T>> invokers, List<Router> routers) {
        this(null, invokers, routers);
    }

    public StaticDirectory(URL url, List<Invoker<T>> invokers) {
        this(url, invokers, null);
    }

    public StaticDirectory(URL url, List<Invoker<T>> invokers, List<Router> routers) {
        super(url == null && invokers != null && !invokers.isEmpty() ? invokers.get(0).getUrl() : url, routers);
        if (invokers == null || invokers.isEmpty())
            throw new IllegalArgumentException("invokers == null");
        this.invokers = invokers;
    }

    @Override
    public Class<T> getInterface() {
        return invokers.get(0).getInterface();
    }
}

```

该类我就不多讲解，比较简单。

(三) RegistryDirectory

该类继承了AbstractDirectory类，是基于注册中心的动态 Directory 实现类，会根据注册中心的推送变更 List<Invoker>

1. 属性

```

private static final Logger logger = LoggerFactory.getLogger(RegistryDirectory.class);

/**
 * cluster实现类对象
 */
private static final Cluster cluster = ExtensionLoader.getExtensionLoader(Cluster.class).getAdaptiveExtension();

/**
 * 路由工厂
 */
private static final RouterFactory routerFactory =
ExtensionLoader.getExtensionLoader(RouterFactory.class).getAdaptiveExtension();

/**
 * 配置规则工厂
 */
private static final ConfiguratorFactory configuratorFactory =
ExtensionLoader.getExtensionLoader(ConfiguratorFactory.class).getAdaptiveExtension();

/**
 * 服务key
 */
private final String serviceKey; // Initialization at construction time, assertion not null
/**
 * 服务类型
 */

```

2. toConfigurators

```

public static List<Configurator> toConfigurators(List<URL> urls) {
    // 如果为空，则返回空集合
    if (urls == null || urls.isEmpty()) {
        return Collections.emptyList();
    }

    List<Configurator> configurators = new ArrayList<Configurator>(urls.size());
    // 遍历url集合
    for (URL url : urls) {
        //如果是协议是empty的值，则清空配置集合
        if (Constants.EMPTY_PROTOCOL.equals(url.getProtocol())) {
            configurators.clear();
            break;
        }
        // 覆盖的参数集合
        Map<String, String> override = new HashMap<String, String>(url.getParameters());
        //The anyhost parameter of override may be added automatically, it can't change the judgement of changing
        url
        // 覆盖的anyhost参数可以自动添加，也不能改变更改url的判断
        override.remove(Constants.ANYHOST_KEY);
        // 如果需要覆盖添加的值为0，则清空配置
        if (override.size() == 0) {
            configurators.clear();
            continue;
        }
        // 加入配置规则集合
    }
}

```

该方法是处理配置规则url集合，转换覆盖url映射以便在重新引用时使用，每次发送所有规则，网址将被重新组装和计算。

3.destroy

```

@Override
public void destroy() {
    // 如果销毁了，则返回
    if (isDestroyed()) {
        return;
    }
    // unsubscribe.
    try {
        if (getConsumerUrl() != null && registry != null && registry.isAvailable()) {
            // 取消订阅
            registry.unsubscribe(getConsumerUrl(), this);
        }
    } catch (Throwable t) {
        logger.warn("unexpeced error when unsubscribe service " + serviceKey + "from registry" + registry.getUrl(),
t);
    }
    super.destroy(); // must be executed after unsubscribing
    try {
        // 清空所有的invoker
        destroyAllInvokers();
    } catch (Throwable t) {
        logger.warn("Failed to destroy service " + serviceKey, t);
    }
}

```

该方法是销毁方法。

4.destroyAllInvokers

```

private void destroyAllInvokers() {
    Map<String, Invoker<T>> localUrlInvokerMap = this.urlInvokerMap; // Local reference
    // 如果invoker集合不为空
    if (localUrlInvokerMap != null) {
        // 遍历
        for (Invoker<T> invoker : new ArrayList<Invoker<T>>(localUrlInvokerMap.values())) {
            try {
                // 销毁invoker
                invoker.destroy();
            } catch (Throwable t) {
                logger.warn("Failed to destroy service " + serviceKey + " to provider " + invoker.getUrl(), t);
            }
        }
        // 清空集合
        localUrlInvokerMap.clear();
    }
    methodInvokerMap = null;
}

```

该方法是关闭所有的invoker服务。

5.notify

```

@Override
public synchronized void notify(List<URL> urls) {
    List<URL> invokerUrls = new ArrayList<URL>();
    List<URL> routerUrls = new ArrayList<URL>();
    List<URL> configuratorUrls = new ArrayList<URL>();
    // 遍历url
    for (URL url : urls) {
        // 获得协议
        String protocol = url.getProtocol();
        // 获得类别
        String category = url.getParameter(Constants.CATEGORY_KEY, Constants.DEFAULT_CATEGORY);
        // 如果是路由规则
        if (Constants.ROUTERS_CATEGORY.equals(category)
            || Constants.ROUTE_PROTOCOL.equals(protocol)) {
            // 则在路由规则集合中加入
            routerUrls.add(url);
        } else if (Constants.CONFIGURATORS_CATEGORY.equals(category)
            || Constants.OVERRIDE_PROTOCOL.equals(protocol)) {
            // 如果是配置规则，则加入配置规则集合
            configuratorUrls.add(url);
        } else if (Constants.PROVIDERS_CATEGORY.equals(category)) {
            // 如果是服务提供者，则加入服务提供者集合
            invokerUrls.add(url);
        } else {
            logger.warn("Unsupported category " + category + " in notified url: " + url + " from registry " +
getAddress() + " to consumer " + NetUtils.getLocalHost());
        }
    }
}

```

当服务有变化的时候，执行该方法。首先将url根据路由规则、服务提供者和配置规则三种类型分开，分别放入三个集合，然后对每个集合进行修改或者通知

6.refreshInvoker

```

private void refreshInvoker(List<URL> invokerUrls) {
    if (invokerUrls != null && invokerUrls.size() == 1 && invokerUrls.get(0) != null
        && Constants.EMPTY_PROTOCOL.equals(invokerUrls.get(0).getProtocol())) {
        // 设置禁止访问
        this.forbidden = true; // Forbid to access
        // methodInvokerMap 置空
        this.methodInvokerMap = null; // Set the method invoker map to null
        // 关闭所有的invoker
        destroyAllInvokers(); // Close all invokers
    } else {
        // 关闭禁止访问
        this.forbidden = false; // Allow to access
        // 引用老的urlInvokerMap
        Map<String, Invoker<T>> oldUrlInvokerMap = this.urlInvokerMap; // Local reference
        // 传入的invokerUrls 为空，说明是路由规则或配置规则发生改变，此时invokerUrls 是空的，直接使用
        cachedInvokerUrls。
        if (invokerUrls.isEmpty() && this.cachedInvokerUrls != null) {
            invokerUrls.addAll(this.cachedInvokerUrls);
        } else {
            // 否则把所有的invokerUrls加入缓存
            this.cachedInvokerUrls = new HashSet<URL>();
            this.cachedInvokerUrls.addAll(invokerUrls); //Cached invoker urls, convenient for comparison
        }
        // 如果invokerUrls为空，则直接返回
        if (invokerUrls.isEmpty()) {
            return;
        }
    }
}

```

该方法是处理服务提供者 URL 集合。根据 invokerURL 列表转换为 invoker 列表。转换规则如下：

1. 如果 url 已经被转换为 invoker，则不在重新引用，直接从缓存中获取，注意如果 url 中任何一个参数变更也会重新引用。
2. 如果传入的 invoker 列表不为空，则表示最新的 invoker 列表。
3. 如果传入的 invokerUrl 列表是空，则表示只是下发的 override 规则或 route 规则，需要重新交叉对比，决定是否需要重新引用。

7.toMergeMethodInvokerMap

```

private Map<String, List<Invoker<T>>> toMergeMethodInvokerMap(Map<String, List<Invoker<T>>> methodMap) {
    // 循环方法，按照 method + group 聚合 Invoker 集合
    Map<String, List<Invoker<T>>> result = new HashMap<String, List<Invoker<T>>>();
    // 遍历方法集合
    for (Map.Entry<String, List<Invoker<T>>> entry : methodMap.entrySet()) {
        // 获得方法
        String method = entry.getKey();
        // 获得invoker集合
        List<Invoker<T>> invokers = entry.getValue();
        // 获得组集合
        Map<String, List<Invoker<T>>> groupMap = new HashMap<String, List<Invoker<T>>>();
        // 遍历invoker集合
        for (Invoker<T> invoker : invokers) {
            // 获得url携带的组配置
            String group = invoker.getUrl().getParameter(Constants.GROUP_KEY, "");
            // 获得该组对应的invoker集合
            List<Invoker<T>> groupInvokers = groupMap.get(group);
            // 如果为空，则新创建一个，然后加入集合
            if (groupInvokers == null) {
                groupInvokers = new ArrayList<Invoker<T>>();
                groupMap.put(group, groupInvokers);
            }
            groupInvokers.add(invoker);
        }
        // 如果只有一个组
        if (groupMap.size() == 1) {

```

该方法是通过按照 method + group 来聚合 Invoker 集合。

8.toRouters

```

private List<Router> toRouters(List<URL> urls) {
    List<Router> routers = new ArrayList<Router>();
    // 如果为空，则直接返回空集合
    if (urls == null || urls.isEmpty()) {
        return routers;
    }
    if (urls != null && !urls.isEmpty()) {
        // 遍历url集合
        for (URL url : urls) {
            // 如果为empty协议，则直接跳过
            if (Constants.EMPTY_PROTOCOL.equals(url.getProtocol())) {
                continue;
            }
            // 获得路由规则
            String routerType = url.getParameter(Constants.ROUTER_KEY);
            if (routerType != null && routerType.length() > 0) {
                // 设置协议
                url = url.setProtocol(routerType);
            }
            try {
                // 获得路由
                Router router = routerFactory.getRouter(url);
                if (!routers.contains(router))
                    // 加入集合
                    routers.add(router);
            } catch (Throwable t) {

```

该方法是对url集合进行路由的解析，返回路由集合。

9.toInvokers

```

private Map<String, Invoker<T>> toInvokers(List<URL> urls) {
    Map<String, Invoker<T>> newUrlInvokerMap = new HashMap<String, Invoker<T>>();
    // 如果为空，则返回空集合
    if (urls == null || urls.isEmpty()) {
        return newUrlInvokerMap;
    }
    Set<String> keys = new HashSet<String>();
    // 获得引用服务的协议
    String queryProtocols = this.queryMap.get(Constants.PROTOCOL_KEY);
    // 遍历url
    for (URL providerUrl : urls) {
        // If protocol is configured at the reference side, only the matching protocol is selected
        // 如果在参考侧配置协议，则仅选择匹配协议
        if (queryProtocols != null && queryProtocols.length() > 0) {
            boolean accept = false;
            // 分割协议
            String[] acceptProtocols = queryProtocols.split(",");
            // 遍历协议
            for (String acceptProtocol : acceptProtocols) {
                // 如果匹配，则是接受的协议
                if (providerUrl.getProtocol().equals(acceptProtocol)) {
                    accept = true;
                    break;
                }
            }
            if (!accept) {

```

该方法是将url转换为调用者，如果url已被引用，则不会重新引用。

10.mergeUrl

```

private URL mergeUrl(URL providerUrl) {
    // 合并消费端参数
    providerUrl = ClusterUtils.mergeUrl(providerUrl, queryMap); // Merge the consumer side parameters

    // 合并配置规则
    List<Configurator> localConfigurators = this.configurators; // Local reference
    if (localConfigurators != null && !localConfigurators.isEmpty()) {
        for (Configurator configurator : localConfigurators) {
            providerUrl = configurator.configure(providerUrl);
        }
    }

    // 不检查连接是否成功，总是创建 Invoker
    providerUrl = providerUrl.addParameter(Constants.CHECK_KEY, String.valueOf(false)); // Do not check whether
    the connection is successful or not, always create Invoker!

    // The combination of directoryUrl and override is at the end of notify, which can't be handled here
    // 合并提供者参数，因为 directoryUrl 与 override 合并在 notify 的最后，这里不能够处理
    this.overrideDirectoryUrl = this.overrideDirectoryUrl.addParametersIfAbsent(providerUrl.getParameters()); // // Merge the provider side parameters

    // 1.0版本兼容
    if ((providerUrl.getPath() == null || providerUrl.getPath().length() == 0)
        && "dubbo".equals(providerUrl.getProtocol())) { // Compatible version 1.0
        //fix by tony.chenl DUBBO-44
        String path = directoryUrl.getParameter(Constants.INTERFACE_KEY);
    }
}

```

该方法是合并 URL 参数，优先级为配置规则 > 服务消费者配置 > 服务提供者配置。

11.toMethodInvokers

```

private Map<String, List<Invoker<T>>> toMethodInvokers(Map<String, Invoker<T>> invokersMap) {
    Map<String, List<Invoker<T>>> newMethodInvokerMap = new HashMap<String, List<Invoker<T>>>();
    // According to the methods classification declared by the provider URL, the methods is compatible with the
    registry to execute the filtered methods
    List<Invoker<T>> invokersList = new ArrayList<Invoker<T>>();
    if (invokersMap != null && invokersMap.size() > 0) {
        // 遍历调用者列表
        for (Invoker<T> invoker : invokersMap.values()) {
            String parameter = invoker.getUrl().getParameter(Constants.METHODS_KEY);
            // 按服务提供者 URL 所声明的 methods 分类
            if (parameter != null && parameter.length() > 0) {
                // 分割参数得到方法集合
                String[] methods = Constants.COMMA_SPLIT_PATTERN.split(parameter);
                if (methods != null && methods.length > 0) {
                    // 遍历方法集合
                    for (String method : methods) {
                        if (method != null && method.length() > 0
                            && !Constants.ANY_VALUE.equals(method)) {
                            // 获得该方法对应的invoker，如果为空，则创建
                            List<Invoker<T>> methodInvokers = newMethodInvokerMap.get(method);
                            if (methodInvokers == null) {
                                methodInvokers = new ArrayList<Invoker<T>>();
                                newMethodInvokerMap.put(method, methodInvokers);
                            }
                            methodInvokers.add(invoker);
                        }
                    }
                }
            }
        }
    }
}

```

该方法是将调用者列表转换为与方法的映射关系。

12.destroyUnusedInvokers

```

private void destroyUnusedInvokers(Map<String, Invoker<T>> oldUrlInvokerMap, Map<String, Invoker<T>>
newUrlInvokerMap) {
    if (newUrlInvokerMap == null || newUrlInvokerMap.size() == 0) {
        destroyAllInvokers();
        return;
    }
    // check deleted invoker
    // 记录已经删除的invoker
    List<String> deleted = null;
    if (oldUrlInvokerMap != null) {
        Collection<Invoker<T>> newInvokers = newUrlInvokerMap.values();
        // 遍历旧的invoker集合
        for (Map.Entry<String, Invoker<T>> entry : oldUrlInvokerMap.entrySet()) {
            if (!newInvokers.contains(entry.getValue())) {
                if (deleted == null) {
                    deleted = new ArrayList<String>();
                }
                // 加入该invoker
                deleted.add(entry.getKey());
            }
        }
    }
    if (deleted != null) {
        // 遍历需要删除的invoker url集合
        for (String url : deleted) {
    }
}

```

该方法是销毁不再使用的 Invoker 集合。

13. doList

```

@Override
public List<Invoker<T>> doList(Invocation invocation) {
    // 如果禁止访问，则抛出异常
    if (forbidden) {
        // 1. No service provider 2. Service providers are disabled
        throw new RpcException(RpcException.FORBIDDEN_EXCEPTION,
            "No provider available from registry " + getUrl().getAddress() + " for service " +
            getConsumerUrl().getServiceKey() + " on consumer " + NetUtils.getLocalHost()
            + " use dubbo version " + Version.getVersion() + ", please check status of
            providers(disabled, not registered or in blacklist).");
    }
    List<Invoker<T>> invokers = null;
    Map<String, List<Invoker<T>>> localMethodInvokerMap = this.methodInvokerMap; // Local reference
    if (localMethodInvokerMap != null && localMethodInvokerMap.size() > 0) {
        // 获得方法名
        String methodName = RpcUtils.getMethodName(invocation);
        // 获得参数名
        Object[] args = RpcUtils.getArguments(invocation);
        if (args != null && args.length > 0 && args[0] != null
            && (args[0] instanceof String || args[0].getClass().isEnum())) {
            // 根据第一个参数枚举路由
            invokers = localMethodInvokerMap.get(methodName + "." + args[0]); // The routing can be enumerated
            according to the first parameter
        }
        if (invokers == null) {
            // 根据方法名获得 Invoker 集合
    }
}

```

该方法是通过会话域来获得Invoker集合。

后记

该部分相关的源码解析地址：<https://github.com/CrazyHZM/i...>

该文章讲解了集群中关于directory实现的部分，关键是RegistryDirectory，其中涉及到众多方法，需要好好品味。接下来我将开始对集群模块关于loadbalance部分进行讲解。