crazyhzm 265

Dubbo源码解析（七）注册中心——zookeeper

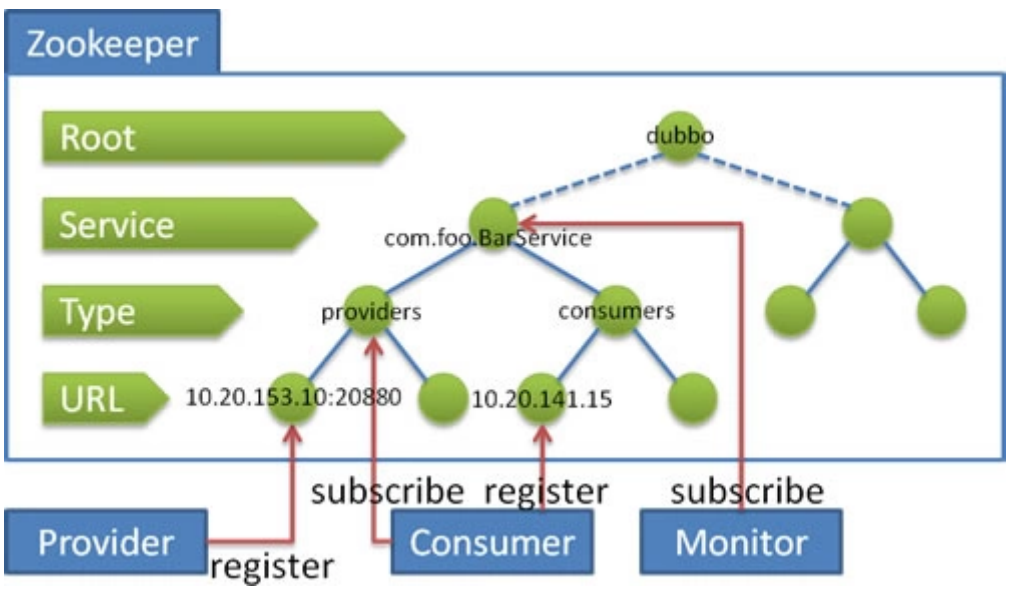
java dubbo zookeeper 阅读约 32 分钟

注册中心——zookeeper

> 目标：解释以为zookeeper实现的注册中心原理，解读duubo-registry-zookeeper的源码

这篇文章是讲解注册中心的最后一篇文章。这篇文章讲的是dubbo的注册中心用zookeeper来实现。这种实现注册中心的方法也是dubbo推荐的方法。为了能更加理解zookeeper在dubbo中的应用，接下来我先简单的介绍一下zookeeper。

因为dubbo是一个分布式的RPC开源框架，各个服务之间单独部署，就会出现资源之间不一致的问题。而zookeeper就有保证分布式一致性的特性。ZooKeeper是一种为分布式应用所设计的高可用、高性能且一致的开源协调服务。关于dubbo为什么会推荐使用zookeeper作为它的注册中心实现，有很多书籍以及博客讲解了zookeeper的特性以及优势，这不是本章的重点，我要讲的是zookeeper的数据结构，dubbo服务是如何被zookeeper的数据结构存储管理的，因为这影响到下面源码的解读。zookeeper采用的是树形结构来组织数据节点，它类似于一个标准的文件系统。先来看看下面这张图：



该图是官方文档里面的一张图，展示了dubbo在zookeeper中存储的形式以及节点层级，

1. dubbo的Root层是根目录，通过<dubbo:registry group="dubbo" />的“group”来设置zookeeper的根节点，缺省值是“dubbo”。
2. Service层是服务接口的全名。
3. Type层是分类，一共有四种分类，分别是providers（服务提供者列表）、consumers（服务消费者列表）、routes（路由规则列表）、configurations（配置规则列表）。
4. URL层：根据不同的Type目录：可以有服务提供者 URL 、服务消费者 URL 、路由规则 URL 、配置规则 URL 。不同的Type关注的URL不同。

zookeeper以每个斜杠来分割每一层的znode，比如第一层根节点dubbo就是“/dubbo”，而第二层的Service层就是/com.foo.Barservice，zookeeper的每个节点通过路径来表示以及访问，例如服务提供者启动时，向/dubbo/com.foo.Barservice/providers目录下写入自己的URL地址。关于流程调用说明，见官方文档：

> 文档地址：http://dubbo.apache.org/zh-cn...

了解了dubbo在zookeeper中的节点层级，就可以看相关的源码了，下图是包的结构：

跟前面三种实现方式一样的目录，也就两个类，看起来非常的舒服，接下来就来解析这两个类。

（一）ZookeeperRegistry

该类继承了FailbackRegistry类，该类就是针对注册中心核心的功能注册、订阅、取消注册、取消订阅，查询注册列表进行展开，基于zookeeper来实现。

1.属性

```
// 日志记录
private final static Logger logger = LoggerFactory.getLogger(ZookeeperRegistry.class);

// 默认的zookeeper端口
private final static int DEFAULT_ZOOKEEPER_PORT = 2181;

// 默认zookeeper根节点
private final static String DEFAULT_ROOT = "dubbo";

// zookeeper根节点
private final String root;

// 服务接口集合
private final Set<String> anyServices = new ConcurrentHashSet<String>();

// 监听器集合
private final ConcurrentMap<URL, ConcurrentMap<NotifyListener, ChildListener>> zkListeners = new
ConcurrentHashMap<URL, ConcurrentMap<NotifyListener, ChildListener>>();

// zookeeper客户端实例
private final ZookeeperClient zkClient;
```

其实你会发现zookeeper虽然是最被推荐的，反而它的实现逻辑相对简单，因为调用了zookeeper服务组件，很多的逻辑不需要在dubbo中自己去实现。上面的属性介绍也很简单，不需要多说，更多的是调用zookeeper客户端。

2.构造方法

```
public ZookeeperRegistry(URL url, ZookeeperTransporter zookeeperTransporter) {
    super(url);
    if (url.isAnyHost()) {
        throw new IllegalStateException("registry address == null");
    }
    // 获得url携带的分组配置，并且作为zookeeper的根节点
    String group = url.getParameter(Constants.GROUP_KEY, DEFAULT_ROOT);
    if (!group.startsWith(Constants.PATH_SEPARATOR)) {
        group = Constants.PATH_SEPARATOR + group;
    }
    this.root = group;
    // 创建zookeeper client
    zkClient = zookeeperTransporter.connect(url);
    // 添加状态监听器，当状态为重连的时候调用恢复方法
    zkClient.addStateListener(new StateListener() {
        @Override
        public void stateChanged(int state) {
            if (state == RECONNECTED) {
                try {
                    // 恢复
                    recover();
                } catch (Exception e) {
                    logger.error(e.getMessage(), e);
                }
            }
        }
```

这里有以下几个关注点：

1. 参数中ZookeeperTransporter是一个接口，并且在dubbo中有ZkclientZookeeperTransporter和CuratorZookeeperTransporter两个实现类，ZookeeperTransporter还是一个可扩展的接口，基于 Dubbo SPI Adaptive 机制，会根据url中携带的参数去选择用哪个实现类。
2. 上面我说明了dubbo在zookeeper节点层级有一层是root层，该层是通过group属性来设置的。

3. 给客户端添加一个监听器，当状态为重连的时候调用FailbackRegistry的恢复方法

3.appendDefaultPort

```
static String appendDefaultPort(String address) {
    if (address != null && address.length() > 0) {
        int i = address.indexOf(':');
        // 如果地址本身没有端口，则使用默认端口2181
        if (i < 0) {
            return address + ":" + DEFAULT_ZOOKEEPER_PORT;
        } else if (Integer.parseInt(address.substring(i + 1)) == 0) {
            return address.substring(0, i + 1) + DEFAULT_ZOOKEEPER_PORT;
        }
    }
    return address;
}
```

该方法是拼接使用默认的zookeeper端口，就是方地址本身没有端口的时候才使用默认端口。

4.isAvailable && destroy

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

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

这里两个方法分别是检测zookeeper是否连接以及销毁连接，很简单，都是调用了zookeeper客户端封装好的方法。

5.doRegister && doUnregister

```
@Override
protected void doRegister(URL url) {
    try {
        // 创建URL节点，也就是URL层的节点
        zkClient.create(toUrlPath(url), url.getParameter(Constants.DYNAMIC_KEY, true));
    } catch (Throwable e) {
        throw new RpcException("Failed to register " + url + " to zookeeper " + getUrl() + ", cause: " +
e.getMessage(), e);
    }
}

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

这两个方法分别是注册和取消注册，也很简单，调用都是客户端create和delete方法，一个是创建一个节点，另一个是删除节点，该操作都在URL层。

6.doSubscribe

```
@Override
protected void doSubscribe(final URL url, final NotifyListener listener) {
    try {
        // 处理所有Service层发起的订阅，例如监控中心的订阅
        if (Constants.ANY_VALUE.equals(url.getServiceInterface())) {
            // 获得根目录
            String root = toRootPath();
            // 获得url对应的监听器集合
            ConcurrentMap<NotifyListener, ChildListener> listeners = zkListeners.get(url);
            // 不存在就创建监听器集合
            if (listeners == null) {
                zkListeners.putIfAbsent(url, new ConcurrentHashMap<NotifyListener, ChildListener>());
                listeners = zkListeners.get(url);
            }
            // 获得节点监听器
            ChildListener zkListener = listeners.get(listener);
            // 如果该节点监听器为空，则创建
            if (zkListener == null) {
                listeners.putIfAbsent(listener, new ChildListener() {
                    @Override
                    public void childChanged(String parentPath, List<String> currentChilds) {
                        // 遍历现有的节点，如果现有的服务集合中没有该节点，则加入该节点，然后订阅该节点
                        for (String child : currentChilds) {
                            // 解码
                            child = URL.decode(child);
                            if (!anyServices.contains(child)) {
```

这个方法是订阅，逻辑实现比较多，可以分两段来看，这里的实现把所有Service层发起的订阅以及指定的Service层发起的订阅分开处理。所有Service层类似于监控中心发起的订阅。指定的Service层发起的订阅可以看作是服务消费者的订阅。订阅的大致逻辑类似，不过还是有几个区别：

1. 所有Service层发起的订阅中的ChildListener是在在 Service 层发生变更时，才会做出解码，用anyServices属性判断是否是新增的服务，最后调用父类的subscribe订阅。而指定的Service层发起的订阅是在URL层发生变更的时候，调用notify，回调回调NotifyListener的逻辑，做到通知服务变更。
2. 所有Service层发起的订阅中客户端创建的节点是Service节点，该节点为持久节点，而指定的Service层发起的订阅中创建的节点是Type节点，该节点也是持久节点。这里补充一下zookeeper的持久节点是节点创建后，就一直存在，直到有删除操作来主动清除这个节点，不会因为创建该节点的客户端会话失效而消失。而临时节点的生命周期和客户端会话绑定。也就是说，如果客户端会话失效，那么这个节点就会自动被清除掉。注意，这里提到的是会话失效，而非连接断开。另外，在临时节点下面不能创建子节点。
3. 指定的Service层发起的订阅中调用了两次notify，第一次是增量的通知，也就是只是通知这次增加的服务节点，而第二个是全量的通知。

7.doUnsubscribe

```
@Override
protected void doUnsubscribe(URL url, NotifyListener listener) {
    // 获得监听器集合
    ConcurrentMap<NotifyListener, ChildListener> listeners = zkListeners.get(url);
    if (listeners != null) {
        // 获得子节点的监听器
        ChildListener zkListener = listeners.get(listener);
        if (zkListener != null) {
            // 如果为全部的服务接口，例如监控中心
            if (Constants.ANY_VALUE.equals(url.getServiceInterface())) {
                // 获得根目录
                String root = toRootPath();
                // 移除监听器
                zkClient.removeChildListener(root, zkListener);
            } else {
                // 遍历分类数组进行移除监听器
                for (String path : toCategoriesPath(url)) {
                    zkClient.removeChildListener(path, zkListener);
                }
            }
        }
    }
}
```

该方法是取消订阅，也是分为两种情况，所有的Service发起的取消订阅还是指定的Service发起的取消订阅。可以看到所有的Service发起的取消订阅就直接移除了根目录下所有的监听器，而指定的Service发起的取消订阅是移除了该Service层下面的所有Type节点监听器。如果不太明白再回去看看前面的那个节点层级图。

8.lookup

```
@Override
public List<URL> lookup(URL url) {
    if (url == null) {
        throw new IllegalArgumentException("lookup url == null");
    }
    try {
        List<String> providers = new ArrayList<String>();
        // 遍历分组类别
        for (String path : toCategoriesPath(url)) {
            // 获得子节点
            List<String> children = zkClient.getChildren(path);
            if (children != null) {
                providers.addAll(children);
            }
        }
        // 获得 providers 中，和 consumer 匹配的 URL 数组
        return toUrlsWithoutEmpty(url, providers);
    } catch (Throwable e) {
        throw new RpcException("Failed to lookup " + url + " from zookeeper " + getUrl() + ", cause: " +
e.getMessage(), e);
    }
}
```

该方法就是查询符合条件的已经注册的服务。调用了toUrlsWithoutEmpty方法，在后面会讲到。

9.toServicePath

```
private String toServicePath(URL url) {
    String name = url.getServiceInterface();
    // 如果是包括所有服务，则返回根节点
    if (Constants.ANY_VALUE.equals(name)) {
        return toRootPath();
    }
    return toRootDir() + URL.encode(name);
}
```

该方法是获得服务路径，拼接规则：Root + Type。

10.toCategoriesPath

```
private String[] toCategoriesPath(URL url) {
    String[] categories;
    // 如果url携带的分类配置为*，则创建包括所有分类的数组
    if (Constants.ANY_VALUE.equals(url.getParameter(Constants.CATEGORY_KEY))) {
        categories = new String[]{Constants.PROVIDERS_CATEGORY, Constants.CONSUMERS_CATEGORY,
                Constants.ROUTERS_CATEGORY, Constants.CONFIGURATORS_CATEGORY};
    } else {
        // 返回url携带的分类配置
        categories = url.getParameter(Constants.CATEGORY_KEY, new String[]{Constants.DEFAULT_CATEGORY});
    }
    String[] paths = new String[categories.length];
    for (int i = 0; i < categories.length; i++) {
        // 加上服务路径
        paths[i] = toServicePath(url) + Constants.PATH_SEPARATOR + categories[i];
    }
    return paths;
}

private String toCategoryPath(URL url) {
    return toServicePath(url) + Constants.PATH_SEPARATOR + url.getParameter(Constants.CATEGORY_KEY,
Constants.DEFAULT_CATEGORY);
}
```

第一个方法是获得分类数组，也就是url携带的服务下的所有Type节点数组。第二个是获得分类路径，分类路径拼接规则：Root + Service + Type

11.toUrlPath

```
private String toUrlPath(URL url) {
    return toCategoryPath(url) + Constants.PATH_SEPARATOR + URL.encode(url.toFullString());
}
```

该方法是获得URL路径，拼接规则是Root + Service + Type + URL

12.toUrlsWithoutEmpty && toUrlsWithEmpty

```
private List<URL> toUrlsWithoutEmpty(URL consumer, List<String> providers) {
    List<URL> urls = new ArrayList<URL>();
    if (providers != null && !providers.isEmpty()) {
        // 遍历服务提供者
        for (String provider : providers) {
            // 解码
            provider = URL.decode(provider);
            if (provider.contains("://")) {
                // 把服务转化成url的形式
                URL url = URL.valueOf(provider);
                // 判断是否匹配，如果匹配， 则加入到集合中
                if (UrlUtils.isMatch(consumer, url)) {
                    urls.add(url);
                }
            }
        }
    }
    return urls;
}

private List<URL> toUrlsWithEmpty(URL consumer, String path, List<String> providers) {
    // 返回和服务消费者匹配的服务提供者url
    List<URL> urls = toUrlsWithoutEmpty(consumer, providers);
    // 如果不存在，则创建`empty://` 的 URL返回
    if (urls == null || urls.isEmpty()) {
        int i = path.lastIndexOf('/');
```

第一个toUrlsWithoutEmpty方法是获得 providers 中，和 consumer 匹配的 URL 数组，第二个toUrlsWithEmpty方法是调用了第一个方法后增加了若不存在匹配，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。

（二）ZookeeperRegistryFactory

该类继承了AbstractRegistryFactory类，实现了AbstractRegistryFactory抽象出来的createRegistry方法，看一下原代码：

```
public class ZookeeperRegistryFactory extends AbstractRegistryFactory {

    private ZookeeperTransporter zookeeperTransporter;

    public void setZookeeperTransporter(ZookeeperTransporter zookeeperTransporter) {
        this.zookeeperTransporter = zookeeperTransporter;
    }

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

}
```

可以看到就是实例化了ZookeeperRegistry而已，所有这里就不解释了。

后记

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

该文章讲解了dubbo利用zookeeper来实现注册中心，其中关键的是需要弄明白dubbo在zookeeper中存储的节点层级意义，也就是root层、service层、type层以及url层分别代表什么，其他的逻辑并不复杂大多数调用了zookeeper客户端的能力，有兴趣的同学也可以深入的去了解zookeeper。如果我在哪一部分写的不够到位或者写错了，欢迎给我提意见，我的私人微信号码：HUA799695226。