crazyhzm 265

Dubbo源码解析（四）注册中心——dubbo

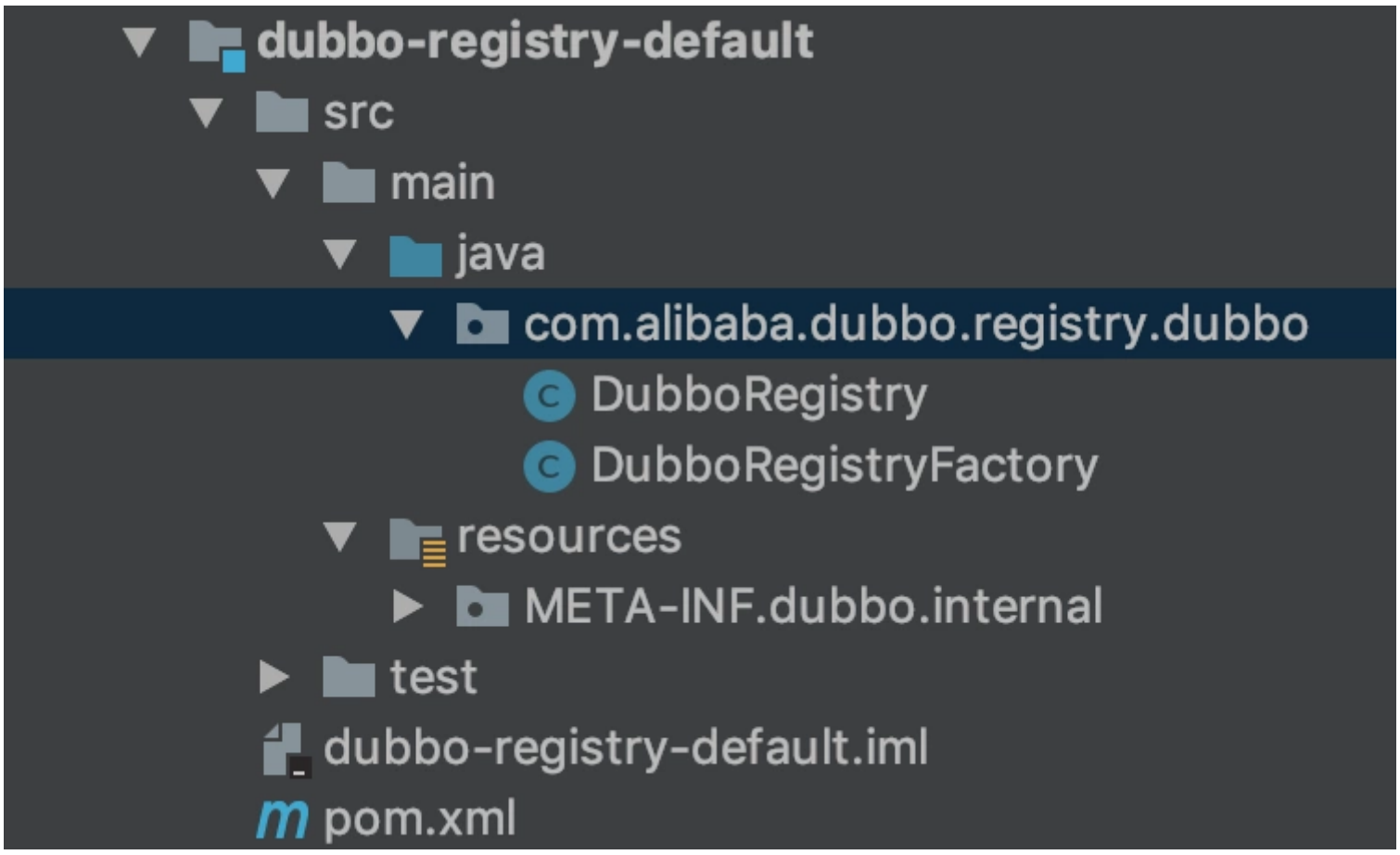
java dubbo 阅读约 21 分钟

注册中心——dubbo

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

dubbo内置的注册中心实现方式有四种，这是第一种，也是dubbo默认的注册中心实现方式。我们可以从上篇文章中看到RegistryFactory接口的@SPI默认值是dubbo。

我们先来看看包下面有哪些类：



可以看到该包下就两个类，下面就来解读这两个类。

（一）DubboRegistry

该类继承了FailbackRegistry类，该类里面封装了一个重连机制，而注册中心核心的功能注册、订阅、取消注册、取消订阅，查询注册列表都是调用了我上一篇文章《dubbo源码解析（三）注册中心——开篇》中讲到的实现方法，毕竟这种实现注册中心的方式是dubbo默认的方式，不过dubbo推荐使用zookeeper，这个后续讲解。

■ 1.属性

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

// Reconnecting detection cycle: 3 seconds (unit:millisecond)
// 重新连接周期：3秒
private static final int RECONNECT_PERIOD_DEFAULT = 3 * 1000;

// Scheduled executor service
// 任务调度器
private final ScheduledExecutorService reconnectTimer = Executors.newScheduledThreadPool(1, new
NamedThreadFactory("DubboRegistryReconnectTimer", true));

// Reconnection timer, regular check connection is available. If unavailable, unlimited reconnection.
// 重新连接执行器，定期检查连接可用，如果不可用，则无限制重连
private final ScheduledFuture<?> reconnectFuture;

// The lock for client acquisition process, lock the creation process of the client instance to prevent repeated
clients
// 客户端的锁，保证客户端的原子性，可见行，线程安全。
private final ReentrantLock clientLock = new ReentrantLock();

// 注册中心Invoker
private final Invoker<RegistryService> registryInvoker;

// 注册中心服务对象
private final RegistryService registryService;
```

看上面的源码，可以看到这里的重连是建立了一个计时器，并且会定期检查连接是否可用，如果不可用，就无限重连。只要懂一点线程相关的知识，这里的属性还是比较好理解的。

■ 2.构造函数DubboRegistry

先来看看源码：

```
public DubboRegistry(Invoker<RegistryService> registryInvoker, RegistryService registryService) {
    // 调用父类FailbackRegistry的构造函数
    super(registryInvoker.getUrl());
    this.registryInvoker = registryInvoker;
    this.registryService = registryService;
    // Start reconnection timer
    // 优先取url中key为reconnect.perio的配置，如果没有，则使用默认的3s
    this.reconnectPeriod = registryInvoker.getUrl().getParameter(Constants.REGISTRY_RECONNECT_PERIOD_KEY,
RECONNECT_PERIOD_DEFAULT);
    // 每reconnectPeriod秒去连接，首次连接也延迟reconnectPeriod秒
    reconnectFuture = reconnectTimer.scheduleWithFixedDelay(new Runnable() {
        @Override
        public void run() {
            // Check and connect to the registry
            try {
                connect();
            } catch (Throwable t) { // Defensive fault tolerance
                logger.error("Unexpected error occur at reconnect, cause: " + t.getMessage(), t);
            }
        }
    }, reconnectPeriod, reconnectPeriod, TimeUnit.MILLISECONDS);
}
```

这个构造方法中有两个关键点：

1. 关于等待时间优先从url配置中取得，如果没有这个值，再设置为默认值3s。
2. 创建了一个重连计时器，一定的间隔时间去检查是否断开，如果断开就进行连接。

■ 3.connect

该方法是连接注册中心的实现，来看看源码：

```
protected final void connect() {
    try {
        // Check whether or not it is connected
        // 检查注册中心是否已连接
        if (isAvailable()) {
            return;
        }
        if (logger.isInfoEnabled()) {
            logger.info("Reconnect to registry " + getUrl());
        }
        // 获得客户端锁
        clientLock.lock();
        try {
            // Double check whether or not it is connected
            // 二次查询注册中心是否已经连接
            if (isAvailable()) {
                return;
            }
            // 恢复注册和订阅
            recover();
        } finally {
            // 释放锁
            clientLock.unlock();
        }
    } catch (Throwable t) { // Ignore all the exceptions and wait for the next retry
        if (getUrl().getParameter(Constants.CHECK_KEY, true)) {
```

我们可以看到这里的重连机制其实就是调用了父类FailbackRegistry的recover方法，关于recover方法我在《dubbo源码解析（三）注册中心——开篇》中已经讲解过了。还有要关注的就是需要保证客户端线程安全。需要获得锁和释放锁。

■ 4.isAvailable

该方法就是用来检查注册中心是否连接，源码如下：

```
public boolean isAvailable() {
    if (registryInvoker == null)
        return false;
    return registryInvoker.isAvailable();
}
```

■ 5.destroy

该方法是销毁方法，主要是销毁重连计时器、注册中心的Invoker和任务调度器，源码如下：

```
@Override
public void destroy() {
    super.destroy();
    try {
        // Cancel the reconnection timer
        // 取消重新连接计时器
        if (!reconnectFuture.isCancelled()) {
            reconnectFuture.cancel(true);
        }
    } catch (Throwable t) {
        logger.warn("Failed to cancel reconnect timer", t);
    }
    // 销毁注册中心的Invoker
    registryInvoker.destroy();
    // 关闭任务调度器
    ExecutorUtil.gracefulShutdown(reconnectTimer, reconnectPeriod);
}
```

这里用到了ExecutorUtil中的gracefulShutdown，因为ExecutorUtil是common模块中的类，我在第一篇中讲到我会穿插在各个文章中介绍这个模块，所以我咋这里介绍一下这个gracefulShutdown方法，我们可以看一下这个源码：

```
    try {
        // Disable new tasks from being submitted
        // 停止接收新的任务并且等待已经提交的任务（包含提交正在执行和提交未执行）执行完成
        // 当所有提交任务执行完毕，线程池即被关闭
        es.shutdown();
    } catch (SecurityException ex2) {
        return;
    } catch (NullPointerException ex2) {
        return;
    }
    try {
        // Wait a while for existing tasks to terminate
        // 当等待超过设定时间时，会监测ExecutorService是否已经关闭，如果没关闭，再关闭一次
        if (!es.awaitTermination(timeout, TimeUnit.MILLISECONDS)) {
            // 试图停止所有正在执行的线程，不再处理还在池队列中等待的任务
            // ShutdownNow()并不代表线程池就一定立即就能退出，它可能必须要等待所有正在执行的任务都执行完成了才能退出。
            es.shutdownNow();
        }
    } catch (InterruptedException ex) {
        es.shutdownNow();
        Thread.currentThread().interrupt();
    }
    if (!isTerminated(es)) {
        newThreadToCloseExecutor(es);
    }
}
```

可以看到这个销毁任务调度器，也就是退出线程池，调用了shutdown、shutdownNow方法，这里也替大家恶补了一下线程池关闭的方法区别。

■ 6.doRegister && doUnregister && doSubscribe && doUnsubscribe && lookup

关于这个五个方法都是调用了RegistryService的方法，读者可自主查看《dubbo源码解析（三）注册中心——开篇》来理解内部实现。

（二）DubboRegistryFactory

该类继承了AbstractRegistryFactory类，实现了AbstractRegistryFactory抽象出来的createRegistry方法，是dubbo这种实现的注册中心的工厂类，里面做了一些初始化的处理，以及创建注册中心DubboRegistry的对象实例。因为该类的属性比较好理解，所以下面就不在展开讲解了。

■ 1.getRegistryURL

获取注册中心url，类似于初始化注册中心url的方法。

```
private static URL getRegistryURL(URL url) {
    return url.setPath(RegistryService.class.getName())
            // 移除暴露服务和引用服务的参数
            .removeParameter(Constants.EXPORT_KEY).removeParameter(Constants.REFER_KEY)
            // 添加注册中心服务接口class值
            .addParameter(Constants.INTERFACE_KEY, RegistryService.class.getName())
            // 启用sticky 粘性连接，让客户端总是连接同一提供者
            .addParameter(Constants.CLUSTER_STICKY_KEY, "true")
            // 决定在创建客户端时建立连接
            .addParameter(Constants.LAZY_CONNECT_KEY, "true")
            // 不重连
            .addParameter(Constants.RECONNECT_KEY, "false")
            // 方法调用超时时间为10s
            .addParameterIfAbsent(Constants.TIMEOUT_KEY, "10000")
            // 每个客户端上一个接口的回调服务实例的限制为10000个
            .addParameterIfAbsent(Constants.CALLBACK_INSTANCES_LIMIT_KEY, "10000")
            // 注册中心连接超时时间10s
            .addParameterIfAbsent(Constants.CONNECT_TIMEOUT_KEY, "10000")
            // 添加方法级配置
            .addParameter(Constants.METHODS_KEY, StringUtils.join(new HashSet<String>
(Arrays.asList(Wrapper.getWrapper(RegistryService.class).getDeclaredMethodNames())), ","))
            //.addParameter(Constants.STUB_KEY, RegistryServiceStub.class.getName())
            //.addParameter(Constants.STUB_EVENT_KEY, Boolean.TRUE.toString()) //for event dispatch
            //.addParameter(Constants.ON_DISCONNECT_KEY, "disconnect")
            .addParameter("subscribe.1.callback", "true")
            .addParameter("unsubscribe.1.callback", "false");
}
```

看上面的源码可以很直白的看书就是对url中的配置做一些初始化设置。几乎每个key对应的意义我都在上面展示了，会比较好理解。

■ 2.createRegistry

该方法就是实现了AbstractRegistryFactory抽象出来的createRegistry方法，该子类就只关注createRegistry方法，其他公共的逻辑都在AbstractRegistryFactory已经实现。看一下源码：

```
@Override
public Registry createRegistry(URL url) {
    // 类似于初始化注册中心
    url = getRegistryURL(url);
    List<URL> urls = new ArrayList<URL>();
    // 移除备用的值
    urls.add(url.removeParameter(Constants.BACKUP_KEY));
    String backup = url.getParameter(Constants.BACKUP_KEY);
    if (backup != null && backup.length() > 0) {
        // 分割备用地址
        String[] addresses = Constants.COMMA_SPLIT_PATTERN.split(backup);
        for (String address : addresses) {
            urls.add(url.setAddress(address));
        }
    }
    // 创建RegistryDirectory，里面有多个Registry的Invoker
    RegistryDirectory<RegistryService> directory = new RegistryDirectory<RegistryService>(RegistryService.class,
url.addParameter(Constants.INTERFACE_KEY,
RegistryService.class.getName()).addParameterAndEncoded(Constants.REFER_KEY, url.toParameterString()));
    // 将directory中的多个Invoker伪装成一个Invoker
    Invoker<RegistryService> registryInvoker = cluster.join(directory);
    // 代理
    RegistryService registryService = proxyFactory.getProxy(registryInvoker);
    // 创建注册中心对象
    DubboRegistry registry = new DubboRegistry(registryInvoker, registryService);
    directory.setRegistry(registry);
```

在这个方法中做了实例化了DubboRegistry，并且做了通知和订阅的操作。相关集群、代理以及包含了很多Invoker的Directory我会在后续文章中讲到。

后记

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