芋道源码 —— 知识星球

我是一段不羁的公告！
记得给芳芳这 3 个项目加油，添加一个 STAR 噢。
https://github.com/YunaiV/SpringBoot-Labs
https://github.com/YunaiV/onemall
https://github.com/YunaiV/ruoyi-vue-pro
2018-08-04
Dubbo

精尽 Dubbo 源码分析 —— 注册中心（二）之 Zookeeper

本文基于 Dubbo 2.6.1 版本，望知悉。

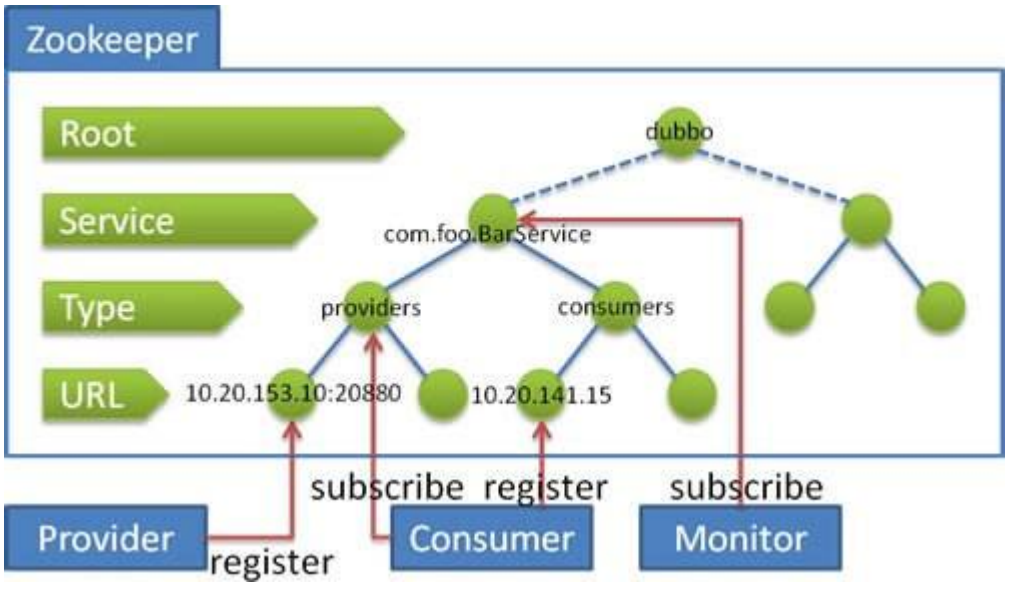
1. 概述

前置阅读文章：

《精尽 Dubbo 源码分析 —— Zookeeper 客户端》
《精尽 Dubbo 源码分析 —— 注册中心（一）之抽象 API》

在《注册中心（一）之抽象 API》 中，我们分享的那是相当抽象。因此，在本文中，我们会分享 Dubbo 使用 Zookeeper 作为注册中心的代码，同时也会分享服务暴露和引用时，对注册中心的使用。

下面，我们先来看下 《Dubbo 用户指南 —— zookeeper 注册中心》 文档，内容如下：



流程说明：

服务提供者启动时：向 /dubbo/com.foo.BarService/providers 目录下写入自己的 URL 地址

服务消费者启动时：订阅 /dubbo/com.foo.BarService/providers 目录下的提供者 URL 地址。并向 /dubbo/com.foo.BarService/consumers 目录下写入自己的 URL 地址

监控中心启动时：订阅 /dubbo/com.foo.BarService 目录下的所有提供者和消费者 URL 地址。

在图中，我们可以看到 Zookeeper 的节点层级，自上而下是：

- Root 层：根目录，可通过 <dubbo:registry group="dubbo" /> 的 "group" 设置 Zookeeper 的根节点，缺省使用 "dubbo" 。
- Service 层：服务接口全名。
- Type 层：分类。目前除了我们在图中看到的 "providers"(服务提供者列表) "consumers"(服务消费者列表) 外，还有 "routes"(路由规则列表) 和 "configurations"(配置规则列表)。
- URL 层：URL ，根据不同 Type 目录，下面可以是服务提供者 URL 、服务消费者 URL 、路由规则 URL 、配置规则 URL 。
- 实际上 URL 上带有 "category" 参数，已经能判断每个 URL 的分类，但是 Zookeeper 是基于节点目录订阅的，所以增加了 Type 层。

实际上，服务消费者启动后，不仅仅订阅了 "providers" 分类，也订阅了 "routes" "configurations" 分类。

2. ZookeeperRegistryFactory

com.alibaba.dubbo.registry.zookeeper.ZookeeperRegistryFactory ，实现 AbstractRegistryFactory 抽象类，Zookeeper Registry 工厂。代码如下：

```
public class ZookeeperRegistryFactory extends AbstractRegistryFactory {

    /**
     * Zookeeper 工厂
     */
    private ZookeeperTransporter zookeeperTransporter;

    /**
     * 设置 Zookeeper 工厂
     *
     * 该方法，通过 Dubbo SPI 注入
     *
     * @param zookeeperTransporter Zookeeper 工厂对象
     */
    public void setZookeeperTransporter(ZookeeperTransporter zookeeperTransporter) {
        this.zookeeperTransporter = zookeeperTransporter;
    }

    @Override
    public Registry createRegistry(URL url) {
        return new ZookeeperRegistry(url, zookeeperTransporter);
    }

}
```

3. ZookeeperRegistry

com.alibaba.dubbo.registry.zookeeper.ZookeeperRegistry ，实现 FailbackRegistry 抽象类，ZookeeperRegistry 。

3.1 属性 + 构造方法

```
1: /**
2:  * 默认端口
3:  */
4: private final static int DEFAULT_ZOOKEEPER_PORT = 2181;
5: /**
6:  * 默认 Zookeeper 根节点
7:  */
8: private final static String DEFAULT_ROOT = "dubbo";
9:
10: /**
11:  * Zookeeper 根节点
12:  */
13: private final String root;
14: /**
15:  * Service 接口全名集合
16:  */
17: private final Set<String> anyServices = new ConcurrentHashSet<String>();
18: /**
19:  * 监听器集合
20:  */
21: private final ConcurrentMap<URL, ConcurrentMap<NotifyListener, ChildListener>> zkListeners = new ConcurrentHashMa
22: /**
23:  * Zookeeper 客户端
24:  */
25: private final ZookeeperClient zkClient;
26:
27: public ZookeeperRegistry(URL url, ZookeeperTransporter zookeeperTransporter) {
28:     super(url);
29:     if (url.isAnyHost()) {
30:         throw new IllegalStateException("registry address == null");
31:     }
32:     // 获得 Zookeeper 根节点
33:     String group = url.getParameter(Constants.GROUP_KEY, DEFAULT_ROOT); // `url.parameters.group` 参数值
34:     if (!group.startsWith(Constants.PATH_SEPARATOR)) {
35:         group = Constants.PATH_SEPARATOR + group;
36:     }
37:     this.root = group;
38:     // 创建 Zookeeper Client
39:     zkClient = zookeeperTransporter.connect(url);
40:     // 添加 StateListener 对象。该监听器，在重连时，调用恢复方法。
41:     zkClient.addStateListener(new StateListener() {
42:         public void stateChanged(int state) {
43:             if (state == RECONNECTED) {
44:                 try {
45:                     recover();
46:                 } catch (Exception e) {
47:                     logger.error(e.getMessage(), e);
48:                 }
49:             }
50:         }
```

```
51:     });
52: }
```

root 属性，Zookeeper 根节点，即首图的 Root 层。
anyServices 属性，Service 接口接口全名集合。该属性适可用于监控中心，订阅整个 Service 层。因为，Service 层是动态的，可以有不断有新的 Service 服务发布（注意，不是服务实例）。在 #doSubscribe(url, notifyListener) 方法中，会更容易理解。
zkListeners 属性，监听器集合，建立 NotifyListener 和 ChildListener 的映射关系。
zkClient 属性，Zookeeper 客户端。
构造方法

- 第 28 至 31 行：设置注册中心的 URL 。
- 第 32 至 37 行：设置在 Zookeeper 的根节点，缺省使用 DEFAULT_ROOT 。
- 第 39 行：调用 ZookeeperTransporter#connect(url) 方法，基于 Dubbo SPI Adaptive 机制，根据 url 参数，加载对应的 ZookeeperTransporter 实现类，创建对应的 ZookeeperClient 实现类的对应。
- 第 41 至 51 行：添加 StateListener 对象到 ZookeeperClient 对象中。该监听器，在重连时，在第 45 行的代码，调用 #recover() 方法，进行恢复逻辑，重新发起注册和订阅。

3.2 doRegister

```
1: @Override
2: protected void doRegister(URL url) {
3:     try {
4:         zkClient.create(toUrlPath(url), url.getParameter(Constants.DYNAMIC_KEY, true));
5:     } catch (Throwable e) {
6:         throw new RpcException("Failed to register " + url + " to zookeeper " + getUrl() + ", cause: " + e.getMess
7:     }
8: }
```

第 4 行：调用 #toUrlPath(url) 方法，获得 URL 的路径。
第 4 行：url.parameters.dynamic ，是否动态数据。若为 false ，该数据为持久数据，当注册方退出时，数据依然保存在注册中心。
第 4 行：调用 ZookeeperClient#create(url, ephemeral) 方法，创建 URL 节点，即我们在首图看到的 URL 层。

3.2.1 toUrlPath

```
/**
 * 获得 URL 的路径
 *
 * Root + Service + Type + URL
 *
 * 被 {@link #doRegister(URL)} 和 {@link #doUnregister(URL)} 调用
 *
 * @param url URL
 * @return 路径
 */
private String toUrlPath(URL url) {
    return toCategoryPath(url) + Constants.PATH_SEPARATOR + URL.encode(url.toFullString());
}
```

3.2.2 toCategoryPath

```
/**
 * 获得分类路径
 *
 * Root + Service + Type
 *
 * @param url URL
 * @return 分类路径
 */
private String toCategoryPath(URL url) {
    return toServicePath(url) + Constants.PATH_SEPARATOR + url.getParameter(Constants.CATEGORY_KEY, Constants.DEFAULT
}
```

3.2.3 toServicePath

```
/**
 * 获得服务路径
 *
 * Root + Type
 *
 * @param url URL
 * @return 服务路径
 */
private String toServicePath(URL url) {
    String name = url.getServiceInterface();
    if (Constants.ANY_VALUE.equals(name)) {
        return toRootPath();
    }
    return toRootDir() + URL.encode(name);
}
```

3.2.4 toRootDir

```
/**
 * 获得根目录
 *
 * Root
 *
 * @return 路径
 */
private String toRootDir() {
    if (root.equals(Constants.PATH_SEPARATOR)) {
        return root;
    }
    return root + Constants.PATH_SEPARATOR;
}

/**
 * Root
 *
 * @return 根路径
 */
private String toRootPath() {
```

```
    return root;
}
```

3.3 doUnregister

```
@Override
protected void doUnregister(URL url) {
    try {
        zkClient.delete(toUrlPath(url));
    } catch (Throwable e) {
        throw new RpcException("Failed to unregister " + url + " to zookeeper " + getUrl() + ", cause: " + e.getMessa
    }
}
```

3.4 doSubscribe

```
 1: @Override
 2: protected void doSubscribe(final URL url, final NotifyListener listener) {
 3:     try {
 4:         // 处理所有 Service 层的发起订阅，例如监控中心的订阅
 5:         if (Constants.ANY_VALUE.equals(url.getServiceInterface())) {
 6:             String root = toRootPath();
 7:             // 获得 url 对应的监听器集合
 8:             ConcurrentMap<NotifyListener, ChildListener> listeners = zkListeners.get(url);
 9:             if (listeners == null) { // 不存在，进行创建
10:                 zkListeners.putIfAbsent(url, new ConcurrentHashMap<NotifyListener, ChildListener>());
11:                 listeners = zkListeners.get(url);
12:             }
13:             // 获得 ChildListener 对象
14:             ChildListener zkListener = listeners.get(listener);
15:             if (zkListener == null) { // 不存在 ChildListener 对象，进行创建 ChildListener 对象
16:                 listeners.putIfAbsent(listener, new ChildListener() {
17:                     public void childChanged(String parentPath, List<String> currentChilds) {
18:                         for (String child : currentChilds) {
19:                             child = URL.decode(child);
20:                             // 新增 Service 接口全名时（即新增服务），发起该 Service 层的订阅
21:                             if (!anyServices.contains(child)) {
22:                                 anyServices.add(child);
23:                                 subscribe(url.setPath(child).addParameters(Constants.INTERFACE_KEY, child,
24:                                         Constants.CHECK_KEY, String.valueOf(false)), listener);
25:                             }
26:                         }
27:                     }
28:                 });
29:                 zkListener = listeners.get(listener);
30:             }
31:             // 创建 Service 节点。该节点为持久节点。
32:             zkClient.create(root, false);
33:             // 向 Zookeeper ，Service 节点，发起订阅
34:             List<String> services = zkClient.addChildListener(root, zkListener);
35:             // 首次全量数据获取完成时，循环 Service 接口全名数组，发起该 Service 层的订阅
36:             if (services != null && !services.isEmpty()) {
37:                 for (String service : services) {
38:                     service = URL.decode(service);
```

```
39:                     anyServices.add(service);
40:                     subscribe(url.setPath(service).addParameters(Constants.INTERFACE_KEY, service,
41:                             Constants.CHECK_KEY, String.valueOf(false)), listener);
42:                 }
43:             }
44:         // 处理指定 Service 层的发起订阅，例如服务消费者的订阅
45:         } else {
46:             // 子节点数据数组
47:             List<URL> urls = new ArrayList<URL>();
48:             // 循环分类数组
49:             for (String path : toCategoriesPath(url)) {
50:                 // 获得 url 对应的监听器集合
51:                 ConcurrentMap<NotifyListener, ChildListener> listeners = zkListeners.get(url);
52:                 if (listeners == null) { // 不存在，进行创建
53:                     zkListeners.putIfAbsent(url, new ConcurrentHashMap<NotifyListener, ChildListener>());
54:                     listeners = zkListeners.get(url);
55:                 }
56:                 // 获得 ChildListener 对象
57:                 ChildListener zkListener = listeners.get(listener);
58:                 if (zkListener == null) { // 不存在 ChildListener 对象，进行创建 ChildListener 对象
59:                     listeners.putIfAbsent(listener, new ChildListener() {
60:                         public void childChanged(String parentPath, List<String> currentChilds) {
61:                             // 变更时，调用 `#notify(...)` 方法，回调 NotifyListener
62:                             ZookeeperRegistry.this.notify(url, listener, toUrlsWithEmpty(url, parentPath, current
63:                         }
64:                     });
65:                     zkListener = listeners.get(listener);
66:                 }
67:                 // 创建 Type 节点。该节点为持久节点。
68:                 zkClient.create(path, false);
69:                 // 向 Zookeeper ，PATH 节点，发起订阅
70:                 List<String> children = zkClient.addChildListener(path, zkListener);
71:                 // 添加到 `urls` 中
72:                 if (children != null) {
73:                     urls.addAll(toUrlsWithEmpty(url, path, children));
74:                 }
75:             }
76:             // 首次全量数据获取完成时，调用 `#notify(...)` 方法，回调 NotifyListener
77:             notify(url, listener, urls);
78:         }
79:     } catch (Throwable e) {
80:         throw new RpcException("Failed to subscribe " + url + " to zookeeper " + getUrl() + ", cause: " + e.getMe
81:     }
82: }
```

整个方法分成两部分，分别：

========== 第二部分【第 44 至 78 行】 ==========

处理指定 Service 层的发起订阅，例如服务消费者的订阅。

第 47 行：子节点数据数组，即 Service 层下的所有 URL 。

第 49 行：循环分类数组。其中，调用 `#toCategoriesPath(url)` 方法，获得 分类数组。

第 51 至 55 行：获得订阅的 `url` 对应的监听器集合。

第 56 至 66 行：获得 `listener`(NotifyListener) 对应的 ChildListener 对象。在 URL 层发生变更时，会调用 `NotifyListener#notify(url, listener, currentChilds)` 方法，回调 NotifyListener 的逻辑。酱紫，如果 Service 下增加新的服务提供者实例(新的 URL)，服务消费者可创建新的 Invoker 对象，用于调用该服务提供者。

第 68 行：创建 Type 节点。该节点为持久节点。

第 70 行：向 Zookeeper 的 Path 节点，发起订阅。

第 72 至 74 行：添加到 `urls` 中。

第 77 行：首次全量数据获取完成时，调用 NotifyListener#notify(url, listener, currentChilds) 方法，回调 NotifyListener 的逻辑。酱紫，服务消费者可创建所有的 Invoker 对象，用于调用服务提供者们。

回看【第 77 行】和【第 62 行】，全量 + 增量，仔细理解下。

========== 第一部分【第 5 至 43 行】 ==========

处理所有 Service 层的发起订阅，例如监控中心的订阅

第 8 至 12 行：获得订阅的 url 对应的监听器集合。

第 13 至 30 行：获得 listener(NotifyListener) 对应的 ChildListener 对象。在 Service 层发生变更时，若是新增 Service 接口全名时（即新增服务），调用 #subscribe(url, listener) 方法，发起该 Service 层的订阅（【第 45 至 78 行】的逻辑）。是否是新增的服务，通过 anyServices 属性来判断。

第 32 行：创建 Service 节点。该节点为持久节点。

第 34 行：向 Zookeeper 的 Service 节点，发起订阅。

第 36 至 43 行：首次全量数据获取完成时，循环 Service 接口全名数组，调用 #subscribe(url, listener) 方法，发起该 Service 层的订阅（【第 45 至 78 行】的逻辑）。

友情提示：如果觉得比较绕，或者笔者讲的不清晰，胖友可以进行调试理解。

3.4.1 toCategoriesPath

```
/**
 * 获得分类路径数组
 *
 * Root + Service + Type
 *
 * @param url URL
 * @return 分类路径数组
 */
private String[] toCategoriesPath(URL url) {
    // 获得分类数组
    String[] categories;
    if (Constants.ANY_VALUE.equals(url.getParameter(Constants.CATEGORY_KEY))) { // * 时，
        categories = new String[]{Constants.PROVIDERS_CATEGORY, Constants.CONSUMERS_CATEGORY,
                Constants.ROUTERS_CATEGORY, Constants.CONFIGURATORS_CATEGORY};
    } else {
        categories = url.getParameter(Constants.CATEGORY_KEY, new String[]{Constants.DEFAULT_CATEGORY});
    }
    // 获得分类路径数组
    String[] paths = new String[categories.length];
    for (int i = 0; i < categories.length; i++) {
        paths[i] = toServicePath(url) + Constants.PATH_SEPARATOR + categories[i];
    }
    return paths;
}
```

3.4.2 toUrlsWithEmpty

```
/**
 * 获得 providers 中，和 consumer 匹配的 URL 数组
 *
 * 若不存在匹配，则创建 `empty://` 的 URL返回。通过这样的方式，可以处理类似服务提供者为空的情况。
 *
 * @param consumer 用于匹配 URL
 * @param path 被匹配的 URL 的字符串
```

```
     * @param providers 匹配的 URL 数组
     * @return 匹配的 URL 数组
     */
    private List<URL> toUrlsWithEmpty(URL consumer, String path, List<String> providers) {
        // 获得 providers 中，和 consumer 匹配的 URL 数组
        List<URL> urls = toUrlsWithoutEmpty(consumer, providers);
        // 若不存在匹配，则创建 `empty://` 的 URL返回
        if (urls == null || urls.isEmpty()) {
            int i = path.lastIndexOf('/');
            String category = i < 0 ? path : path.substring(i + 1);
            URL empty = consumer.setProtocol(Constants.EMPTY_PROTOCOL).addParameter(Constants.CATEGORY_KEY, category);
            urls.add(empty);
        }
        return urls;
    }
```

#toUrlsWithoutEmpty() 方法，代码如下：

```
    /**
     * 获得 providers 中，和 consumer 匹配的 URL 数组
     *
     * @param consumer 用于匹配 URL
     * @param providers 被匹配的 URL 的字符串
     * @return 匹配的 URL 数组
     */
    private List<URL> toUrlsWithoutEmpty(URL consumer, List<String> providers) {
        List<URL> urls = new ArrayList<URL>();
        if (providers != null && !providers.isEmpty()) {
            for (String provider : providers) {
                provider = URL.decode(provider);
                if (provider.contains("://")) { // 是 url
                    URL url = URL.valueOf(provider); // 将字符串转化成 URL
                    if (UrlUtils.isMatch(consumer, url)) { // 匹配
                        urls.add(url);
                    }
                }
            }
        }
        return urls;
    }
```

3.5 doUnsubscribe

```
@Override
protected void doUnsubscribe(URL url, NotifyListener listener) {
    ConcurrentMap<NotifyListener, ChildListener> listeners = zkListeners.get(url);
    if (listeners != null) {
        ChildListener zkListener = listeners.get(listener);
        if (zkListener != null) {
            // 向 Zookeeper ，移除订阅
            zkClient.removeChildListener(toUrlPath(url), zkListener);
        }
    }
}
```

3.6 lookup

```
/**
 * 查询符合条件的已注册数据，与订阅的推模式相对应，这里为拉模式，只返回一次结果。
 *
 * @param url 查询条件，不允许为空，如：consumer://10.20.153.10/com.alibaba.foo.BarService?version=1.0.0&application=
 * @return 已注册信息列表，可能为空，含义同{@link com.alibaba.dubbo.registry.NotifyListener#notify(List<URL>)}的参数。
 * @see com.alibaba.dubbo.registry.NotifyListener#notify(List)
 */
@Override
public List<URL> lookup(URL url) {
    if (url == null) {
        throw new IllegalArgumentException("lookup url == null");
    }
    try {
        // 循环分类数组，获得所有的 URL 数组
        List<String> providers = new ArrayList<String>();
        for (String path : toCategoriesPath(url)) {
            List<String> children = zkClient.getChildren(path);
            if (children != null) {
                providers.addAll(children);
            }
        }
        // 匹配
        return toUrlsWithoutEmpty(url, providers);
    } catch (Throwable e) {
        throw new RpcException("Failed to lookup " + url + " from zookeeper " + getUrl() + ", cause: " + e.getMessage
    }
}
```

3.7 isAvailable

```
@Override
public boolean isAvailable() {
    return zkClient.isConnected();
}
```

3.8 destroy

```
@Override
public void destroy() {
    super.destroy();
    try {
        zkClient.close();
    } catch (Exception e) {
        logger.warn("Failed to close zookeeper client " + getUrl() + ", cause: " + e.getMessage(), e);
    }
}
```

4. 调用

4.1 服务提供者

回头看 《精尽 Dubbo 源码分析 —— 服务暴露（二）之远程暴露（Dubbo）》 的 「3.2.2 export」 小节，我们可以看到：

第 14 行：调用 `#register(registryUrl, registedProviderUrl)` 方法，向注册中心注册服务提供者（自己）。代码如下：

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

4.2 服务消费者

回头看 《精尽 Dubbo 源码分析 —— 服务引用（二）之远程引用（Dubbo）》 的 「3.2.2 doRefer」 小节，我们可以看到：

第 20 至 25 行：调用 `RegistryService#register(url)` 方法，向注册中心注册自己（服务消费者）。
第 35 行：调用 `Directory#subscribe(url)` 方法，向注册中心订阅服务提供者 + 路由规则 + 配置规则。
- 在该方法中，会循环获得到的服务体用这列表，调用 `Protocol#refer(type, url)` 方法，创建每个调用服务的 Invoker 对象。

666. 彩蛋

嘿嘿，写完 Zookeeper 作为注册中心是否清晰了一些？！

文章目录

回到首页