【十六】Dubbo集群 之 目录服务

在接触到一个概念之前，我们需要搞清楚它到底有啥含义？是干啥的？只有整明白后，才能更快地基于它展开学习和研究。Dubbo这个RPC框架，其设计，用今天软件开发中流行的话术讲，是基于DDD领域驱动设计的，自然比较遵从业界约定俗成的认知，涉及不少相关领域方面的概念，比如框架层中每一分层的名称包括功能特性，在集群这个模块我们接触到的负载均衡、路由，以及本文需要展开详述的目录服务也是分布式软件系统中一个领域概念。

目录服务，有时也称作 名字服务，下述是维基百科中关于它的描述，简单理解就是提供 名称 到 资源值、服务、Action等 的映射服务，比如被广为认知的DNS：

> *目录服务(https://zh.wikipedia.org/wiki/%E7%9B%AE%E5%BD%95%E6%9C%8D%E5%8A%A1)（英语：Directory service）是一个储存、组织和提供信息访问服务的软件系统，在软件工程中，一个目录是指一组名字和值的映射。它允许根据一个给出的名字来查找对应的值，与词典相似。...名字服务（英语：Name service）是一个简单的目录服务，名字服务将一个网络资源的名字与它的网络地址进行映射。...目录服务是一种共享的基础信息服务，可用来定位、管理和组织通用项目和网络资源。*

可见，有了目录服务， 一个用户不必记住某个网络资源的物理地址，只需要提供这个网络资源的名字就可以找到它 。

接口定义

前面两篇关于Dubbo集群实现剖析的文章中，已经说过：1） 负载均衡 是客户端在一个微服务的所有可用服务引用实例中基于某个策略挑选一个作为处理RPC请求的对象；2） 路由 是基于配置中心的路由覆写规则，为当前RPC方法的匹配到目标微服务的所有满足条件的引用实例集合。

此前在关于RPC远程方法调用实现剖析中，有讲过，一个RPC方法请求体中，方法级别的信息入参及入参类型在 `Invocation` 中体现，而其引用微服务级别的信息元数据却是体现在URL配置总线中的。在注册中心的支持下，客户端并不需要指定其引用微服务的物理地址，简单配置后者的服务名称便可。也就是说需要通过名称获取所有的目标微服务的引用实例 `Invoker对象` ，而这正是Dubbo集群中 `Directory` 存在的价值。

接口定义源码如下：

JAVA

```java
public interface Directory<T> extends Node {

    //get service type.
    Class<T> getInterface();

    //list invokers.
    List<Invoker<T>> list(Invocation invocation) throws RpcException;

}
```

> **NOTE**
>
> 在一个微观的粒度，我们总是很难看清一件事物的本质，就像盲人摸象，把象腿当做柱子，如果想更快速获知对 `Directory` 的总体印象，请参考分组折叠处理的相关描述。

AbstractDirectory

抽象基类 `AbstractDirectory` 是 `Directory<T>` 的模板实现，它定义了一些基本的行为规范。首先尽管 `Directory` 只服务于被引用的特定微服务，但其执行环境依然是高并发充满竞争的，一些公共的并发状态管理是必要的。

其实 `Directory` 在Dubbo中被当做一个组件模块对待，于注册中心而言，它是一个应用层次的客户端，承担着一些向框架上层提供服务的职责，因而有着自己的生命体征，这在分层设计体系中也是一个比较优秀的设计思想。

如下述源码，作为注册中心的应用客户端，如果它被销毁了，则被引用微服务的任一将要发生RPC调用行为的 `Invoker` 对象需要及时感知到这种状态的变化， `destroyed` 被声明为 `volatile` 。

JAVA

```java
private volatile boolean destroyed = false;

public boolean isDestroyed() {
    return destroyed;
}

@Override
public void destroy() {
    destroyed = true;
}

@Override
public List<Invoker<T>> list(Invocation invocation) throws RpcException {
    if (destroyed) {
        throw new RpcException("Directory already destroyed .url: " + getUrl());
    }

    return doList(invocation);
}

protected abstract List<Invoker<T>> doList(Invocation invocation) throws RpcException;
```

再以 consumerUrl 为例，它于注册中心来说，是其应用客户端的关键标识，客户端用其观察注册中心的一个数据节点，可能因为某些变动，比如切换一个注册中心，框架上层会执行一些重新订阅操作，这时所有引用微服务的相关实例也需要第一时间感知到这一变化，避免在一个错误的节点上执行相关行为，因而 consumerUrl 也被声明成 volatile 。

JAVA

```java
private final URL url;

private volatile URL consumerUrl;

public AbstractDirectory(URL url) {
    this(url, null);
}

public AbstractDirectory(URL url, RouterChain<T> routerChain) {
    this(url, url, routerChain);
}

public AbstractDirectory(URL url, URL consumerUrl, RouterChain<T> routerChain) {
    if (url == null) {
        throw new IllegalArgumentException("url == null");
    }

    if (url.getProtocol().equals(REGISTRY_PROTOCOL)) {
        Map<String, String> queryMap =
StringUtils.parseQueryString(url.getParameterAndDecoded(REFER_KEY));
        this.url = url.addParameters(queryMap).removeParameter(MONITOR_KEY);
    } else {
        this.url = url;
    }

    this.consumerUrl = consumerUrl;
    setRouterChain(routerChain);
}

public URL getConsumerUrl() {
    return consumerUrl;
}

public void setConsumerUrl(URL consumerUrl) {
    this.consumerUrl = consumerUrl;
}
```

另外源码中的 `url` 表达的是啥呢？要知道在微服务框架中，引用微服务这个行为是要通过注册中心的。以Dubbo的尿性，涉及到数据传递和临时存取的都会借由一个URL数据对象，如 `registry://registry-host/org.apache.dubbo.registry.RegistryService?refer=URL.encode("consumer://consumer-host/com.foo.FooService?version=1.0.0")`，表示被引用微服务的URL数据被编码到了 `refer` 这个参数中，这导致相关特征参数的存取困难，因而构造函数中执行了解码处理。

其次 `Directory` 只是根据注册中心为当前客户列出所有目标被引用微服务的所有可用候选集，需要进一步应用客户端本身定义的一些筛选过滤规则，也就是路由处理，避免PRC调用时的不必要开销。

JAVA

```java
protected RouterChain<T> routerChain;

public RouterChain<T> getRouterChain() {
    return routerChain;
}

public void setRouterChain(RouterChain<T> routerChain) {
    this.routerChain = routerChain;
}

protected void addRouters(List<Router> routers) {
    routers = routers == null ? Collections.emptyList() : routers;
    routerChain.addRouters(routers);
}
```

`AbstractDirectory` 类的声明有如下一段注册也对该特性加以说明了：

> *Invoker list returned from this Directory's list method have been filtered by Routers*

`Directory` 的实现

本文件将只着重该接口实现 `RegistryDirectory` 的分析，其业务逻辑尽管不是太复杂，但结构不是那么明晰，关联内容太多，试图读懂源码剖为困难。但是它的主体接口实现逻辑却出奇的简单，如下源码，禁用时直接抛错，带分组特性时不执行路由筛选处理，尽量尝试应用路由链，若出错则直接返回候选集。

JAVA

```
@Override
public List<Invoker<T>> doList(Invocation invocation) {
    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).");
    }

    if (multiGroup) {
        return this.invokers == null ? Collections.emptyList() : this.invokers;
    }

    List<Invoker<T>> invokers = null;
    try {
        invokers = routerChain.route(getConsumerUrl(), invocation);
    } catch (Throwable t) {
        logger.error("Failed to execute router: " + getUrl() + ", cause: " +
t.getMessage(), t);
    }

    return invokers == null ? Collections.emptyList() : invokers;
}
```

实际该类作为注册中心的客户端，几乎大部分业务代码都在同步注册中心数据，决定是否刷新 `List<Invoker<T>>` 类型的候选集，刷新过程涉及到多个部件的联动，比较复杂，下述章节将逐步展示对其的实现剖析。

同步覆写规则

产生一个服务实例需要基于综合的各种信息，一般它们被携带在配置总线一个URL实例中。然而总线中的信息来源多样，有程序运行时所产生的，也有源于操作系统环境变量、本地文件配置和内存临时配置，还有注册中心的动态配置（包括元数据、覆写规则、注册信息等）。能及时汇总各个维度的信息是保证服务的高可用性的前提，尤其是最后一种跨机配置数据，穿越复杂的网络链路，还需要保障在各个微服务间的视图一致性，因而相比而言存取更加困难，不得不依托于 Zookeeper 这样的第三方分布式协调中间件做配置的同步处理。

本文所讲的配置同步，主要是指覆写规则的同步，它分为两部分，一部分来源于动态配置中心，另一部分则来源于注册中心，尽管有时动态配置中心和注册中心实际上是同一个服务。

从动态配置中心同步 ACL

这种方式的同步主要是基于 `AbstractConfiguratorListener` 类实现。

关于如何实现配置同步不是本文重点，详情请移步《Dubbo 配置管理》，如果深入该文你就会发现动态配置的同步实际上有两种，一种是拉取模式，还一种是基于事件的推送模式，文章还会引导你移步至《Zookeeper 与 Dubbo》，它会告诉你推送模式是如何由第三方客户端驱动框架上层注入的相关监听器同步动态配置的。

如下源码， `initWith(key)` 方法中使用的是拉模式，首先使用用户接口 `Environment` 或者使用SPI加载获取动态配置的客户端—— `DynamicConfiguration` 实例，然后就 key 所对应的 path 加入当前 `ConfigurationListener` 监听器（由this所在类实现），接着使用客户端取得原生的覆写规则，最后由该规则生成一个可用于覆写本地URL数据的 `Configurator` 列表。

JAVA

```
protected final void initWith(String key) {
    DynamicConfiguration dynamicConfiguration =
        DynamicConfiguration.getDynamicConfiguration();

    dynamicConfiguration.addListener(key, this);

    String rawConfig = dynamicConfiguration.
        getRule(key, DynamicConfiguration.DEFAULT_GROUP);

    if (!StringUtils.isEmpty(rawConfig)) {
        genConfiguratorsFromRawRule(rawConfig);
    }
}
```

显然 `initWith(key)` 方法是用在 `AbstractConfiguratorListener` 的实现类的构造函数中的，实际含义是生成动态配置客户端后，立马将覆写规则全量拉取到本地，这是一个必须操作，因推模式需要等到响应的 path 发生了变化。

推模式是基于事件回调机制的，如下：在收到 `ConfigChangeType.DELETED` 事件时，需要把对应的覆写规则处理器列表清理掉，其它事件则重新生成该列表；随后调用有子类覆写的 `notifyOverrides()` 方法告知相关方，覆写规则已经发生变动，请同步变更相应的URL数据。后面这个同步通知一般也都是基于回调相应提供的监听器实现的。

```java
public abstract class AbstractConfiguratorListener implements ConfigurationListener {


    protected List<Configurator> configurators = Collections.emptyList();

    @Override
    public void process(ConfigChangeEvent event) {
        if (logger.isInfoEnabled()) {
            logger.info("Notification of overriding rule, change type is: "
                + event.getChangeType() + ", raw config content is:\n " +
event.getValue());
        }

        if (event.getChangeType().equals(ConfigChangeType.DELETED)) {
            configurators.clear();
        } else {
            if (!genConfiguratorsFromRawRule(event.getValue())) {
                return;
            }
        }

        notifyOverrides();
    }

    protected abstract void notifyOverrides();


    public List<Configurator> getConfigurators() {
        return configurators;
    }

    public void setConfigurators(List<Configurator> configurators) {
        this.configurators = configurators;
    }

    ...
}
```

上述可以看到无论是推模式还是拉模式，都会调用如下实现的 `genConfiguratorsFromRawRule(rawConfig)` 方法，若发生异常，则返回 false，对应上述它被调用的逻辑就是直接返回。相关细节请移步《Dubbo 配置管理》，这里不再赘述。

```java
private boolean genConfiguratorsFromRawRule(String rawConfig) {
    boolean parseSuccess = true;
    try {
        configurators = Configurator.toConfigurators(ConfigParser
            .parseConfigurators(rawConfig)).orElse(configurators);
    } catch (Exception e) {
        logger.error("Failed to parse raw dynamic config and" +
            "it will not take effect, the raw config is: " + rawConfig, e);
        parseSuccess = false;
    }
    return parseSuccess;
}
```

配置中心缺席

然而，在没有单独设置配置中心时，对应的 ConfigurationListener 是不会发生作用的。通过仔细查看扩展点 DynamicConfigurationFactory 的实现就会发现有不对劲的地方，如下：

```java
@SPI("nop")
public interface DynamicConfigurationFactory {

    DynamicConfiguration getDynamicConfiguration(URL url);
}
```

它标注了 @SPI("nop")，也即生成的代理类会将行为委托给如下实现类，从 NopDynamicConfiguration 实现来看，啥事也没干。

```java
public class NopDynamicConfigurationFactory extends AbstractDynamicConfigurationFactory
{
    @Override
    protected DynamicConfiguration createDynamicConfiguration(URL url) {
        return new NopDynamicConfiguration(url);
    }
}

public class NopDynamicConfiguration implements DynamicConfiguration {

    public NopDynamicConfiguration(URL url) {}

    @Override
    public Object getInternalProperty(String key) {}

    @Override
    public void addListener(String key, String group, ConfigurationListener listener) {}

    @Override
    public void removeListener(String key, String group, ConfigurationListener listener)
{}

    @Override
    public String getRule(String key, String group, long timeout) throws
IllegalStateException {
        return null;
    }

    @Override
    public String getProperties(String key, String group, long timeout) throws
IllegalStateException {
        return null;
    }
}
```

ACL 在客户端的应用

上文有关目录服务的介绍中，很明显它是为客户端提供服务的，是客户端微服务发现机制的实现。覆写规则根据作用范围的不同，分为应用级别和微服务级别，因此在 RegistryDirectory 实现中，有两个 AbstractConfiguratorListener 覆写规则监听器实现，分别是 ConsumerConfigurationListener 和 ReferenceConfigurationListener ，二者的代码很简短，都被申明为了静态的私有内部类，需要结合上下文理解。

ConsumerConfigurationListener

从 Directory 接口的声明中可知，每一个被引用微服务对应会拥有它的一个实例。而 ConsumerConfigurationListener 实例在整个应用中也就声明了一个实例全局的实例，独此一份，因所有实例需要关注应用基本的覆写规则的变化，它们自身的 subscribe(url) 方法被调用时，自己就会被作为订阅者增添一个监听器。它们监听的节点都是“/({namespace} | dubbo)/config/dubbo/{app}.configurators”。

当覆写规则有变化时，本地接受到通知后，便刷新应用中的当前所有微服务的所有引用实例。

```java
private static class ConsumerConfigurationListener extends AbstractConfiguratorListener
{
    List<RegistryDirectory> listeners = new ArrayList<>();

    ConsumerConfigurationListener() {
        this.initWith(ApplicationModel.getApplication() + CONFIGURATORS_SUFFIX);
    }

    void addNotifyListener(RegistryDirectory listener) {
        this.listeners.add(listener);
    }

    @Override
    protected void notifyOverrides() {
        listeners.forEach(listener -> listener.refreshInvoker(Collections.emptyList()));
    }
}

private static final ConsumerConfigurationListener CONSUMER_CONFIGURATION_LISTENER
        = new ConsumerConfigurationListener();

public void subscribe(URL url) {
    ...
    CONSUMER_CONFIGURATION_LISTENER.addNotifyListener(this);
    ...
}
```

ReferenceConfigurationListener

同上述不同的是，一个被引用的微服务调用 `subscribe(url)` 时，会为该微服务分配一个 `ReferenceConfigurationListener` 对象，而它监听的节点仅限于自身相关，对应动态配置项为 “`/({namespace} | dubbo)/config/dubbo/{interfaceName}[:{version}][:{group}].configurators`”。

当覆写规则有变化时，本地接受到通知后，便刷新应用中的当前 `RegistryDirectory` 实例所对应微服务。

```java
private ReferenceConfigurationListener serviceConfigurationListener;

private static class ReferenceConfigurationListener extends AbstractConfiguratorListener
{
    private RegistryDirectory directory;
    private URL url;

    ReferenceConfigurationListener(RegistryDirectory directory, URL url) {
        this.directory = directory;
        this.url = url;
        this.initWith(DynamicConfiguration.getRuleKey(url) + CONFIGURATORS_SUFFIX);
    }

    @Override
    protected void notifyOverrides() {
        // to notify configurator/router changes
        directory.refreshInvoker(Collections.emptyList());
    }
}

public void subscribe(URL url) {
    ...
    serviceConfigurationListener = new ReferenceConfigurationListener(this, url);
    ...
}
```

从注册中心同步

因注册中心和动态配置中心可以源于同一服务，而同步他们的数据使用都是订阅观察模式，因此在没有另外提供单独的动态配置中心时，可以以注册中心客户端的身份去同步覆写规则。如果二者同时提供了，就会汇总来自他们的覆写规则，一起发生作用。和其它客户端订阅操作一样，只需配置URL数据，调用如下 `subscribe(url)` 方法即可。

```java
public void subscribe(URL url) {
    setConsumerUrl(url);
    CONSUMER_CONFIGURATION_LISTENER.addNotifyListener(this);
    serviceConfigurationListener = new ReferenceConfigurationListener(this, url);
    registry.subscribe(url, this);
}
```

显然上述代码的目的很明显，就是针对某个特定的被引用微服务做订阅处理（`url` 指定了微服务提供者），等待注册中心相关的通知，在通知中执行某些同步处理。源码中的最后一行调用了 `org.apache.dubbo.registry.RegistryService#subscribe(URL url, NotifyListener listener)`，显然 this 所在的类实现了 `NotifyListener` 接口。明白了这点，咱可以验证其实现方法继续往下探究。

JAVA

```java
@Override
public synchronized void notify(List<URL> urls) {
    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 "";
            }));

    List<URL> configuratorURLs = categoryUrls.getOrDefault(
            CONFIGURATORS_CATEGORY, Collections.emptyList());

    this.configurators = Configurator.toConfigurators(
            configuratorURLs).orElse(this.configurators);

    List<URL> routerURLs = categoryUrls.getOrDefault(
            ROUTERS_CATEGORY, Collections.emptyList());

    toRouters(routerURLs).ifPresent(this::addRouters);

    // providers
    List<URL> providerURLs = categoryUrls.getOrDefault(
            PROVIDERS_CATEGORY, Collections.emptyList());

    refreshOverrideAndInvoker(providerURLs);
}
```

首先上述代码被修饰了方法级别的 `synchronized`，表示针对某个被引用微服务的当前 `Directory` 对象，它本身作为互斥锁，锁住整个 `notify(urls)` 方法，避免前面一个通知还没响应完，后一个就开始执行了。

然后方法体，熟悉 Java 8 的同学会倍感亲切，先将入参做过滤处理，然后分组，按URL数据类型分为覆写规则、路由规则和微服务实例集，最后按分组做相应的业务逻辑处理：

1. 将覆写规则转换为 `Configurator` 覆写规则处理器；
2. 将路由规则数据转换为 `Router` 路由器；
3. 将得到的当前对应引用微服务的所有可用实例，使用 `Configurator` 做覆写刷新处理；

在继续往下探讨前，我们先看看上述源码中出现的两个有关过滤的函数。`isValidCategory(url)` 的大致意思是需要满足条件 `"route" == url.protocol || (url["category"] | "providers") ∈ ["routers", "providers", "configurators", "dynamicconfigurators", "appdynamicconfigurators"]`，而 ``方法则要求 url["compatible_config"]``的值为空。

JAVA

```
private boolean isValidCategory(URL url) {
    String category = url.getParameter(CATEGORY_KEY, DEFAULT_CATEGORY);
    if ((ROUTERS_CATEGORY.equals(category) || ROUTE_PROTOCOL.equals(url.getProtocol()))
||
            PROVIDERS_CATEGORY.equals(category) ||
            CONFIGURATORS_CATEGORY.equals(category) ||
DYNAMIC_CONFIGURATORS_CATEGORY.equals(category) ||
            APP_DYNAMIC_CONFIGURATORS_CATEGORY.equals(category)) {
        return true;
    }
    logger.warn("Unsupported category " + category + " in notified url: " + url + " from
registry " +
            getUrl().getAddress() + " to consumer " + NetUtils.getLocalHost());
    return false;
}

private boolean isNotCompatibleFor26x(URL url) {
    return StringUtils.isEmpty(url.getParameter(COMPATIBLE_CONFIG_KEY));
}
```

另外被调用的 `toRouters(urls)` 方法也值得提一提，如下源码在根据URL数据获取 `Router` 实例时，如果含有 `url["router"]` 参数，则会设 `url.protocol = url["router"]`，原因是 `getRouter(url)` 方法含有 `@Adaptive("protocol")` 声明，SPI机制会在动态生成的代理类中，先使用 `url.protocol` 的值作为 key 去加载它所映射目标 `Router` 类的实例，然后将 `getRouter(url)` 方法委托给它执行。

JAVA

```java
private static final RouterFactory ROUTER_FACTORY =
    ExtensionLoader.getExtensionLoader(RouterFactory.class).getAdaptiveExtension();

private Optional<List<Router>> toRouters(List<URL> urls) {
    if (urls == null || urls.isEmpty()) {
        return Optional.empty();
    }

    List<Router> routers = new ArrayList<>();
    for (URL url : urls) {
        if (EMPTY_PROTOCOL.equals(url.getProtocol())) {
            continue;
        }
        String routerType = url.getParameter(ROUTER_KEY);
        if (routerType != null && routerType.length() > 0) {
            url = url.setProtocol(routerType);
        }
        try {
            Router router = ROUTER_FACTORY.getRouter(url);
            if (!routers.contains(router)) {
                routers.add(router);
            }
        } catch (Throwable t) {
            logger.error("convert router url to router error, url: " + url, t);
        }
    }

    return Optional.of(routers);
}
```

由 `notify(urls)` 方法体中的最后的 `refreshOverrideAndInvoker(urls)` 调用语句，我们会跟踪进入如下 `overrideDirectoryUrl()` 方法，它是我们此刻关注的重点，

JAVA

```
private void overrideDirectoryUrl() {
    this.overrideDirectoryUrl = directoryUrl;
    List<Configurator> localConfigurators = this.configurators;
    doOverrideUrl(localConfigurators);

    List<Configurator> localAppDynamicConfigurators =
        CONSUMER_CONFIGURATION_LISTENER.getConfigurators();
    doOverrideUrl(localAppDynamicConfigurators);

    if (serviceConfigurationListener != null) {
        List<Configurator> localDynamicConfigurators =
            serviceConfigurationListener.getConfigurators();
        doOverrideUrl(localDynamicConfigurators);
    }
}

private void doOverrideUrl(List<Configurator> configurators) {
    if (CollectionUtils.isNotEmpty(configurators)) {
        for (Configurator configurator : configurators) {
            this.overrideDirectoryUrl = configurator.configure(overrideDirectoryUrl);
        }
    }
}
```

上述源码中汇总了 3 种覆写规则：1）来自注册中心的；2）来自配置中心的针对当前应用的；3）来自配置中心的针对当前当前被引用微服务的。将所有这些覆写规则处理器按顺序在 `overrideDirectoryUrl` 这条URL类型的数据均应用一遍，可见来自注册中心的覆写规则优先级更高。

`overrideDirectoryUrl` 最初的样子，也就是 `directoryUrl` 是怎么样的？下面我带你一步步去搜索有关它的一些端倪，先看看如下构造函数，入参 `serviceType` 对应着当前被引用微服务的接口类型，第二个入参 `url` 表示的是当前客户端如何通过注册中心引用目标微服务，被引用微服务的相关元数据被包含在 `refer` 参数中，如 `registry://registry-host/org.apache.dubbo.registry.RegistryService?refer=URL.encode("consumer://consumer-host/com.foo.FooService?version=1.0.0")` 。

也就说这里的 `serviceKey` 表示的是注册中心的客户端服务，如这里的 `org.apache.dubbo.registry.RegistryService` 。

```java
public RegistryDirectory(Class<T> serviceType, URL url) {
    super(url);
    if (serviceType == null) {
        throw new IllegalArgumentException("service type is null.");
    }
    if (url.getServiceKey() == null || url.getServiceKey().length() == 0) {
        throw new IllegalArgumentException("registry serviceKey is null.");
    }
    this.serviceType = serviceType;
    this.serviceKey = url.getServiceKey();

    //①
    this.queryMap = StringUtils.parseQueryString(url.getParameterAndDecoded(REFER_KEY));
    this.overrideDirectoryUrl = this.directoryUrl = turnRegistryUrlToConsumerUrl(url);

    String group = directoryUrl.getParameter(GROUP_KEY, "");
    this.multiGroup = group != null && (ANY_VALUE.equals(group) || group.contains(","));
}
```

代码①处是最初生成 directoryUrl 的地方，做的事情不多，简单的将 refer 参数解码出来转入 queryMap 容器，保留入参 url 中的除参数的其它部分，随后在其后附上 queryMap 中的所有参数，使用如下方法生成了目标 directoryUrl ，从方法名称可以看出，它的目的是将 registry url 转换成 consumer url 。

```java
private URL turnRegistryUrlToConsumerUrl(URL url) {
    // save any parameter in registry that will be useful to the new url.
    String isDefault = url.getParameter(DEFAULT_KEY);
    if (StringUtils.isNotEmpty(isDefault)) {
        queryMap.put(REGISTRY_KEY + "." + DEFAULT_KEY, isDefault);
    }
    return URLBuilder.from(url)
            .setPath(url.getServiceInterface())
            .clearParameters()
            .addParameters(queryMap)
            .removeParameter(MONITOR_KEY)
            .build();
}
```

便为理解，特地通过调试找来了一个实际的例子，如下：

```java
//url (registry url)
zookeeper://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?application=demo-consumer&dubbo=2.0.2&pid=63267&refer=application%3Ddemo-consumer%26check%3Dtrue%26dubbo%3D2.0.2%26interface%3Dorg.apache.dubbo.samples.basic.api.DemoService%26lazy%3Dfalse%26methods%3DsayHello%26pid%3D63267%26register.ip%3D192.168.0.6%26release%3D2.7.3%26side%3Dconsumer%26sticky%3Dfalse%26timestamp%3D1572445953542&release=2.7.3&timestamp=1572445959013

//directoryUrl (consumer url)
zookeeper://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?application=demo-consumer&check=true&dubbo=2.0.2&interface=org.apache.dubbo.samples.basic.api.DemoService&lazy=false&methods=sayHello&pid=63267&register.ip=192.168.0.6&release=2.7.3&side=consumer&sticky=false&timestamp=1572445953542
```

微服务引用实例刷新

在花费了大量篇幅剖析覆写规则同步后，终于轮到本文的主角登场了——微服务引用实例的刷新处理。以水力发电打比方，如果说前面这个覆写规则处理是将水库中的水引入的话，那后面这个刷新处理就相当于将发电机获得的电能输入到储能设备，显然中间还有个发电机将水的势能转换为电能的过程，“势能 → 电能”这个转换过程，对应的由URL数据到 `Invoker` 对象的转换处理，这是一个复杂的过程，详情请移步至《Dubbo微服务注册》。尽管“电能到储能设备的输送”这个过程很简单，但涉及的细节也挺多，下面我们将按粒度由小及大、从分到总来剖析整个过程。

URL数据合入

实际上这是就单个被引用微服务的URL数据应用覆写规则处理器，源码中多次出现 `override > -D > Consumer > Provider` 这一注释，也就说各种覆写规则的应用是有优先级的，Directory URL 覆写处理中已经由提及过。废话不多说，先看如下 `mergeUrl(providerUrl)` 的实现，为了不致信息损失，下面这段代码我们保留了所有注释。

JAVA

```java
//Merge url parameters. the order is: override > -D >Consumer > Provider
private URL mergeUrl(URL providerUrl) {
    providerUrl = ClusterUtils.mergeUrl(providerUrl, queryMap); // Merge the consumer
side parameters

    providerUrl = overrideWithConfigurator(providerUrl);

    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
    this.overrideDirectoryUrl =
this.overrideDirectoryUrl.addParametersIfAbsent(providerUrl.getParameters()); // Merge
the provider side parameters

    if ((providerUrl.getPath() == null || providerUrl.getPath()
            .length() == 0) && DUBBO_PROTOCOL.equals(providerUrl.getProtocol())) { //
Compatible version 1.0
        //fix by tony.chenl DUBBO-44
        String path = directoryUrl.getParameter(INTERFACE_KEY);
        if (path != null) {
            int i = path.indexOf('/');
            if (i >= 0) {
                path = path.substring(i + 1);
            }
            i = path.lastIndexOf(':');
            if (i >= 0) {
                path = path.substring(0, i);
            }
            providerUrl = providerUrl.setPath(path);
        }
    }
    return providerUrl;
}
```

`providerUrl` 是从注册中心通过事件回调同步到本机的，它代表一个被引用微服务中的其中一个实例，就单个实例而言它有可能是朝不保夕的，突然每个事件通知，它就不见了，本地的具体应对情况，下文会提及。

来看看源码中的总体步骤，先合入由当前应用引用服务时传入的客户端参数，然后就该 `providerUrl` 按优先级先后应用所有的覆写规则，最后将来自服务端的参数合入到 `overrideDirectoryUrl` 。

另外还做了兼容处理，在 `dubbo 1.0` 这个版本中，允许注册中心同步下来的 `url.path` 为空，这时需要将 `providerUrl` 的 path 设置为服务接口的名称，它是从 `url["interface"]` 中取得的。但是它的字符串形式可能比较复杂，形如 `[sth + "/"]{interfaceName}[":" + sth2]` ，截取的是 `{interfaceName}` 这一部分。

该章节剩下的代码是关于合入覆写规则的，可以参考Directory URL 覆写处理对照理解。

JAVA

```
private URL overrideWithConfigurator(URL providerUrl) {
    // override url with configurator from "override://" URL for dubbo 2.6 and before
    providerUrl = overrideWithConfigurators(this.configurators, providerUrl);

    // override url with configurator from configurator from "app-name.configurators"
    providerUrl =
overrideWithConfigurators(CONSUMER_CONFIGURATION_LISTENER.getConfigurators(),
providerUrl);

    // override url with configurator from configurators from "service-
name.configurators"
    if (serviceConfigurationListener != null) {
        providerUrl =
overrideWithConfigurators(serviceConfigurationListener.getConfigurators(), providerUrl);
    }

    return providerUrl;
}

private URL overrideWithConfigurators(List<Configurator> configurators, URL url) {
    if (CollectionUtils.isNotEmpty(configurators)) {
        for (Configurator configurator : configurators) {
            url = configurator.configure(url);
        }
    }
    return url;
}
```

候选集过滤处理

微服务开发中中，为了提高可用性，往往一个微服务会部署多个实例。也就是说，注册中心会在数据同步时，为一个可用微服务返回这一到多个可用实例的URL数据，这时就需要系统地对他们进行“URL数据 → Invoker对象”的转换处理。这中间还得更具客户端一些参数做一些过滤处理，该功能统一实现在 `toInvokers(urls)` 方法中，方法体比较长，我们拆开分析。

一般而言一个微服务提供者会实现多种类型的通信协议支持，尽可能满足接入客户端的风格喜好和能力差异，可能会同一个微服务的多个实例各自支持不同协议。接入的客户端若指定了自己所能接受的通讯协议支持集—— `url.protocol` 参数（支持多个时以“,”分割），如果一个被引用微服务实例并不支持该协议，显然这个实例就不应该在候选集中。这部分逻辑体现在循环体中对单个实例如下处理上：

JAVA

```
//TAG: 根据协议支持过滤候选集
private Map<String, Invoker<T>> toInvokers(List<URL> 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 (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;
    }
    ...
}
```

另外如果一个服务实例被标记为 `url["disabled"] = true` 或 `url["enabled"] = false`，那么表示它因为一些特殊原因被禁用了，这个实例也会被排除在候选集之外。如下所示，一个服务实例，若本地缓存不存在时，需要将服务实例的URL数据转化为Invoker对象加入到本地候选集缓存，这时被禁用的实例便被略过了：

JAVA

```
//TAG：到Invoker对象转换时略过被禁用实例
private Map<String, Invoker<T>> toInvokers(List<URL> urls) {
    ...
    try {
        boolean enabled = true;
        if (url.hasParameter(DISABLED_KEY)) {
            enabled = !url.getParameter(DISABLED_KEY, false);
        } else {
            enabled = url.getParameter(ENABLED_KEY, true);
        }
        if (enabled) {
            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);
    }

    ...
}
```

最后再总体的看下 `toInvokers(urls)` 方法的实现，如下，步骤已经很明了，遍历从注册中心同步的目标微服务的所有服务实例的 URL 数据，首先将本地客户端所不支持的通讯协议的实例剔除；然后在当前 `providerUrl` 上应用所有最近同步的覆写规则得到最新的 URL 视图 key，并利用 key 结合 Set 集合特性做排重处理，对于重复的实例数据直接忽略处理；最后将根据 key 找不到的（可能已经缓存在本地）且没被禁用标识的实例转换为 `Invoker` 实例加入。

```java
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(PROTOCOL_KEY);
    for (URL providerUrl : urls) {
        ...//TAG: 根据协议支持过滤候选集

        URL url = mergeUrl(providerUrl);

        String key = url.toFullString();
        if (keys.contains(key)) {
            continue;
        }
        keys.add(key);

        Map<String, Invoker<T>> localUrlInvokerMap = this.urlInvokerMap;
        Invoker<T> invoker = localUrlInvokerMap == null ? null :
localUrlInvokerMap.get(key);
        if (invoker == null) { // Not in the cache, refer again

            ...//TAG: 到Invoker对象转换时略过被禁用实例

        } else {
            newUrlInvokerMap.put(key, invoker);
        }
    }
    keys.clear();
    return newUrlInvokerMap;
}
```

源码中 `localUrlInvokerMap` 相当于新申请了一个指针，指向 `this.urlInvokerMap` 指针所指向某个 `Map<String, Invoker<T>>` 容器，因为 `this.urlInvokerMap` 后面可能会指向新的容器。

一个服务实例的 URL 数据在前后两次事件通知中应用覆写规则后，其值可能保持一样，也有可能因为客户端的一些原因不一样。前者会被直接加入到 `newUrlInvokerMap` 这个新的容器中，而后者对应的 `Invoker` 实例可能已经缓存在 `localUrlInvokerMap` 这个老的容器中，只是因为键发生变化，找不到了，这时只要对应应用了覆写规则的 URL 数据没有被标识禁用，便直接做重新引用处理，加入到 `newUrlInvokerMap` 中。也就是说 Dubbo会不管三七二十一，一个服务实例，其 URL 数据在应用覆写规则后，只要发生变化，便会对其重新引用 。

本地服务实例缓存清理

从上述章节的实现剖析来看，被引用了的微服务实例会被缓存到本地。然而，因为某些原因当前微服务的多个实例中，某些个实例可能会变得不可用，比如运维人员施加了下线处理操作，获得开发人员有目的地禁用某些在线的实例，以便在线排查 bug，或者是当前应用通过动态的写入一些覆写规则过滤掉某些实例。

当注册中心的客户端通过事件回调同步到某个微服务已经不存在相应的实例时，便会经由监听器 `notify(url,listener,urls)` 给监听方只含有一条数据 `urls` 列表，其中URL数据的 `url.protocol` 为空，这时就需要执行如下所有缓存中微服务引用实例的清理处理，简直是毁天灭地。

JAVA

```
private void destroyAllInvokers() {
    Map<String, Invoker<T>> localUrlInvokerMap = this.urlInvokerMap; // local reference
    if (localUrlInvokerMap != null) {
        for (Invoker<T> invoker : new ArrayList<>(localUrlInvokerMap.values())) {
            try {
                invoker.destroy();
            } catch (Throwable t) {
                logger.warn("Failed to destroy service " + serviceKey
                    + " to provider " + invoker.getUrl(), t);
            }
        }
        localUrlInvokerMap.clear();
    }
    invokers = null;
}
```

源码中， `localUrlInvokerMap` 指针所指向的容器中的实例一一调用了 `destroy()` 做销毁处理，最后将所有实例移除。此间 `urlInvokerMap` 的指向有可能发生变化。

另外，如候选集过滤这一章节的最后所介绍的，因为覆写规则有更新，导致某些实例因为应用它们后得到一个全新的 key `url.toFullString()` ，这时Dubbo会做重新引用处理，实际上就是产生了一个新的 `Invoker` 实例，而老的实例实际上还缓存于内存中，这时也需要配合一些清理操作，主要是调用 `invoker.destroy()` 。如下，其处理就是比较新老两个 `Map<String, Invoker<T>>` 集合，如果新集合中为空直接调用 `destroyAllInvokers()` ，否则会先筛选出新的集合中不存在但老的集合中存在的实例，然后再逐个给 `destroy()` 并做回收内存。

JAVA

```
private void destroyUnusedInvokers(Map<String, Invoker<T>> oldUrlInvokerMap,
        Map<String, Invoker<T>> newUrlInvokerMap) {
    if (newUrlInvokerMap == null || newUrlInvokerMap.size() == 0) {
        destroyAllInvokers();
        return;
    }
    // check deleted invoker
    List<String> deleted = null;
    if (oldUrlInvokerMap != null) {
        Collection<Invoker<T>> newInvokers = newUrlInvokerMap.values();
        for (Map.Entry<String, Invoker<T>> entry : oldUrlInvokerMap.entrySet()) {
            if (!newInvokers.contains(entry.getValue())) {
                if (deleted == null) {
                    deleted = new ArrayList<>();
                }
                deleted.add(entry.getKey());
            }
        }
    }

    if (deleted != null) {
        for (String url : deleted) {
            if (url != null) {
                Invoker<T> invoker = oldUrlInvokerMap.remove(url);
                if (invoker != null) {
                    try {
                        invoker.destroy();
                        if (logger.isDebugEnabled()) {
                            logger.debug("destroy invoker[" + invoker.getUrl() + "]
success. ");
                        }
                    } catch (Exception e) {
                        logger.warn("destroy invoker[" + invoker.getUrl()
                            + "] failed. " + e.getMessage(), e);
                    }
                }
            }
        }
    }
}
```

分组处理

Dubbo中如果一个微服务接口有多种实现，可以使用 url["group"] 标识分组，在服务提供端和消费端都根据需要做相应配置，具体可以参考官方服务分组(http://dubbo.apache.org/zh-cn/docs/user/demos/service-group.html)。另外有一类场景是，需要将不同分组的服务做聚合处理，关于这个特性请参考官方分组聚合(http://dubbo.apache.org/zh-cn/docs/user/demos/group-merger.html)，另外在《Dubbo集群 之 容错》一文也特地剖析了其实现。

对于这类带有分组特性的服务实例，Dubbo的目录服务需要另加特殊处理，如下源码所示，先根据 group 信息进行分组，在有多个分组的情况下，对每一个分组先做一次“折叠处理”，也即将同一分组中的多个可用候选Invoker对象伪装成一个虚拟的 Invoker 对象。

JAVA

```
 private static final Cluster CLUSTER =
    ExtensionLoader.getExtensionLoader(Cluster.class).getAdaptiveExtension();

private List<Invoker<T>> toMergeInvokerList(List<Invoker<T>> invokers) {
    List<Invoker<T>> mergedInvokers = new ArrayList<>();
    Map<String, List<Invoker<T>>> groupMap = new HashMap<>();
    for (Invoker<T> invoker : invokers) {
        String group = invoker.getUrl().getParameter(GROUP_KEY, "");
        groupMap.computeIfAbsent(group, k -> new ArrayList<>());
        groupMap.get(group).add(invoker);
    }

    if (groupMap.size() == 1) {
        mergedInvokers.addAll(groupMap.values().iterator().next());
    } else if (groupMap.size() > 1) {
        for (List<Invoker<T>> groupList : groupMap.values()) {//TAG-
            StaticDirectory<T> staticDirectory = new StaticDirectory<>(groupList);
            staticDirectory.buildRouterChain();
            mergedInvokers.add(CLUSTER.join(staticDirectory));
        }
    } else {
        mergedInvokers = invokers;
    }
    return mergedInvokers;
}
```

让我们先梳理下这神奇的过程是怎么发生的？这还得从 Directory、Router、Cluster 三者间的关系说起：它们都是服务于为某个特定被引用微服务的，首先 Directory 根据注册中心同步的数据列出它所有可用实例， Router 则在此基础上根据路由配置做一些过滤筛选处理，得到最终可用的候选集，最后使用某类 Cluster 实现（大部分是一种容错机制)将候选集（一个 List<Invoker> 列表)伪装成一个单一的 Invoker 对象，具体执行RPC调用时，会经由某种 LoadBalance 负载策略从候选集挑选一个 Invoker 实例，将RPC调用委托给该实例执行，如果期间出现异常，则执行重试处理。

其实这一过程正是当前所在类 RegistryDirectory 所参与的，可见上述这个 折叠处理 实际上是在中嵌套了一层相似的操作。 接下来的章节我们来看看这个被嵌套过程中的最核心组成 StaticDirectory 的实现。

StaticDirectory

同 RegistryDirectory 一样， StaticDirectory 也是扩展自 AbstractDirectory 抽象类。它的对象是根据需要随时产生的，用于特定微服务的可用候选集此间不会发生变化，因此实现相对简单很多。核心的代码片段如下，从 doList(invocation) 方法可以看出，如果没有调用 buildRouterChain() 方法，则 list(invocation) 方法直接返回构造函数传入的候选集。

JAVA

```java
public class StaticDirectory<T> extends AbstractDirectory<T> {
    private final List<Invoker<T>> invokers;
    ...

    public void buildRouterChain() {
        RouterChain<T> routerChain = RouterChain.buildChain(getUrl());
        routerChain.setInvokers(invokers);
        this.setRouterChain(routerChain);
    }

    @Override
    protected List<Invoker<T>> doList(Invocation invocation) throws RpcException {
        List<Invoker<T>> finalInvokers = invokers;
        if (routerChain != null) {
            try {
                finalInvokers = routerChain.route(getConsumerUrl(), invocation);
            } catch (Throwable t) {
                logger.error("Failed to execute router: " + getUrl() + ", cause: " +
t.getMessage(), t);
            }
        }
        return finalInvokers == null ? Collections.emptyList() : finalInvokers;
    }
}
```

刷新总流程

在梳理完“电能 → 势能”的这个转换过程的中间细节后，终于到这里，我们可以从更加宏观的视觉来看看微服务引用实例的刷新主流程了。主体流程如下：

- 当注册中心同步回来的 `invokerUrls` 数据明确告知关于当前被引用微服务没有可用对象时：
  a. 将 `forbidden` 标记设置为 true，对外禁用当前 `directory` 对象；
  b. 清空 `invokers` 候选集，并重置 `routerChain` 中的候选集；
  c. 调用 `destroyAllInvokers()` 方法销毁所有的本地 `invoker` 候选对象；
- 入参 `invokerUrls` 数据为空，且最近一次调用 `refresh(invokerUrls)` 缓存在 `cachedInvokerUrls` 的数据也为空，则直接 return 返回；
- 有可用候选集的情况下：
  1. 将 `forbidden` 标记设置为 false；
  2. 重新实例化 `cachedInvokerUrls` 容器对象，记录全部 `invokerUrls` 到其中；
  3. 传入 `invokerUrls` 参数，调用 `toInvokers(urls)` 方法剔除不满足协议要求的数据，得到新的候选集；
  4. 若新的候选集没有可用服务实例，便退出返回；
  5. 将新候选集设给 `routerChain`，由其负责在执行具体RPC请求时经由路由链中对候选集的做进一步筛选过滤处理；

6. 根据是否有分组参数，决定是否对候选集做 **折叠处理**；
7. 调用 `destroyUnusedInvokers(oldUrlInvokerMap, newUrlInvokerMap)` 清理缓存在本地无用候选 `Invoker` 对象。

具体执行流程如下源码所示，

JAVA

```java
private volatile Map<String, Invoker<T>> urlInvokerMap;

private volatile List<Invoker<T>> invokers;~

private volatile Set<URL> cachedInvokerUrls;

private void refreshInvoker(List<URL> invokerUrls) {
    Assert.notNull(invokerUrls, "invokerUrls should not be null");

    if (invokerUrls.size() == 1
            && invokerUrls.get(0) != null
            && EMPTY_PROTOCOL.equals(invokerUrls.get(0).getProtocol())) {
        this.forbidden = true;
        this.invokers = Collections.emptyList();
        routerChain.setInvokers(this.invokers);
        destroyAllInvokers();
    } else {
        this.forbidden = false;
        Map<String, Invoker<T>> oldUrlInvokerMap = this.urlInvokerMap;
        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);
        }
        if (invokerUrls.isEmpty()) {
            return;
        }
        Map<String, Invoker<T>> newUrlInvokerMap = toInvokers(invokerUrls);

        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()));
        routerChain.setInvokers(newInvokers);
        this.invokers = multiGroup ? toMergeInvokerList(newInvokers) : newInvokers;
        this.urlInvokerMap = newUrlInvokerMap;

        try {
            destroyUnusedInvokers(oldUrlInvokerMap, newUrlInvokerMap);
        } catch (Exception e) {
            logger.warn("destroyUnusedInvokers error. ", e);
        }
    }
}
```

源码中用到了几个申明了 `volatile` 修饰符的变量，表示他们是跨线程的共享资源，使用他们的时候有些特点，状态型的变量总是在临界区的入口处改变，而数据型的变量则在出口处改变，中间执行逻辑处理时要妥善对待数据型的变量，因为它随时有可能发生变化，比如调用 `unmodifiableList(list)` 给 `routerChain` 设置的候选集。

完结