Dubbo源码解析 — 服务订阅以及通知

前言

由于服务的注册内容比较多，所以把上一部分单独做一篇，订阅和通知合在一起。

subscribe

承接上一篇，我们说完了服务的发布，现在看看服务的订阅。考虑了一下，我还是决定在这里贴一下Protocol接口的相关注释，方便大家理解：

```
/**
     * 暴露远程服务：<br>
     * 1. 协议在接收请求时，应记录请求来源方地址信息：RpcContext.getContext().se
tRemoteAddress();<br>
     * 2. export()必须是幂等的，也就是暴露同一个URL的Invoker两次，和暴露一次没有
区别。<br>
     * 3. export()传入的Invoker由框架实现并传入，协议不需要关心。<br>
     *
     * @param <T> 服务的类型
     * @param invoker 服务的执行体
     * @return exporter 暴露服务的引用，用于取消暴露
     * @throws RpcException 当暴露服务出错时抛出，比如端口已占用
     */
    @Adaptive
    <T> Exporter<T> export(Invoker<T> invoker) throws RpcException;

    /**
     * 引用远程服务：<br>
     * 1. 当用户调用refer()所返回的Invoker对象的invoke()方法时，协议需相应执行同
URL远端export()传入的Invoker对象的invoke()方法。<br>
     * 2. refer()返回的Invoker由协议实现，协议通常需要在此Invoker中发送远程请
求。<br>
     * 3. 当url中有设置check=false时，连接失败不能抛出异常，并内部自动恢复。<br>
     *
     * @param <T> 服务的类型
     * @param type 服务的类型
     * @param url 远程服务的URL地址
     * @return invoker 服务的本地代理
     * @throws RpcException 当连接服务提供方失败时抛出
     */
    @Adaptive
    <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException;
```

这里很容易就可以明白，export发布，refer引用，然后获取invoker，然后执行invoker的方法即可。

继续看一下RegistryProtocol的refer方法：

```
public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException
 {

    //处理注册中心的协议，用url中registry参数的值作为真实的注册中心协议
    url = url.setProtocol(url.getParameter(Constants.REGISTRY_KEY, Const
ants.DEFAULT_REGISTRY)).removeParameter(Constants.REGISTRY_KEY);

    //拿到真正的注册中心实例，我们的例子中就是zookeeperRegistry
    Registry registry = registryFactory.getRegistry(url);

    if (RegistryService.class.equals(type))
        //todo 不太理解，貌似是注册中心服务本身的暴露
        return proxyFactory.getInvoker((T) registry, type, url);
    }

    //分组聚合处理，http://alibaba.github.io/dubbo-doc-static/Merge+By+Grou
p-zh.htm
    // group="a,b" or group="*"
    Map<String, String> qs = StringUtils.parseQueryString(url.getParamet
erAndDecoded(Constants.REFER_KEY));

    String group = qs.get(Constants.GROUP_KEY);

    if (group != null && group.length() > 0 ) {
        if ( ( Constants.COMMA_SPLIT_PATTERN.split( group ) ).length > 1
                || "*".equals( group ) ) {
            return doRefer( getMergeableCluster(), registry, type, url
 );
        }
    }

    return doRefer(cluster, registry, type, url);
}
```

继续看一下doRefer方法：

```
private <T> Invoker<T> doRefer(Cluster cluster, Registry registry,
Class<T> type, URL url) {

    //这个directory把同一个serviceInterface对应的多个invoker管理起来提供概念上
的化多为单一，供路由、均衡算法等使用
    RegistryDirectory<T> directory = new RegistryDirectory<T>(type, ur
l);
    directory.setRegistry(registry);
    directory.setProtocol(protocol);
    URL subscribeUrl = new URL(Constants.CONSUMER_PROTOCOL, NetUtils.get
LocalHost(), 0, type.getName(), directory.getUrl().getParameters());

    //注册自己
    if (! Constants.ANY_VALUE.equals(url.getServiceInterface())
            && url.getParameter(Constants.REGISTER_KEY, true)) {
        registry.register(subscribeUrl.addParameters(Constants.CATEGORY_
KEY, Constants.CONSUMERS_CATEGORY,
                Constants.CHECK_KEY, String.valueOf(false)));
    }

    //订阅目标服务提供方

directory.subscribe(subscribeUrl.addParameter(Constants.CATEGORY_KEY, Co
nstants.PROVIDERS_CATEGORY
    + "," + Constants.CONFIGURATORS_CATEGORY
    + "," + Constants.ROUTERS_CATEGORY));

    //合并所有相同invoker
    return cluster.join(directory);
}
```

服务消费方不仅会订阅相关的服务，也会注册自身供其他层使用（服务治理）。特别要注意的是订阅时，同时订阅了三个分类类型：***providers，routers，configurators***。

继续深挖dubbo中服务消费方订阅服务的细节，上面方法中最终把订阅细节委托给RegistryDirectory.subscribe方法，注意，这个方法接受的参数，此时的url已经把category设置为providers，routers，configurators：

```
public void subscribe(URL url) {
    setConsumerUrl(url);
    registry.subscribe(url, this);
}
```

这里registry就是zookeeperRegistry，这在doRefer方法可以看到明确的注入。然后和注册服务时一样，订阅会先由FailbackRegistry完成失效重试的处理，最终会交给zookeeperRegistry.doSubscribe方法。zookeeperRegistry实例拥有ZookeeperClient类型引用，该类型对象封装了和zookeeper通信的逻辑（默认是使用zkclient客户端）。我们在doSubscribe中能看到对zkClient的使用，而在ZK中的路径模式，统一看上一篇关于服务注册的最后一张图。

订阅之后，下面我们讲解notify，我们在doSubscribe中可以看到NotifyListener的调用。

Notify

这里先解决一个小问题，之前在其他博客上也看到过，dubbo中有一个ChildListener接口，并且我们在ZookeeperRegistry中会看到如下：

```
private final ConcurrentMap<URL, ConcurrentMap<NotifyListener, ChildList
ener>> zkListeners = new ConcurrentHashMap<URL, ConcurrentMap<NotifyList
ener, ChildListener>>();
```

这里我还没用细究这个ChildListener的作用，借鉴了一篇博客里的讲解：

> 这个ChildListener接口用于**把zkclient的事件（IZkChildListener）转换到registry事件（NotifyListener）**。这么做的深意不是特别的理解，可能是因为我并没有太多zookeeper的使用经验导致的，这里的做法可以*更好的把zkclient的api和dubbo真身的注册中心逻辑分离开*，毕竟dubbo除了zkclient以外还可以选择curator。

我们主要看一下RegistryDirectory的notify方法：

```
public synchronized void notify(List<URL> urls) {

    ...

    // configurators 更新缓存的服务提供方配置规则
    if (configuratorUrls != null && configuratorUrls.size() >0 ) {
        this.configurators = toConfigurators(configuratorUrls);
    }

    // routers  更新缓存的路由配置规则
    if (routerUrls != null && routerUrls.size() >0 ){
        List<Router> routers = toRouters(routerUrls);
        if(routers != null){ // null - do nothing
            setRouters(routers);
        }
    }

    // 合并override参数
    List<Configurator> localConfigurators = this.configurators;

    this.overrideDirectoryUrl = directoryUrl;
    if (localConfigurators != null && localConfigurators.size() > 0) {
        for (Configurator configurator : localConfigurators) {
            this.overrideDirectoryUrl = configurator.configure(overrideD
irectoryUrl);
        }
    }

    // providers
    refreshInvoker(invokerUrls);
}
```

这里在每次消费方接受到注册中心的通知后，大概会做下面这些事儿：

- 更新服务提供方的配置规则
- 更新路由规则
- 重建invoker实例

我们这里主要看dubbo如何“重建invoker实例”，也就是最后一行代码调用的方法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())) {

        //如果传入的参数只包含一个empty://协议的url，表明禁用当前服务
        this.forbidden = true; // 禁止访问
        this.methodInvokerMap = null; // 置空列表
        destroyAllInvokers(); // 关闭所有Invoker
    } else {
        this.forbidden = false; // 允许访问
        Map<String, Invoker<T>> oldUrlInvokerMap = this.urlInvokerMap;

        if (invokerUrls.size() == 0 && this.cachedInvokerUrls != null) {
             //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
            invokerUrls.addAll(this.cachedInvokerUrls);
        } else {
            this.cachedInvokerUrls = new HashSet<URL>();
            //缓存invokerUrls列表，便于交叉对比
            this.cachedInvokerUrls.addAll(invokerUrls);
        }

        if (invokerUrls.size() ==0 ){
            return;
        }

        Map<String, Invoker<T>> newUrlInvokerMap = toInvokers(invokerUrls) ;// 将URL列表转成Invoker列表
        Map<String, List<Invoker<T>>> newMethodInvokerMap = toMethodInvokers(newUrlInvokerMap); // 换方法名映射Invoker列表

        // state change
        //如果计算错误，则不进行处理.
        if (newUrlInvokerMap == null || newUrlInvokerMap.size() == 0 ) {

            logger.error(new IllegalStateException("urls to invokers error .invokerUrls.size :"+invokerUrls.size() + ", invoker.size :0. urls :"+invokerUrls.toString()));
            return ;
        }

        this.methodInvokerMap = multiGroup ? toMergeMethodInvokerMap(newMethodInvokerMap) : newMethodInvokerMap;
        this.urlInvokerMap = newUrlInvokerMap;

        try {
            // 关闭未使用的Invoker
            destroyUnusedInvokers(oldUrlInvokerMap,newUrlInvokerMap);
        }catch (Exception e) {
            logger.warn("destroyUnusedInvokers error. ", e);
```

```
                }
            }
        }
```

到这里我们已经完成了服务通知的业务逻辑，有兴趣的童鞋可以深究一下toInvokers方法，它又会走一遍url -> invoker的逻辑（服务引用）。