【十七】Dubbo微服务导入导出

微服务的架构特点是，服务节点在网络中呈网状分布，尽管节点间以直连形式发生通讯，但在通讯连接建立之前，两节点间是彼此相互孤立的，只有通过注册中心这个第三方媒介的协助，客户端给定名称标识获得服务方的可用物理地址列表，再在此基础上利用某些策略挑选其中一个作为最终的服务提供方。

在具体实现上，类似Zookeeper这类注册中心只是负责了数据的存取和节点相关变化的通知推送而已，从框架上层来看，本地实现的客户端是通过感知相应事件，以异步的方式完成数据的同步的。然而具体实现上，订阅者在发起订阅操作时会主动从注册中心拉取数据，生命周期的此后部分发生变化的数据或子节点们则由回调事件感知，也就是客户端和注册中心存在着推拉结合、推为主的互动模式。这种以响应式为主的数据同步方式的好处是可以节省业务请求之前的准备时间，进而大大的提高服务的可用性。从前面相关文章中我们已经知道，无论是负载均衡、路由处理，还是服务发现其实都是利用这个机制在本机客户端完成的。

基础

《Dubbo集群 之 目录服务》一文中所谓的目录服务实际上就是业界所谓的服务发现，因其在当前客户端以被引用微服务作为粒度单元，它的多个实例组成了一个集群，即便没有集群，单个实例也能通过 `DubboProtocol` 等被单独导入，因而 `RegistryDirectory` 目录服务实现是分属于集群中的。本文要讨论分析的 `RegistryProtocol` 将侧重在服务导出，由于这个动作本身所在当前服务是一个服务提供者实例，和注册中心进行数据同步的主要目的是让服务消费者能感知自身的存在，但是并不需要和同一个服务的其它提供者实例有啥关联处理，因而服务导出，也就本文所讲的服务注册是发生在一个比集群更大粒度的分布式网络中的，这样也就比较容易理解为啥 `RegistryProtocol` 会调用 `RegistryDirectory` 做服务导入处理了。

综上，无论是服务导入，还是服务导出，都需要放到一个比集群更大的粒度——微服务分布式网络中，只有将数据同步到注册中心这个第三方媒介，或者从中同步数据，才能让微服务实例彼此间能发现或者感知对方。假如说 `DubboProtocol` 让一个微服务实例意识到自身个体的存在，那么 `RegistryProtocol` 则是在注册中心的基础上让它意识到个体间关系的存在。

官网中如下实例，服务的导入导出的输入数据源依然是用URL配置总线加以表达的，表征微服务实例或微服务引用实例的URL数据被编码置入到 `url["export"]` 或 `url["refer"]` 这个参数中，也正因为如此，源码中涉及多处相关URL数据的处理。

```java
//① 服务导入
//      1.1）直接导入
"dubbo://service-host/com.foo.FooService?version=1.0.0"
//      1.2）经注册中心导入
"registry://registry-host/org.apache.dubbo.registry.RegistryService?
refer=URL.encode(\"consumer://consumer-host/com.foo.FooService?version=1.0.0\")"

//② 服务导出
//      2.1）直接导出
"dubbo://service-host/com.foo.FooService?version=1.0.0"
//      2.2）经注册中心导出
"registry://registry-host/org.apache.dubbo.registry.RegistryService?
export=URL.encode(\"dubbo://service-host/com.foo.FooService?version=1.0.0\")"
```

如果想要通过本文深入的理解 `RegistryProtocol` ，还是建议想仔细阅读《Dubbo RPC 之Protocol协议层》的三篇文章。在此先再次拧出官方文档中如下关于 `Protocol` 的言简意赅的介绍：

> *RPC 协议扩展，封装远程调用细节。*

对应本文的 `RegistryProtocol` 来说，其实就是将本地同注册中心的数据同步这个细节封装起来，一方面满足了接口实现，另一方面也将注册中心——准确来说是注册中心的客户端同框架上层解耦了。

在进一步剖析源码实现前，先扫清一些认知上的障碍，以便接下来更加系统深入的理解整个实现。

SPI相关

`RegistryProtocol` 是接口 `Protocol` 的一个扩展点具类，每一个具类都是单例的，根据Dubbo的SPI机制，会为扩展点接口动态生成一个代理类，代理类的接口方法实现中，会根据接口本身的相关注解，结合SPI配置文件的映射关系，根据名称获取到它的某个具类的实例，最后将当前方法委托给该实例的对应方法。结合 `@SPI("dubbo")` 这个注解，默认实例的映射名称为 `"dubbo"` ，而 `RegistryProtocol` 被映射为 `registry` 。

如果一个扩展具类中的 `setter` 方法的参数也是一个扩展点，Dubbo的SPI机制会自动完成其单例的装配处理， `RegistryProtocol` 有多个这样的方法，如下：

```java
//@SPI(FailoverCluster.NAME)
private Cluster cluster;
public void setCluster(Cluster cluster) {
    this.cluster = cluster;
}

//@SPI("dubbo")
private Protocol protocol;
public void setProtocol(Protocol protocol) {
    this.protocol = protocol;
}

//@SPI("dubbo")
private RegistryFactory registryFactory;
public void setRegistryFactory(RegistryFactory registryFactory) {
    this.registryFactory = registryFactory;
}

//@SPI("javassist")
private ProxyFactory proxyFactory;
public void setProxyFactory(ProxyFactory proxyFactory) {
    this.proxyFactory = proxyFactory;
}
```

另外如果一个实现了扩展点接口的具类，其构造函数的入参类型也是该扩展点时，那么说明它是一个包装类，这时当前扩展点除包装类外的其它具类均会被所有的包装类给做一次装饰处理，具体行为取决于他们的总和。

根据各自的注解和实现类的情况，对应的默认具类如下包装类体现在第二级上：

- **cluster**：`org.apache.dubbo.rpc.cluster.support.`**`FailoverCluster`**；
  - **MockClusterWrapper**
- **protocol**：`org.apache.dubbo.rpc.protocol.dubbo.`**`DubboProtocol`**；
  - **ProtocolListenerWrapper**、**ProtocolFilterWrapper**、**QosProtocolWrapper**
- **registryFactory**：`org.apache.dubbo.registry.dubbo.`**`DubboRegistryFactory`**；
- **proxyFactory**：`org.apache.dubbo.rpc.proxy.javassist.`**`JavassistProxyFactory`**；
  - **StubProxyFactoryWrapper**

单例模式的 RegistryProtocol

本文所讨论的 `RegistryProtocol` 是一个扩展点，在《Dubbo之SPI扩展点加载》一文中，已经阐述了，于整个应用而言，使用 `ExtensionLoader.getExtensionLoader(SomeClz.class)` 加载的扩展点具类等价于是单例的。如下代码前面那个 public 的构造函数主要是为了赋值全局静态变量 `INSTANCE`，便于后续的引用处理。

```java
public class RegistryProtocol implements Protocol {
    ...
    private static RegistryProtocol INSTANCE;

    public RegistryProtocol() {
        INSTANCE = this;
    }

    public static RegistryProtocol getRegistryProtocol() {
        if (INSTANCE == null) {

ExtensionLoader.getExtensionLoader(Protocol.class).getExtension(REGISTRY_PROTOCOL);
        }
        return INSTANCE;
    }
}
```

> **NOTE**
>
> 从 RegistryProtocol 单例这个角度来看，下文中的 providerConfigurationListener 变量也等价于是单例的。

服务导入

在阅读这一章节的内容之前，最好先熟读《Dubbo集群 之 目录服务》，文中剖析的 RegistryDirectory 存在的目的是为指定的被引用服务接口列出其所有可用的服务实例，该列表会根据注册中心的响应节点变化而动态改变，具体实现上主要仰赖于类似基于和注册中心以事件回调方式同步覆写规则，从而刷新本地缓存的 Invoker 引用实例。

大体步骤

服务导入对外的接口方位为 public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException ，从定义看相当简洁，假如对应 RegistryProtocol 中的实现是给客户端呈上的一道菜，发出 refer(…) 指令后， RegistryDirectory 按指令办事，将原料和佐料准备好后，根据既定的烹饪程序做好这道菜。相对应的我们可以认为：

1. 对应微服务上线的所有实例在注册中心注册的数据节点，以及由配置中心同步的覆写规则这些则可以认为是原料；
2. 入参 url 中参数指定了引用服务时的限定条件，这就相当于是辅料，相当于为适配客户口味而调制的调味剂；
3. 当前客户端基于中心同步事件回调执行的逻辑，类如利用覆写规则执行刷新服务实例的过程，就好比其中一个烹饪环节，而新得到的实例就像烹制好了的整菜的一部分；
4. 烹饪有好几个环节，各个环节的有机组合和应用才能最终做好这道菜，服务导入涉及如下环节：
    - 构建 RegistryDirectory 实例，并为其备好：

    - 用于数据同步的 Registry 实例；
    - 用于单个微服务实例导入的 Protocol 实例；
  - 构建用于客户端执行目标微服务实例集过滤或筛选的的路由链 RouterChain 实例；
  - 到注册中心的为指定 url 数据的客户端订阅特定微服务指定类型节点的变化；
  - 选用合适的容错机制或者其他类型的 Cluster 将服务实例候选集伪装成一个 Invoker<T> 实例；

大体步骤实现源码如下：

```JAVA
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;
}
```

IMPORTANT

每一个被引用微服务在当前客户端均会存在一个 `RegistryDirectory` 实例，其中声明了一个用于装载该服务引用实例的容器—— `volatile List<Invoker<T>> invokers` 。

当发起 `subscribe(subscribeUrl)` 操作后，会间接发起对 `Registry#subscribe(URL url, NotifyListener listener)` 的调用，后面这个订阅处理会确保注册中心有相应节点的路径存在，并随即增加相应的监听器和主动获取被订阅 provider 类型节点的所有子节点（页节点），也即被引用微服务的实例集合，该集合会被转换成对应的 `List<Invoker<T>>` 并赋值给 `invokers` 变量。此后如果注册中心因为有实例的增加或者删减而导致代表实例的页节点有变动时，则会通过监听器知会客户端，这时 `invokers` 变量则会被重新赋值刷新处理。

事件发生前后，若代表服务端实例的URL数据没有变化，则其对应的 `Invoker<T>` 实例会被原样保留，只是引用会被挪入到由 `invokers` 指向的新产生的 `List<Invoker<T>>` 类型容器中。

`RegistryDirectory` 在刷新 `Invoker<T>` 实例时会调用 `protocol.refer(serviceType, url)` ，这里的 `protocol` 是由 `RegistryProtocol` 负责赋值的，负责在协议层完成单个服务实例的引用(也即导入处理)，默认是 `DubboProtocol` 。

处理细节

代码看似很简单，但是隐藏的细节却相当丰富，需要一一详述：

- 基于注册中心的服务导入中，当前客户端自身所关心的数据全部承载在 `regUrl["refer"]` 中，在构建获取 `subscribeUrl` 时，需要先解析得到 `rawUrl = URL.decode(regUrl["refer"])` ，假定 `rawUrl[^"register.ip"]` 表示 `rawUrl` 移除 `"register.ip"` 后所剩的所有参数，则最终 `subscribeUrl` 的构建形式如下：

```
"consumer://" + (rawUrl["register.ip"] | {local ip}) + "/" + {type.getName()} + "?" +
{rawUrl[^"register.ip"]}

//eg:
//consumer://192.168.0.7/org.apache.dubbo.samples.basic.api.DemoService?
//application=demo-consumer&check=true&dubbo=2.0.2&
//interface=org.apache.dubbo.samples.basic.api.DemoService&
//lazy=false&methods=testVoid,sayHello&pid=69391&release=2.7.3&
//side=consumer&sticky=false&timestamp=1573374561281
```

- 在调用 `RegistryDirectory#subscribe(…)` 时，会为入参置 `url["category"] = "providers,configurators,routers"` ，也就是任何以 `RegistryDirectory` 导入的引用微服务均会：1）监听目标微服务的实例上下线情况；2）同步来自注册中心的覆写规则变化，根据需要刷新本地配置；3）路由规则的同步刷新，改变过滤或筛选规则，实际上也就是改变可用的目标服务实例的候选范围；
- 如果没有指定 `regUrl["interface"] = "*"` 和 `regUrl["register"] = false` ，`RegistryProtocol` 会将当前客户端作为节点注册到注册中心，用于获取注册的 `registeredConsumerUrl` 的逻辑代码如下，其值为置 `subscribeUrl["category", "check"] = "consumers", false` 得到，只是在指定 `regUrl["simplified"] = true` 的情况下，其它参数中只保留 `"application"`、`"version"`、`"group"`、`"dubbo"`、`"release"` 这些。

JAVA

```java
public static final String[] DEFAULT_REGISTER_CONSUMER_KEYS = {
        APPLICATION_KEY, VERSION_KEY, GROUP_KEY, DUBBO_VERSION_KEY, RELEASE_KEY
};

public URL getRegisteredConsumerUrl(final URL consumerUrl, URL registryUrl) {
    if (!registryUrl.getParameter(SIMPLIFIED_KEY, false)) {
        return consumerUrl.addParameters(CATEGORY_KEY, CONSUMERS_CATEGORY,
                CHECK_KEY, String.valueOf(false));
    } else {
        return URL.valueOf(consumerUrl, DEFAULT_REGISTER_CONSUMER_KEYS, null)
            .addParameters(CATEGORY_KEY, CONSUMERS_CATEGORY, CHECK_KEY,
String.valueOf(false));
    }
}
```

doRefer(…) 之前

然而基于注册中心的服务导入，在 `doRefer(…)` 之前还有几处细节需要处理。首先需要规整 `regUrl` ，也即设 `regUrl.protocol = (regUrl["registry"] | "dubbo")` ，移除 `regUrl["registry"]` 。其次对于使用 `RegistryProtocol` 引用 `RegistryService` 类型的服务时，是无需经过服务发现机制引用的，因为它不像其他服务一样，行为由远端主机提供，其实现本质而言就是一个注册中心的客户端，远端只负责相关节点及数据的存取，行为则是由本地提供，因此可以通过本机代理机制直接获取到 `RegistryService` 实例。最后如果客户端配置了 `url["group"]` ，则说明需要做结果聚合处理，此时使用的 `Cluster` 则应该是 `MergeableCluster` ，具体参考《Dubbo集群 之 容错》一文中 `Mergeable(结果聚合)` 这一章节内容。

```java
public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {
    url = URLBuilder.from(url)
            .setProtocol(url.getParameter(REGISTRY_KEY, DEFAULT_REGISTRY))
            .removeParameter(REGISTRY_KEY)
            .build();
    Registry registry = registryFactory.getRegistry(url);
    if (RegistryService.class.equals(type)) {
        return proxyFactory.getInvoker((T) registry, type, url);
    }

    // group="a,b" or group="*"
    Map<String, String> qs =
StringUtils.parseQueryString(url.getParameterAndDecoded(REFER_KEY));
    String group = qs.get(GROUP_KEY);
    if (group != null && group.length() > 0) {
        if ((COMMA_SPLIT_PATTERN.split(group)).length > 1 || "*".equals(group)) {
            return doRefer(getMergeableCluster(), registry, type, url);
        }
    }
    return doRefer(cluster, registry, type, url);
}

private Cluster getMergeableCluster() {
    return ExtensionLoader.getExtensionLoader(Cluster.class).getExtension("mergeable");
}
```

服务导出

同样，微服务的导出也是相对整个微服务分布式网络而言，正如上文所述，一个微服务虽然绝大部分时刻是以集群的形式对外提供服务的，但是的就单个服务实例而言，它并不需要知道这些信息，只有服务的消费者在发起具体请求时需要知晓，也即集群信息是由客户端在注册中心的协助下各自独立维护的。

然而，就如同《Dubbo 配置管理》一文中的开头部分所言，微服务的配置管理离不开注册中心这种分布式协调框架的支持。

由于相关源码牵涉比较多的细节，没法一览知义，下述由浅及深，逐个击破。

相关 URL 数据处理

Dubbo在生成本地微服务实例的初始阶段时，需要先经过配置层的数据读入处理，然后经由框架代理层将对应接口实现转换成对应的一个原始 `Invoker<T>` 对象—— `originInvoker` ，通过该对象的 `getUrl()` 方法能获得原始的URL数据—— `regUrl` 。然后有两种方式可以表示当前微服务使用何种注册中心导出，分别是(此处假设使用 `zookeeper` 作为注册中心)：1） `regUrl.protocol = "registry"` 并且 `regUrl["registry"] = "zookeeper"` ；2） `regUrl.protocol = "zookeeper"` 。针对第一种情况， `RegistryProtocol` 在执行服务到处时会使用如下 `getRegistryUrl(originInvoker)` 获得统一表示，也即第二种的标准表示，同时会移除 `regUrl["registry"]` 参数，若没有明确指定该参数，则会设 `regUrl.protocol = "dubbo"` 。

```java
private URL getRegistryUrl(Invoker<?> originInvoker) {
    URL registryUrl = originInvoker.getUrl();
    if (REGISTRY_PROTOCOL.equals(registryUrl.getProtocol())) {
        String protocol = registryUrl.getParameter(REGISTRY_KEY, DEFAULT_REGISTRY);
        registryUrl = registryUrl.setProtocol(protocol).removeParameter(REGISTRY_KEY);
    }
    return registryUrl;
}
```

`regUrl["export"]` 编码封装了当前被导出微服务本身的信息，需经过 `getProviderUrl(originInvoker)` 解码获得其URL数据——`providerUrl`。

```java
private URL getProviderUrl(final Invoker<?> originInvoker) {
    String export = originInvoker.getUrl().getParameterAndDecoded(EXPORT_KEY);
    if (export == null || export.length() == 0) {
        throw new IllegalArgumentException("The registry export url is null! registry:
" + originInvoker.getUrl());
    }
    return URL.valueOf(export);
}
```

Dubbo的注册中心中存在一类 `"configurators"` 节点，一个微服务的相关的覆写规则会作为其子节点出现。其完整URL数据表示——`overrideSubscribeUrl`，是在 `providerUrl` 的基础上获得的，也即设 `providerUrl["category", "check"] = "configurators", false`、`providerUrl.protocol = "provider"`。当然，它也是服务实例用于订阅配置类节点的。

```java
private URL getSubscribedOverrideUrl(URL registeredProviderUrl) {
    return registeredProviderUrl.setProtocol(PROVIDER_PROTOCOL)
            .addParameters(CATEGORY_KEY, CONFIGURATORS_CATEGORY, CHECK_KEY,
String.valueOf(false));
}
```

URL配置总线在Dubbo中作为载体起到了上下文参数存取和传递的作用，然而环节传递过程中并不是毫无保留的全盘脱出，多出的参数会扰乱下一环的业务处理，也可会造成某些不必要的数据泄露风险，因此无论这种传递是跨方法的还是跨服务进程的，都会经过必要的筛选处理，或增或减，抑或重新组装URL实例。

一个服务实例的大部分配置数据都有可能装载在代表它的URL数据中——`providerUrl = regUrl["export"]`，其中一部分仅限于本实例使用，集群中其它实例或者它的消费者并不需要知晓，或者说不应该暴露给它们。换言之，代表服务实例完成到注册中心注册的URL数据——`registeredProviderUrl`，应该是由 `providerUrl` 裁剪得到的，其获取方式有如下：

1. 检验是否含有 `regUrl["simplified"] = true`：
2. 无，默认情况，去掉 `providerUrl` 中的如下参数：

- 带 "." 前缀的参数；
- "monitor"、"bind.ip"、"bind.port"、"qos.enable"、"qos.host"、"qos.port"、"qos.accept.foreign.ip"、"validation"、"interfaces"

3. 有，按如下步骤组装URL数据：

   a. 保留 "application"、"codec"、"exchanger"、"serialization"、"cluster"、"connections"、"deprecated"、"group"、"loadbalance"、"mock"、"path"、"timeout"、"token"、"version"、"warmup"、"weight"、"timestamp"、"dubbo"、"release" 这些参数；

   b. Dubbo优先使用 url["interface"] 参数表示服务接口，没有该参数的情况下使用 url.path，前者存在的情况下，若和后者不一样，也需要保留；

   c. url["extra-keys"] 也参数原样保留；

   d. 另外参数附有方法前缀的也愿意保留，前缀满足 prefix ∈ url["methods"]（","逗号分隔的方法名称)；

```java
private URL getRegisteredProviderUrl(final URL providerUrl, final URL registryUrl) {
    if (!registryUrl.getParameter(SIMPLIFIED_KEY, false)) {
        return providerUrl.removeParameters(getFilteredKeys(providerUrl))
            .removeParameters(MONITOR_KEY, BIND_IP_KEY, BIND_PORT_KEY, QOS_ENABLE,
                QOS_HOST, QOS_PORT, ACCEPT_FOREIGN_IP, VALIDATION_KEY, INTERFACES);
    } else {
        String extraKeys = registryUrl.getParameter(EXTRA_KEYS_KEY, "");
        if (!providerUrl.getPath().equals(providerUrl.getParameter(INTERFACE_KEY))) {
            if (StringUtils.isNotEmpty(extraKeys)) {
                extraKeys += ",";
            }
            extraKeys += INTERFACE_KEY;
        }
        String[] paramsToRegistry = getParamsToRegistry(DEFAULT_REGISTER_PROVIDER_KEYS
                , COMMA_SPLIT_PATTERN.split(extraKeys));
        return URL.valueOf(providerUrl, paramsToRegistry,
            providerUrl.getParameter(METHODS_KEY, (String[]) null));
    }

}

private static String[] getFilteredKeys(URL url) {
    Map<String, String> params = url.getParameters();
    if (CollectionUtils.isNotEmptyMap(params)) {
        return params.keySet().stream()
                .filter(k -> k.startsWith(HIDE_KEY_PREFIX))
                .toArray(String[]::new);
    } else {
        return new String[0];
    }
}

public static final String[] DEFAULT_REGISTER_PROVIDER_KEYS = {
        APPLICATION_KEY, CODEC_KEY, EXCHANGER_KEY, SERIALIZATION_KEY,
        CLUSTER_KEY, CONNECTIONS_KEY, DEPRECATED_KEY,
        GROUP_KEY, LOADBALANCE_KEY, MOCK_KEY, PATH_KEY, TIMEOUT_KEY,
        TOKEN_KEY, VERSION_KEY, WARMUP_KEY,
        WEIGHT_KEY, TIMESTAMP_KEY, DUBBO_VERSION_KEY, RELEASE_KEY
};

public String[] getParamsToRegistry(String[] defaultKeys, String[]
additionalParameterKeys) {
    int additionalLen = additionalParameterKeys.length;
    String[] registryParams = new String[defaultKeys.length + additionalLen];
    System.arraycopy(defaultKeys, 0, registryParams, 0, defaultKeys.length);
    System.arraycopy(additionalParameterKeys, 0,
        registryParams, defaultKeys.length, additionalLen);
    return registryParams;
}
```

ACL 在服务端的应用

《Dubbo集群 之 目录服务》一文中已经花费大量篇幅，深刻阐述了利用覆写规则同步刷新微服务引用实例的实现，与之相似，当本地服务实例监听到来自系统维护人员通过配置中心修改相关配置的事件后，也会对实例做相应的刷新处理。在其“同步覆写规则”这一章节中已经介绍过，本地接受到的事件中会含有对应的覆写规则的文本数据， AbstractConfiguratorListener 会将其装换成对应的 List<Configurator> configurators 覆写规则处理器列表，实现类会在需要是调用 configurators 改写代表实例的URL数据，正如下述 getConfigedInvokerUrl(configurators, url) 所实现的逻辑那样。而方法最终返回的URL数据则是用于产生新的实例，并替换掉旧的那个。

```JAVA
private static URL getConfigedInvokerUrl(List<Configurator> configurators, URL url) {
    if (configurators != null && configurators.size() > 0) {
        for (Configurator configurator : configurators) {
            url = configurator.configure(url);
        }
    }
    return url;
}
```

相似地，由于一个应用中可以存在多个微服务，因而在服务端依然按照应用级和服务级分别同步覆写规则，对应提供 AbstractConfiguratorListener 抽象类的扩展实现——ProviderConfigurationListener 和 ServiceConfigurationListener ，分别订阅配置中心对应的 "/({namespace} | dubbo)/config/dubbo/{app}.configurators" 节点和“/({namespace} | dubbo)/config/dubbo/{interfaceName}[:{version}][:{group}].configurators”节点，它们都含有如下一个覆写URL数据的方法：

```JAVA
private class (ServiceConfigurationListener | ProviderConfigurationListener) extends AbstractConfiguratorListener{
    ...
    private <T> URL overrideUrl(URL url) {
        return RegistryProtocol.getConfigedInvokerUrl(configurators, url);
    }
    ...
}
```

AbstractConfiguratorListener 的扩展实现类会在构造函数调用其定义的 initWith(key) 方法，一旦被实例化，也意味着该方法被调用，随后便会主动从配置中心的由 key 代表的对应节点拉取到覆写规则的文本数据，并被转换成 Configurator 对象装入 configurators 容器中，而后续如果相关的治理操作改写了规则，那么 ConfigurationListener 监听器实现会被触发，回调逻辑中会对 configurators 重新赋值。

显然，从属于应用的微服务，在应用覆写规则刷新实例时，需要综合应用级别和自身服务级别的覆写规则，如下，两次调用 overrideUrl(url) 这个方法。

```java
private final Map<String, ServiceConfigurationListener> serviceConfigurationListeners
        = new ConcurrentHashMap<>();

private final ProviderConfigurationListener providerConfigurationListener
        = new ProviderConfigurationListener();

private URL overrideUrlWithConfig(URL providerUrl, OverrideListener listener) {
    providerUrl = providerConfigurationListener.overrideUrl(providerUrl);
    ServiceConfigurationListener serviceConfigurationListener =
        new ServiceConfigurationListener(providerUrl, listener);
    serviceConfigurationListeners.put(providerUrl.getServiceKey(),
serviceConfigurationListener);
    return serviceConfigurationListener.overrideUrl(providerUrl);
}
```

从上述源码中不难发现，当前服务端应用的每一个微服务实例均会对应存在一个 `ServiceConfigurationListener` 实例，该实例中绑定了一个 `OverrideListener` 对象，其定义的方法 `doOverrideIfNecessary()` 正是用于实现服务实例刷新的，也被认为是重新导出。该方法会在父类定义的回调方法 `notifyOverrides()` 的实现中被调用，如下源码，也就是说服务治理引发的事件驱动着服务实例的重新导出处理。

```java
private class ProviderConfigurationListener extends AbstractConfiguratorListener {
    ...
    @Override
    protected void notifyOverrides() {
        overrideListeners.values().forEach(listener -> ((OverrideListener)
listener).doOverrideIfNecessary());
    }
}

private class ServiceConfigurationListener extends AbstractConfiguratorListener {
    ...
    @Override
    protected void notifyOverrides() {
        notifyListener.doOverrideIfNecessary();
    }
}
```

可见应用级别的覆写规则会引起对应应用中的所有微服务的 `doOverrideIfNecessary()` 方法的回调，这里我们可以认为 `overrideListeners.values()` 等价于从 `serviceConfigurationListeners.values()` 集合中执行 `map(v → v.notifyListener)` 所得，具体情况下文会涉及。

OverrideListener

上述章节已经说明了 `OverrideListener` 是利用事件回调机制同步覆写规则，从而执行服务实例刷新的。该类实现了 `NotifyListener` 接口，而后者是注册中心客户端所定义的，用于在被关注的节点或节点相关数据变化时，回调指定的业务逻辑。也就是说覆写规则的数据同步方案实际上是有两种实现方

案，一种是拥有单独的配置中心，另外一种直接利用注册中心，如果两种都有的话，则会共同发生作用。没有提供对应的配置中心实现时，相应 `ConfigurationListener` 接口实现就不会发生作用。

先看看对应 `doOverrideIfNecessary()` 方法的实现，如下，步骤很清晰：

1. 首先由 `URL.decode(regUrl["export"])` 解析得到 `originUrl` ；
2. 然后根据它计算出 `key` 值，并由该 `key` 从 `bounds` 取得与 `originInvoker` 对应的 `ExporterChangeableWrapper` 实例 `exporter` ，它的 `invoker` 变量缓存了 `originInvoker` 经过规则覆写后的版本；
3. 随后经 `exporter.getInvoker().getUrl()` 得到最近被覆写过的URL数据 `currentUrl` ；
4. 接着对 `originUrl` 应用同步于注册中心和配置中心的覆写规则，得到新的URL数据 `newUrl` ；
5. 最后若 `currentUrl.equals(newUrl)` ，则表示当前发生的覆写操作并没有引起URL数据的变化，只有不相等时才会执行对应服务实例 `originInvoker` 的重新导出处理；

JAVA

```java
private class OverrideListener implements NotifyListener {

    private final URL subscribeUrl;

    private final Invoker originInvoker;

    private List<Configurator> configurators;

    public OverrideListener(URL subscribeUrl, Invoker originalInvoker) {
        this.subscribeUrl = subscribeUrl;
        this.originInvoker = originalInvoker;
    }
    ...

    public synchronized void doOverrideIfNecessary() {
        final Invoker<?> invoker;
        if (originInvoker instanceof InvokerDelegate) {
            invoker = ((InvokerDelegate<?>) originInvoker).getInvoker();
        } else {
            invoker = originInvoker;
        }
        URL originUrl = RegistryProtocol.this.getProviderUrl(invoker);
        String key = getCacheKey(originInvoker);
        ExporterChangeableWrapper<?> exporter = bounds.get(key);
        if (exporter == null) {
            logger.warn(new IllegalStateException("error state, exporter should not be
null"));
            return;
        }
        //The current, may have been merged many times
        URL currentUrl = exporter.getInvoker().getUrl();
        //Merged with this configuration
        URL newUrl = getConfigedInvokerUrl(configurators, originUrl);
        newUrl =
getConfigedInvokerUrl(providerConfigurationListener.getConfigurators(), newUrl);
        newUrl =
getConfigedInvokerUrl(serviceConfigurationListeners.get(originUrl.getServiceKey())
                .getConfigurators(), newUrl);
        if (!currentUrl.equals(newUrl)) {
            RegistryProtocol.this.reExport(originInvoker, newUrl);
            logger.info("exported provider url changed, origin url: " + originUrl +
                    ", old export url: " + currentUrl + ", new export url: " + newUrl);
        }
    }
}
```

剩下有关的实现是同于同步注册中心的覆写规则，如下源码，先对回调事件的节点列表执行执行匹配检查，如果没有匹配则直接返回，否则将所有匹配的URL数据——`url.protocol = "override"` 或 url["category"] = "configurators" —转换成覆写规则处理器，最后再同样调用 doOverrideIfNecessary()``执行服务实例的重新导出处理。

JAVA

```
private class OverrideListener implements NotifyListener {
    ...
    public synchronized void notify(List<URL> urls) {
        logger.debug("original override urls: " + urls);

        List<URL> matchedUrls = getMatchedUrls(urls,
subscribeUrl.addParameter(CATEGORY_KEY,
                CONFIGURATORS_CATEGORY));
        logger.debug("subscribe url: " + subscribeUrl + ", override urls: " +
matchedUrls);

        // No matching results
        if (matchedUrls.isEmpty()) {
            return;
        }

        this.configurators = Configurator.toConfigurators(classifyUrls(matchedUrls,
UrlUtils::isConfigurator))
                .orElse(configurators);

        doOverrideIfNecessary();
    }
    private List<URL> getMatchedUrls(List<URL> configuratorUrls, URL currentSubscribe)
{
        List<URL> result = new ArrayList<URL>();
        for (URL url : configuratorUrls) {
            URL overrideUrl = url;
            // Compatible with the old version
            if (url.getParameter(CATEGORY_KEY) == null &&
OVERRIDE_PROTOCOL.equals(url.getProtocol())) {
                overrideUrl = url.addParameter(CATEGORY_KEY, CONFIGURATORS_CATEGORY);
            }

            // Check whether url is to be applied to the current service
            if (UrlUtils.isMatch(currentSubscribe, overrideUrl)) {
                result.add(url);
            }
        }
        return result;
    }
}

public static boolean UrlUtils#isConfigurator(URL url) {
    return OVERRIDE_PROTOCOL.equals(url.getProtocol()) ||
            CONFIGURATORS_CATEGORY.equals(url.getParameter(CATEGORY_KEY,
DEFAULT_CATEGORY));
}
```

需要注意的是，老的版本中，一个表示配置类的节点，其 `url.protocol = "override"` ，而新版本则用 `url["category"] = "configurators"` 配置项加以表达。为了适配 `isMatch(consumerUrl, providerUrl)` （`没有要求“url.protocol”也要匹配`），特针对老版本配置类的URL数据中临时加上该项。

ExporterChangeableWrapper

根据 Protocol 的定义，服务导出后需要返回一个对应的 Exporter 实例，其目的主要是用于入参 Invoker<T> 实例相关的销毁处理。对应 RegistryProtocol 中的业务逻辑就是为当前服务实例执行如下动作：

1. 从注册中心注销，也即相应 provider 类 节点解注册；
2. 删除用于同步 注册中心 覆写规则的监听器，也即解订阅相应 configurators 类 节点；
3. 删除用于同步 配置中心 覆写规则的监听器，也即解订阅相应的“/({namespace} | dubbo)/config/dubbo/{interfaceName}[:{version}][:{group}].configurators”节点；
4. 最后利用线程池异步调用 exporter.unexport() 方法最终完成销毁处理，其中 exporter 是用于完成服务实例的本机销毁处理的；

```java
private class ExporterChangeableWrapper<T> implements Exporter<T> {
    ...
    @Override
    public void unexport() {
        String key = getCacheKey(this.originInvoker);
        bounds.remove(key);

        Registry registry = RegistryProtocol.INSTANCE.getRegistry(originInvoker);
        try {
            registry.unregister(registerUrl);
        } catch (Throwable t) {
            logger.warn(t.getMessage(), t);
        }
        try {
            NotifyListener listener = RegistryProtocol.INSTANCE
                .overrideListeners.remove(subscribeUrl);
            registry.unsubscribe(subscribeUrl, listener);
            DynamicConfiguration.getDynamicConfiguration()
                    .removeListener(subscribeUrl.getServiceKey() +
CONFIGURATORS_SUFFIX,

serviceConfigurationListeners.get(subscribeUrl.getServiceKey()));
        } catch (Throwable t) {
            logger.warn(t.getMessage(), t);
        }

        executor.submit(() -> {
            try {
                int timeout = ConfigurationUtils.getServerShutdownTimeout();
                if (timeout > 0) {
                    logger.info("Waiting " + timeout
                        + "ms for registry to notify all consumers before unexport. " +
                            "Usually, this is called when you use dubbo API");
                    Thread.sleep(timeout);
                }
                exporter.unexport();
            } catch (Throwable t) {
                logger.warn(t.getMessage(), t);
            }
        });
    }
}
```

解注册或者解订阅是一个网络I/O操作，总共涉及 3 个这样的操作，耗时相对会比较长，且没法准确预估全部完成的时间，因此使用了配置的大概时间延时执行 `exporter` 的销毁处理，超时配置为 `conf["dubbo.service.shutdown.wait"]` 或 `conf["dubbo.service.shutdown.wait.seconds"]`。

由上文已知，本机导出的初始服务实例记为 `originInvoker`，此后经通知事件同步覆写规则时都是基于它执行刷新进而得到一个新的 `<Invoker<T>, Exporter<T>>` 对象组合的。因而 `originInvoker` 被声明成了 `final` 型，而 `exporter` 却是可变的，而这也是类名含有 `Changeable` 字样的奥义所在，如下所示：

```java
private class ExporterChangeableWrapper<T> implements Exporter<T> {
    ...
    private final Invoker<T> originInvoker;
    private Exporter<T> exporter;
    public ExporterChangeableWrapper(Exporter<T> exporter, Invoker<T> originInvoker) {
        this.exporter = exporter;
        this.originInvoker = originInvoker;
    }

    public Invoker<T> getOriginInvoker() {
        return originInvoker;
    }

    @Override
    public Invoker<T> getInvoker() {
        return exporter.getInvoker();
    }

    public void setExporter(Exporter<T> exporter) {
        this.exporter = exporter;
    }

    private URL subscribeUrl;
    private URL registerUrl;

    public void setSubscribeUrl(URL subscribeUrl) {
        this.subscribeUrl = subscribeUrl;
    }

    public void setRegisterUrl(URL registerUrl) {
        this.registerUrl = registerUrl;
    }
}
```

源码最后呈现的 `subscribeUrl` 和 `registerUrl` ，一个用于订阅 `configurators` 类 节点，另一个则用于注册一个 `provider` 类 节点。由于 `provider` 类 节点是一个服务实例的可公示数据的完整URL表示，因此经过应用覆写规则后， `registerUrl` 是会发生变化的。

InvokerDelegate<T>

可以说， `InvokerDelegate<T>` 这个公有静态内部类是整个 `RegistryProtocol` 源码中涉及代码最少，但理解上却最不直观的一个类，为啥需要它，它到底有啥作用?

先看看其父类 `InvokerWrapper<T>` ， `Wrapper` 的含义是采用委托方式实现某一接口方法，而被委托对象(实现同一接口)的行为被封装了， `Wrapper` 类可以对其行为进行改写或者隐藏，如下述源码所示，被委托的 `invoker` 变量是有自己的 `getUrl()` 实现的，但是 `InvokerWrapper<T>` 却利用构造函数传入的 `url` 将其隐藏了，调用同一方法将会得到该 `url` 。

```java
public class InvokerWrapper<T> implements Invoker<T> {

    private final Invoker<T> invoker;

    private final URL url;

    public InvokerWrapper(Invoker<T> invoker, URL url) {
        this.invoker = invoker;
        this.url = url;
    }

    @Override
    public URL getUrl() {
        return url;
    }
    ...//利用委托机制直接实现所有Invoker<T>接口的其它方法
}
```

再回到 InvokerDelegate<T> 本身，首先它新声明的 invoker 属性“覆写”了父类所定义的，行为上没有发生变化，但是解决了父类中由于 invoker 被申明为私有而无法访问的问题。其它相比而言只增加了一个 getInvoker() 方法，原因是内嵌的 invoker 可能也是一个 InvokerDelegate<T> 类对象，这种情况下只有通过 instanceof 类型判断才能递归获取到最初被封装的那个 Invoker<T> 类对象。

```java
public static class InvokerDelegate<T> extends InvokerWrapper<T> {
    private final Invoker<T> invoker;

    public InvokerDelegate(Invoker<T> invoker, URL url) {
        super(invoker, url);
        this.invoker = invoker;
    }

    public Invoker<T> getInvoker() {
        if (invoker instanceof InvokerDelegate) {
            return ((InvokerDelegate<T>) invoker).getInvoker();
        } else {
            return invoker;
        }
    }
}
```

上文中关于同步覆写规则处理的剖析中，有出现过类似的一段代码，根据其应用，我们知道其目的是为了获取最初服务实例在本地导出时所输入的 providerUrl 。

```java
private class OverrideListener implements NotifyListener {
    ...
    public synchronized void doOverrideIfNecessary() {
        final Invoker<?> invoker;
        if (originInvoker instanceof InvokerDelegate) {
            invoker = ((InvokerDelegate<?>) originInvoker).getInvoker();
        } else {
            invoker = originInvoker;
        }
        ...
    }
}
```

doLocalExport(…) 和 doChangeLocalExport(…)

见名知意，二者对应的是本地的导出处理，分别对应了服务实例的初始导出过程和同步覆写规则时的重新导出过程。显然，这里的本地导出的主要过程是由 `protocol` ，比如说 `DubboProtocol` 来完成的。

上述曾提及Dubbo的框架代理层为当前微服务所最初产生 `Invoker<T>` 实例被记为 `originInvoker` ，其URL数据表示是一个包含了与注册相关信息的完整 `regUrl` ，真正代表本尊的URL数据 `providerUrl` 需要另行解析，并且此后随着来自于配置中心的覆写规则同步，它会发生变化。然而，业务逻辑是随代码固化下来了的，能改变的是相关配置，比如实例所运行的上下文环境、业务相关参数，也就是说变化的只是代表 `originInvoker` 的URL数据。因此具体实现时，`originInvoker` 会被封入到一个 `InvokerDelegate<T>` 类型对象中。一方面可以确保框架后续流程中能够直接获取到服务实例的 `providerUrl` ，避免每次都需要在 `regUrl` 上另加解析，顶层并不需要或者关心该 `regUrl` 。另一方面，框架代理层只需执行一次 `originInvoker` 的生成处理。

章节ExporterChangeableWrapper中已经阐明微服务实例的销毁是一个必须的I/O流程，销毁是以 `originInvoker` 作为参考坐标系的，即便是在并发环境下，来自注册中心或配置中心的覆写规则同步事件可能随时发生，但任意时刻于特定微服务来说当前应用只会存在一个对应的 `Invoker<T>` 实例，初次导出时是 `originInvoker` ，此后则是一个封入了它的 `InvokerDelegate<T>` 类型的包装对象 `delegateInvoker` ，。因而组合了 `originInvoker` 、 `delegateInvoker` 、`delegateInvoker’sExporter` 三者的 `ExporterChangeableWrapper` 类型对象会使用 `ConcurrentMap<String, ExporterChangeableWrapper<?>>` 类型的安全并发容器 `bounds` 做存取处理，键取 `URL.decode(regUrl["export"])[^["dynamic", "enabled"]]` 。

```java
private final ConcurrentMap<String, ExporterChangeableWrapper<?>> bounds = new
ConcurrentHashMap<>();

private <T> ExporterChangeableWrapper<T> doLocalExport(final Invoker<T> originInvoker,
URL providerUrl) {
    String key = getCacheKey(originInvoker);

    return (ExporterChangeableWrapper<T>) bounds.computeIfAbsent(key, s -> {
        Invoker<?> invokerDelegate = new InvokerDelegate<>(originInvoker, providerUrl);
        return new ExporterChangeableWrapper<>(
            (Exporter<T>)protocol.export(invokerDelegate), originInvoker);
    });
}

private <T> ExporterChangeableWrapper doChangeLocalExport(final Invoker<T>
originInvoker, URL newInvokerUrl) {
    String key = getCacheKey(originInvoker);
    final ExporterChangeableWrapper<T> exporter = (ExporterChangeableWrapper<T>)
bounds.get(key);
    if (exporter == null) {
        logger.warn(new IllegalStateException("error state, exporter should not be
null"));
    } else {
        final Invoker<T> invokerDelegate = new InvokerDelegate<T>(originInvoker,
newInvokerUrl);
        exporter.setExporter(protocol.export(invokerDelegate));
    }
    return exporter;
}


//URL.decode(regUrl["export"])[^["dynamic", "enabled"]]
private String getCacheKey(final Invoker<?> originInvoker) {
    URL providerUrl = getProviderUrl(originInvoker);
    String key = providerUrl.removeParameters("dynamic", "enabled").toFullString();
    return key;
}
```

export(Invoker<T>) 导出主流程

终于，在理清楚所有细节后，可以进入到主流程看看具体的导出过程了。下述是所有相关剩下的源码，整体过程如下：

1. 首先，基于从框架代理层生成的 `originInvoker` 对象获得 `regUrl` 、 `providerUrl` 、 `overrideSubscribeUrl` 这 3 个URL数据；
2. 然后，创建并增设用于从配置中心同步覆写规则的两级监听器，并完成 `providerUrl` 的初始化时的改写处理，基于已改写的 `providerUrl` 执行 `originInvoker` 的本地导出处理，得到 `ExporterChangeableWrapper<T>` 类型的 `exporter` 对象；

3. 紧接着去除 `providerUrl` 中只用于服务实例本地总线参数，生成 `registeredProviderUrl` ，同时获取应用层提供的注册中心实例 `registry` ，使用二者完成完成当前服务实例到注册中心的登记处理；
4. 将第二个步骤产生的 `OverrideListener` (实现了 NotifyListener 接口) 监听器设置到 `registry` 的 `overrideSubscribeUrl` 这个 `configurators` 类型的页节点上；
5. 最后，完善 `exporter` 对象的填值处理，创建并返回一个封装了它的 `DestroyableExporter<T>` 对象；

```java
private final Map<URL, NotifyListener> overrideListeners = new ConcurrentHashMap<>();

public <T> Exporter<T> export(final Invoker<T> originInvoker) throws RpcException {
    URL registryUrl = getRegistryUrl(originInvoker);
    // url to export locally
    URL providerUrl = getProviderUrl(originInvoker);

    final URL overrideSubscribeUrl = getSubscribedOverrideUrl(providerUrl);
    final OverrideListener overrideSubscribeListener =
        new OverrideListener(overrideSubscribeUrl, originInvoker);
    overrideListeners.put(overrideSubscribeUrl, overrideSubscribeListener);

    providerUrl = overrideUrlWithConfig(providerUrl, overrideSubscribeListener);
    final ExporterChangeableWrapper<T> exporter = doLocalExport(originInvoker,
providerUrl);

    final Registry registry = getRegistry(originInvoker);
    final URL registeredProviderUrl = getRegisteredProviderUrl(providerUrl,
registryUrl);
    ProviderInvokerWrapper<T> providerInvokerWrapper =
        ProviderConsumerRegTable.registerProvider(originInvoker, registryUrl,
registeredProviderUrl);
    boolean register = providerUrl.getParameter(REGISTER_KEY, true);
    if (register) {
        register(registryUrl, registeredProviderUrl);
        providerInvokerWrapper.setReg(true);
    }

    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);
}

public void register(URL registryUrl, URL registeredProviderUrl) {
    Registry registry = registryFactory.getRegistry(registryUrl);
    registry.register(registeredProviderUrl);
}

public void unregister(URL registryUrl, URL registeredProviderUrl) {
    Registry registry = registryFactory.getRegistry(registryUrl);
    registry.unregister(registeredProviderUrl);
}
```

reExport(Invoker<T>,URL) 重新导出主流程

如下述源码所示，重新导出的流程实际实际上很简单，首先执行本地的重导入处理，然后只是简单的将当前服务实例已应用过同步事件覆写规则的 registeredProviderUrl 设给最初服务实例在本地导出时就生成了的 ExporterChangeableWrapper 类型对象 exporter 。

```java
public <T> void reExport(final Invoker<T> originInvoker, URL newInvokerUrl) {
    // update local exporter
    ExporterChangeableWrapper exporter = doChangeLocalExport(originInvoker,
newInvokerUrl);
    // update registry
    URL registryUrl = getRegistryUrl(originInvoker);
    final URL registeredProviderUrl = getRegisteredProviderUrl(newInvokerUrl,
registryUrl);

    ...//TAG:x

    exporter.setRegisterUrl(registeredProviderUrl);
}
```

然而问题来了，之所以重新导出的原因是运维人员在配置中心改写了相关配置项，从而导致当前微服务实例的覆写规则同步事件收到了通知，这又进一步引起了 `originInvoker` 相关的URL数据的变化。我们都清楚一个微服务的实例是作为临时页节点存储在注册中心的，节点是该实例的完整URL数据表示，此时本地版本已经发生了变化，而注册中心还维持着原样，这肯定会导致不一致。

其实上述源码中 `TAG:x` 处故意给删除了如下一段代码，基本意思是在本地会有一个 `ProviderConsumerRegTable` 缓存容器，类似于注册表，就 `originInvoker` 而言，如果注册表中已经记录的 `registeredProviderUrl` 和当前刷新后的不一致，便先使用旧的值从注册中心执行解注册处理，然后用心的值做登记。

```java
public <T> void reExport(final Invoker<T> originInvoker, URL newInvokerUrl) {
    ...
    //decide if we need to re-publish
    ProviderInvokerWrapper<T> providerInvokerWrapper =
        ProviderConsumerRegTable.getProviderWrapper(registeredProviderUrl,
originInvoker);
    ProviderInvokerWrapper<T> newProviderInvokerWrapper =
        ProviderConsumerRegTable.registerProvider(originInvoker, registryUrl,
registeredProviderUrl);

    if (providerInvokerWrapper.isReg() && !registeredProviderUrl.equals(
            providerInvokerWrapper.getProviderUrl())) {
        unregister(registryUrl, providerInvokerWrapper.getProviderUrl());
        register(registryUrl, registeredProviderUrl);
        newProviderInvokerWrapper.setReg(true);
    }
    ...
}
```

`ProviderInvokerWrapper` 和 `ProviderConsumerRegTable` 相关实现后面有机会再聊。

RegistryProtocol#destroy()

`RegistryProtocol` 的销毁处理显得相当干净利落，先是从 `bounds` 取出所有所有的 `Exporter` 执行其 `unexport()` ，然后删除到配置中心的应用级别的覆写规则同步监听器。

JAVA

```java
public void destroy() {
    List<Exporter<?>> exporters = new ArrayList<Exporter<?>>(bounds.values());
    for (Exporter<?> exporter : exporters) {
        exporter.unexport();
    }
    bounds.clear();

    DynamicConfiguration.getDynamicConfiguration().removeListener(
        ApplicationModel.getApplication() + CONFIGURATORS_SUFFIX,
providerConfigurationListener);
}
```

完结