crazyhzm 265

Dubbo源码解析（三）注册中心——开篇

java dubbo 阅读约 62 分钟

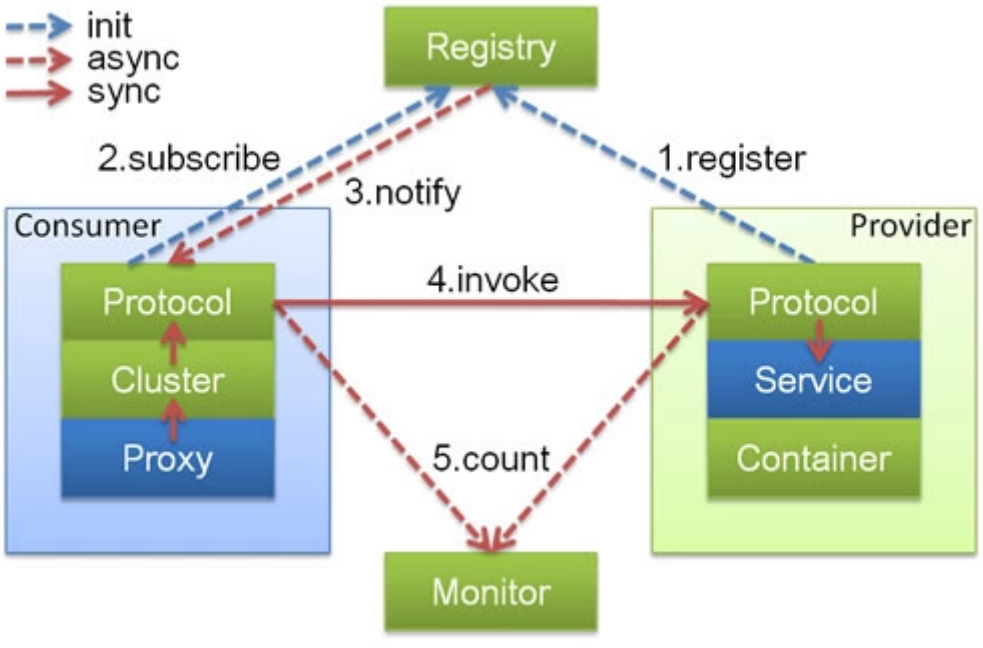
注册中心——开篇

> 目标：解释注册中心在dubbo框架中作用，dubbo-registry-api源码解读

注册中心是什么？

服务治理框架中可以大致分为服务通信和服务管理两个部分，服务管理可以分为服务注册、服务发现以及服务被热加工介入，服务提供者Provider会往注册中心注册服务，而消费者Consumer会从注册中心中订阅相关的服务，并不会订阅全部的服务。

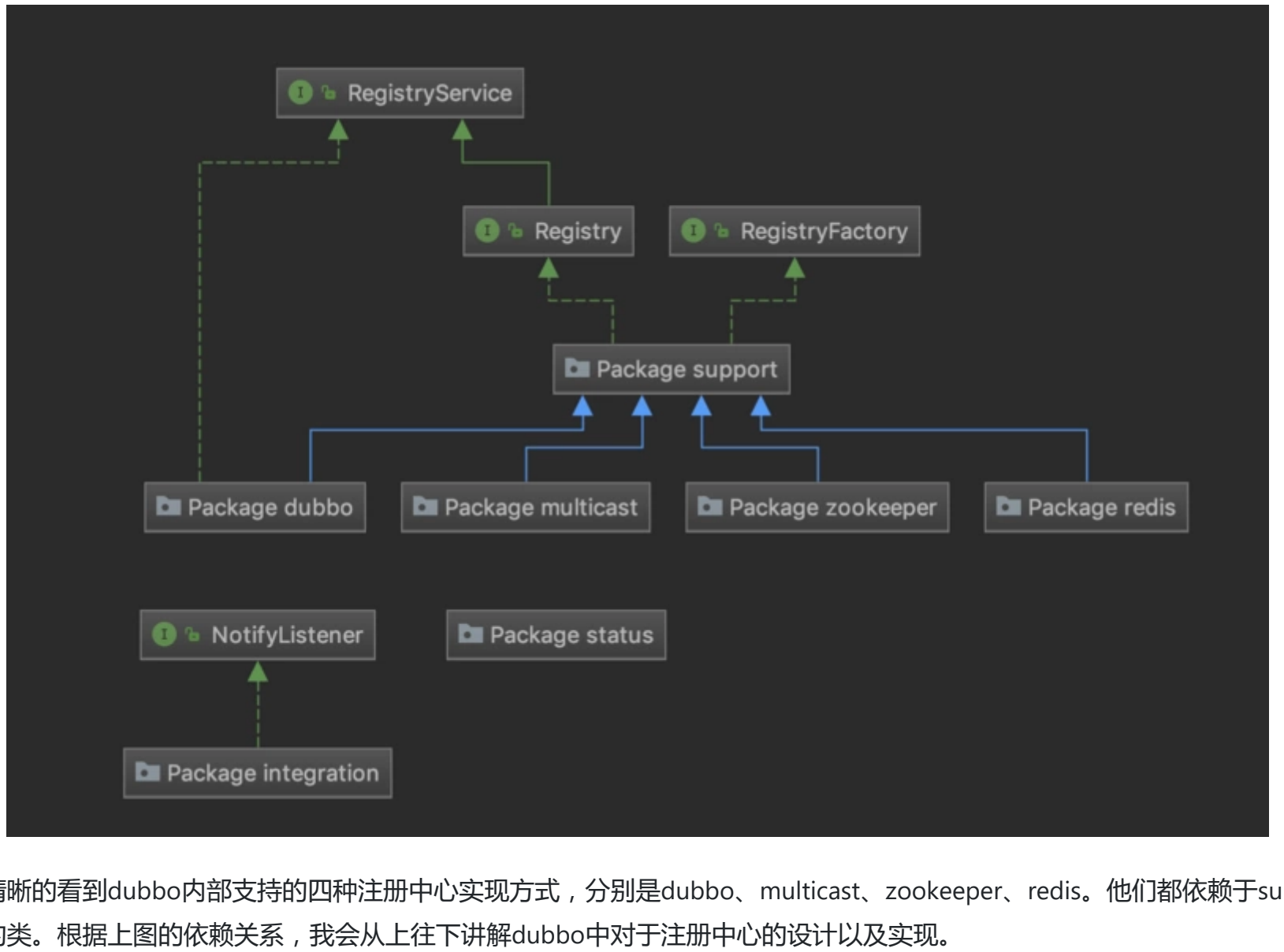
官方文档给出了Provider、Consumer以及Registry之间的依赖关系：



从上图看，可以清晰的看到Registry所起到的作用，我举个例子，Registry类似于一个自动售货机，服务提供者类似于一个商品生产者，他会往这个自动售卖机中添加商品，也就是注册服务，而消费者则会到注册中心中购买自己需要的商品，也就是订阅对应的服务。这样解释应该就可以比较直观的感受到注册中心所担任的是什么角色。

dubbo-registry-api的解读

首先我们来看看这个包下的结构：



可以很清晰的看到dubbo内部支持的四种注册中心实现方式，分别是dubbo、multicast、zookeeper、redis。他们都依赖于support包下面的类。根据上图的依赖关系，我会从上往下讲解dubbo中对于注册中心的设计以及实现。

（一）RegistryService

该接口是注册中心模块的服务接口，提供了注册、取消注册、订阅、取消订阅以及查询符合条件的已注册数据。它的源代码我就不贴出来了，可以查看官方文档中相关部分，还给出了中文注释。

> RegistryService源码地址：http://dubbo.apache.org/zh-cn...

我们可以从注释中看到各个方法要处理的契约都在上面写明了。这个接口就是协定了注册中心的功能，这里统一说明一下URL，又再次提到URL了，在上篇文章中就说明了dubbo是以总线模式来时刻传递和保存配置信息的，也就是配置信息都被放在URL上进行传递，随时可以取得相关配置信息，而这里提到了URL有别的作用，就是作为类似于节点的作用，首先服务提供者（Provider）启动时需要提供服务，就会向注册中心写下自己的URL地址。然后消费者启动时需要去订阅该服务，则会订阅Provider注册的地址，并且消费者也会写下自己的URL。继续拿我上面的例子，商品生产者生产完商品，它会在把该商品放在自动售卖机的某一个栏目内，二消费者需要买该商品的时候，就是通过该地址去购买，并且会留下自己的购买记录。下面来讲讲各个方法：

1. 注册，如果看懂我上面说的url的作用，那么就很清楚该方法的作用了，这里强调一点，就是注释中讲到的允许URl相同但参数不同的URL并存，不能覆盖，也就是说url值必须唯一的，不能有一模一样。

```
void register(URL url);
```

2. 取消注册，该方法也很简单，就是取消注册，也就是商品生产者不在销售该商品， 需要把东西从自动售卖机上取下来，栏目也要取出，这里强调按全URL匹配取消注册。

```
void unregister(URL url);
```

3. 订阅，这里不是根据全URL匹配订阅的，而是根据条件去订阅，也就是说可以订阅多个服务。listener是用来监听处理注册数据变更的事件。

```
void subscribe(URL url, NotifyListener listener);
```

4. 取消订阅，这是按照全URL匹配去取消订阅的。

```
void unsubscribe(URL url, NotifyListener listener);
```

5. 查询注册列表，通过url进行条件查询所匹配的所有URL集合。

```
List<URL> lookup(URL url);
```

（二）Registry

注册中心接口，该接口很好理解，就是把节点以及注册中心服务的方法整合在了这个接口里面。我们来看看源代码：

```
public interface Registry extends Node, RegistryService {
}
```

可以看到该接口并没有自己的方法，就是继承了Node和RegistryService接口。这里的Node是节点的接口，里面协定了关于节点的一些操作方法，我们可以来看看源代码：

```
public interface Node {
    //获得节点地址
    URL getUrl();
    //判断节点是否可用
    boolean isAvailable();
    //销毁节点
    void destroy();

}
```

（三）RegistryFactory

这个接口是注册中心的工厂接口，用来返回注册中心的对象。来看看它的源码：

```
@SPI("dubbo")
public interface RegistryFactory {

    @Adaptive({"protocol"})
    Registry getRegistry(URL url);

}
```

本来方法上有一些英文注释，写的是关于连接注册中心需处理的契约，具体的可以直接看官方文档，还是中文的。

> 地址：http://dubbo.apache.org/zh-cn...

该接口是一个可扩展接口，可以看到该接口上有个@SPI注解，并且默认值为dubbo，也就是默认扩展的是DubboRegistryFactory，并且可以在getRegistry方法上可以看到有@Adaptive注解，那么该接口会动态生成一个适配器RegistryFactory$Adaptive，并且会去首先扩展url.protocol的值对应的实现类。关于SPI扩展机制请观看《dubbo源码解析（二）Dubbo扩展机制SPI》。

（四）NotifyListener

该接口只有一个notify方法，通知监听器。当收到服务变更通知时触发。来看看它的源码：

```
public interface NotifyListener {
    /**
     * 当收到服务变更通知时触发。
     * <p>
     * 通知需处理契约：<br>
     * 1. 总是以服务接口和数据类型为维度全量通知，即不会通知一个服务的同类型的部分数据，用户不需要对比上一次通知结果。<br>
     * 2. 订阅时的第一次通知，必须是一个服务的所有类型数据的全量通知。<br>
     * 3. 中途变更时，允许不同类型的数据分开通知，比如：providers, consumers, routers, overrides，允许只通知其中一种类型，但该类型的数据必须是全量的，不是增量的。<br>
     * 4. 如果一种类型的数据为空，需通知一个empty协议并带category参数的标识性URL数据。<br>
     * 5. 通知者(即注册中心实现)需保证通知的顺序，比如：单线程推送，队列串行化，带版本对比。<br>
     *
     * @param urls 已注册信息列表，总不为空，含义同{@link com.alibaba.dubbo.registry.RegistryService#lookup(URL)}的返回值。
     */
    void notify(List<URL> urls);

}
```

（五）support包下的AbstractRegistry

AbstractRegistry实现的是Registry接口，是Registry的抽象类。为了减轻注册中心的压力，在该类中实现了把本地URL缓存到property文件中的机制，并且实现了注册中心的注册、订阅等方法。

> 源码注释地址：https://github.com/CrazyHZM/i...

■ 1.属性

```
    // URL的地址分隔符，在缓存文件中使用，服务提供者的URL分隔
    private static final char URL_SEPARATOR = ' ';
    // URL地址分隔正则表达式，用于解析文件缓存中服务提供者URL列表
    private static final String URL_SPLIT = "\\s+";
    // 日志输出
    protected final Logger logger = LoggerFactory.getLogger(getClass());
    // 本地磁盘缓存，有一个特殊的key值为registies，记录的是注册中心列表，其他记录的都是服务提供者列表
    private final Properties properties = new Properties();
    // 缓存写入执行器
    private final ExecutorService registryCacheExecutor = Executors.newFixedThreadPool(1, new
NamedThreadFactory("DubboSaveRegistryCache", true));
    // 是否同步保存文件标志
    private final boolean syncSaveFile;
    //数据版本号
    private final AtomicLong lastCacheChanged = new AtomicLong();
    // 已注册 URL 集合
    // 注册的 URL 不仅仅可以是服务提供者的，也可以是服务消费者的
    private final Set<URL> registered = new ConcurrentHashSet<URL>();
    // 订阅URL的监听器集合
    private final ConcurrentMap<URL, Set<NotifyListener>> subscribed = new ConcurrentHashMap<URL,
Set<NotifyListener>>();
    // 某个消费者被通知的某一类型的 URL 集合
    // 第一个key是消费者的URL，对应的就是哪个消费者。
    // value是一个map集合，该map集合的key是分类的意思，例如providers、routes等，value就是被通知的URL集合
    private final ConcurrentMap<URL, Map<String, List<URL>>> notified = new ConcurrentHashMap<URL, Map<String,
List<URL>>>();
```

理解属性的含义对于后面去解读方法很有帮助，从上面可以看到除了注册中心相关的一些属性外，可以看到好几个是个属性跟磁盘缓存文件和读写文件有关的，这就是上面提到的把URL缓存到本地property的相关属性这里有几个需要关注的点：

1. properties：properties的数据跟本地文件的数据同步，当启动时，会从文件中读取数据到properties，而当properties中数据变化时，会写入到file。而properties是一个key对应一个列表，比如说key就是消费者的url，而值就是服务提供者列表、路由规则列表、配置规则列表。就是类似属性notified的含义。需要注意的是properties有一个特殊的key为registies，记录的是注册中心列表。
2. lastCacheChanged：因为每次写入file都是全部覆盖的写入，不是增量的去写入到文件，所以需要有这个版本号来避免老版本覆盖新版本。
3. notified：跟properties的区别是第一数据来源不是文件，而是从注册中心中读取，第二个notified根据分类把同一类的值做了聚合。

■ 2.构造方法AbstractRegistry

先来看看源码：

```
    public AbstractRegistry(URL url) {
        // 把url放到registryUrl中
        setUrl(url);
        // Start file save timer
        // 从url中读取是否同步保存文件的配置，如果没有值默认用异步保存文件
        syncSaveFile = url.getParameter(Constants.REGISTRY_FILESAVE_SYNC_KEY, false);
        // 获得file路径
        String filename = url.getParameter(Constants.FILE_KEY, System.getProperty("user.home") + "/.dubbo/dubbo-
registry-" + url.getParameter(Constants.APPLICATION_KEY) + "-" + url.getAddress() + ".cache");
        File file = null;
        if (ConfigUtils.isNotEmpty(filename)) {
            //创建文件
            file = new File(filename);
            if (!file.exists() && file.getParentFile() != null && !file.getParentFile().exists()) {
                if (!file.getParentFile().mkdirs()) {
                    throw new IllegalArgumentException("Invalid registry store file " + file + ", cause: Failed
to create directory " + file.getParentFile() + "!");
                }
            }
        }
        this.file = file;
        // 把文件里面的数据写入properties
        loadProperties();
        // 通知监听器，URL 变化结果
        notify(url.getBackupUrls());
    }
```

需要关注的几个点：

1. 比如是否同步保存文件、比如保存的文件路径都优先选择URL上的配置，如果没有相关的配置，再选用默认配置。
2. 构造AbstractRegistry会有把文件里面的数据写入到properties的操作以及通知监听器url变化结果，相关方法介绍在下面给出。

■ 3.filterEmpty

```
    protected static List<URL> filterEmpty(URL url, List<URL> urls) {
        if (urls == null || urls.isEmpty()) {
            List<URL> result = new ArrayList<URL>(1);
            result.add(url.setProtocol(Constants.EMPTY_PROTOCOL));
            return result;
        }
        return urls;
    }
```

这个方法的源码都不需要解释了，很简单，就是判断url集合是否为空，如果为空，则把url中key为empty的值加入到集合。该方法只有在notify方法中用到，为了防止通知的URL变化结果为空。

■ 4.doSaveProperties

该方法比较长，我这里不贴源码了，需要的就查看github上的分析，该方法主要是将内存缓存properties中的数据存储到文件中，并且在里面做了版本号的控制，防止老的版本数据覆盖了新版本数据。数据流向是跟loadProperties方法相反。

■ 5.loadProperties

```
    private void loadProperties() {
        if (file != null && file.exists()) {
            InputStream in = null;
            try {
                in = new FileInputStream(file);
                // 把数据写入到内存缓存中
                properties.load(in);
                if (logger.isInfoEnabled()) {
                    logger.info("Load registry store file " + file + ", data: " + properties);
                }
            } catch (Throwable e) {
                logger.warn("Failed to load registry store file " + file, e);
            } finally {
                if (in != null) {
                    try {
                        in.close();
                    } catch (IOException e) {
                        logger.warn(e.getMessage(), e);
                    }
                }
            }
        }
    }
```

该方法就是加载本地磁盘缓存文件到内存缓存，也就是把文件里面的数据写入properties，可以对比doSaveProperties方法，其中关键的实现就是properties.load和properties.store的区别，逻辑并不难。跟doSaveProperties的数据流向相反。

■ 6.getCacheUrls

```
    public List<URL> getCacheUrls(URL url) {
        for (Map.Entry<Object, Object> entry : properties.entrySet()) {
            // key为某个分类，例如服务提供者分类
            String key = (String) entry.getKey();
            // value为某个分类的列表，例如服务提供者列表
            String value = (String) entry.getValue();
            if (key != null && key.length() > 0 && key.equals(url.getServiceKey())
                    && (Character.isLetter(key.charAt(0)) || key.charAt(0) == '_')
                    && value != null && value.length() > 0) {
                //分割出列表的每个值
                String[] arr = value.trim().split(URL_SPLIT);
                List<URL> urls = new ArrayList<URL>();
                for (String u : arr) {
                    urls.add(URL.valueOf(u));
                }
                return urls;
            }
        }
        return null;
    }
```

该方法是获得内存缓存properties中相关value，并且返回为一个集合，从该方法中可以很清楚的看出properties中是存储的什么数据格式。

■ 7.lookup

来看看源码：

```
    @Override
    public List<URL> lookup(URL url) {
        List<URL> result = new ArrayList<URL>();
        // 获得该消费者url订阅的 所有被通知的 服务URL集合
        Map<String, List<URL>> notifiedUrls = getNotified().get(url);
        // 判断该消费者是否订阅服务
        if (notifiedUrls != null && notifiedUrls.size() > 0) {
            for (List<URL> urls : notifiedUrls.values()) {
                for (URL u : urls) {
                    // 判断协议是否为空
                    if (!Constants.EMPTY_PROTOCOL.equals(u.getProtocol())) {
                        // 添加 该消费者订阅的服务URL
                        result.add(u);
                    }
                }
            }
        } else {
            // 原子类 避免在获取注册在注册中心的服务url时能够保证是最新的url集合
            final AtomicReference<List<URL>> reference = new AtomicReference<List<URL>>();
            // 通知监听器。当收到服务变更通知时触发
            NotifyListener listener = new NotifyListener() {
                @Override
                public void notify(List<URL> urls) {
                    reference.set(urls);
                }
            };
```

该方法是实现了RegistryService接口的方法，作用是获得消费者url订阅的服务URL列表。该方法有几个地方有些绕我在这里重点讲解一下：

1. URL可能是消费者URL，也可能是注册在注册中心的服务URL，我在注释中在URL加了修饰，为了能更明白的区分。
2. 订阅了的服务URL一定是在注册中心中注册了的。
3. 关于订阅服务subscribe方法和通知监听器NotifyListener，我会在下面解释。

■ 8.register && unregister

这两个方法实现了RegistryService接口的方法，里面的逻辑很简单，所有我就不贴代码了，以免影响篇幅，如果真想看，可以进到我github查看，下面我会贴出这部分注释github的地址。其中注册的逻辑就是把url加入到属性registered，而取消注册的逻辑就是把url从该属性中移除，该属性在上面有介绍。真正的实现是在FailbackRegistry类中，FailbackRegistry类我会在下面介绍。

■ 9.subscribe && unsubscribe

这两个方法实现了RegistryService接口的方法，分别是订阅和取消订阅，我就贴一个订阅的代码：

```
    @Override
    public void subscribe(URL url, NotifyListener listener) {
        if (url == null) {
            throw new IllegalArgumentException("subscribe url == null");
        }
        if (listener == null) {
            throw new IllegalArgumentException("subscribe listener == null");
        }
        if (logger.isInfoEnabled()) {
            logger.info("Subscribe: " + url);
        }
        // 获得该消费者url 已经订阅的服务 的监听器集合
        Set<NotifyListener> listeners = subscribed.get(url);
        if (listeners == null) {
            subscribed.putIfAbsent(url, new ConcurrentHashSet<NotifyListener>());
            listeners = subscribed.get(url);
        }
        // 添加某个服务的监听器
        listeners.add(listener);
    }
```

从源代码可以看到，其实订阅也就是把服务通知监听器加入到subscribed中，具体的实现也是在FailbackRegistry类中。

■ 10.recover

恢复方法，在注册中心断开，重连成功的时候，会恢复注册和订阅。

```
    protected void recover() throws Exception {
        // register
        //把内存缓存中的registered取出来遍历进行注册
        Set<URL> recoverRegistered = new HashSet<URL>(getRegistered());
        if (!recoverRegistered.isEmpty()) {
            if (logger.isInfoEnabled()) {
                logger.info("Recover register url " + recoverRegistered);
            }
            for (URL url : recoverRegistered) {
                register(url);
            }
        }
        // subscribe
        //把内存缓存中的subscribed取出来遍历进行订阅
        Map<URL, Set<NotifyListener>> recoverSubscribed = new HashMap<URL, Set<NotifyListener>>(getSubscribed());
        if (!recoverSubscribed.isEmpty()) {
            if (logger.isInfoEnabled()) {
                logger.info("Recover subscribe url " + recoverSubscribed.keySet());
            }
            for (Map.Entry<URL, Set<NotifyListener>> entry : recoverSubscribed.entrySet()) {
                URL url = entry.getKey();
                for (NotifyListener listener : entry.getValue()) {
                    subscribe(url, listener);
                }
            }
        }
```

■ 11.notify

```
protected void notify(List<URL> urls) {
    if (urls == null || urls.isEmpty()) return;
    // 遍历订阅URL的监听器集合，通知他们
    for (Map.Entry<URL, Set<NotifyListener>> entry : getSubscribed().entrySet()) {
        URL url = entry.getKey();

        // 匹配
        if (!UrlUtils.isMatch(url, urls.get(0))) {
            continue;
        }
        // 遍历监听器集合，通知他们
        Set<NotifyListener> listeners = entry.getValue();
        if (listeners != null) {
            for (NotifyListener listener : listeners) {
                try {
                    notify(url, listener, filterEmpty(url, urls));
                } catch (Throwable t) {
                    logger.error("Failed to notify registry event, urls: " + urls + ", cause: " + t.getMessage(),
t);
                }
            }
        }
    }
}
protected void notify(URL url, NotifyListener listener, List<URL> urls) {
    if (url == null) {
```

notify方法是通知监听器，url的变化结果，不过变化的是全量数据，全量数据意思就是是以服务接口和数据类型为维度全量通知，即不会通知一个服务的同类型的部分数据，用户不需要对比上一次通知结果。这里要注意几个重点：

1. 发起订阅后，会获取全量数据，此时会调用notify方法。即Registry 获取到了全量数据
2. 每次注册中心发生变更时会调用notify方法虽然变化是增量，调用这个方法的调用方，已经进行处理，传入的urls依然是全量的。
3. listener.notify，通知监听器，例如，有新的服务提供者启动时，被通知，创建新的 Invoker 对象。

■ 12.saveProperties

先来看看源码：

```
private void saveProperties(URL url) {
    if (file == null) {
        return;
    }
    try {
        // 拼接url
        StringBuilder buf = new StringBuilder();
        Map<String, List<URL>> categoryNotified = notified.get(url);
        if (categoryNotified != null) {
            for (List<URL> us : categoryNotified.values()) {
                for (URL u : us) {
                    if (buf.length() > 0) {
                        buf.append(URL_SEPARATOR);
                    }
                    buf.append(u.toFullString());
                }
            }
        }
        // 设置到properties中
        properties.setProperty(url.getServiceKey(), buf.toString());
        // 增加版本号
        long version = lastCacheChanged.incrementAndGet();
        if (syncSaveFile) {
            // 将集合中的数据存储到文件中
            doSaveProperties(version);
        } else {
```

该方法是单个消费者url对应在notified中的数据，保存在到文件，而保存到文件的操作是调用了doSaveProperties方法，该方法跟doSaveProperties的区别是doSaveProperties方法将properties数据全部覆盖性的保存到文件，而saveProperties只是保存单个消费者url的数据。

■ 13.destroy

该方法在JVM关闭时调用，进行取消注册和订阅的操作。具体逻辑就是调用了unregister和unsubscribe方法，有需要看源码的可以进入github查看。

（六）support包下的FailbackRegistry

我在上面讲AbstractRegistry类的时候已经提到了FailbackRegistry，FailbackRegistry继承了AbstractRegistry，AbstractRegistry中的注册订阅等方法，实际上就是一些内存缓存的变化，而真正的注册订阅的实现逻辑在FailbackRegistry实现，并且FailbackRegistry提供了失败重试的机制。

> 源码注释地址：https://github.com/CrazyHZM/i...

■ 1.属性

```
// Scheduled executor service
// 定时任务执行器
private final ScheduledExecutorService retryExecutor = Executors.newScheduledThreadPool(1, new
NamedThreadFactory("DubboRegistryFailedRetryTimer", true));

// Timer for failure retry, regular check if there is a request for failure, and if there is, an unlimited retry
// 失败重试定时器，定时去检查是否有请求失败的，如有，无限次重试。
private final ScheduledFuture<?> retryFuture;

// 注册失败的URL集合
private final Set<URL> failedRegistered = new ConcurrentHashSet<URL>();

// 取消注册失败的URL集合
private final Set<URL> failedUnregistered = new ConcurrentHashSet<URL>();

// 订阅失败的监听器集合
private final ConcurrentMap<URL, Set<NotifyListener>> failedSubscribed = new ConcurrentHashMap<URL,
Set<NotifyListener>>();

// 取消订阅失败的监听器集合
private final ConcurrentMap<URL, Set<NotifyListener>> failedUnsubscribed = new ConcurrentHashMap<URL,
Set<NotifyListener>>();

// 通知失败的URL集合
private final ConcurrentMap<URL, Map<NotifyListener, List<URL>>> failedNotified = new ConcurrentHashMap<URL,
Map<NotifyListener, List<URL>>>();
```

该类的属性比较好理解，也可以很明显看出这些属性都是跟失败重试机制相关。

■ 2.构造函数

```
public FailbackRegistry(URL url) {
    super(url);
    // 从url中读取重试频率，如果为空，则默认5000ms
    this.retryPeriod = url.getParameter(Constants.REGISTRY_RETRY_PERIOD_KEY, Constants.DEFAULT_REGISTRY_RETRY_PERIOD);
    // 创建失败重试定时器
    this.retryFuture = retryExecutor.scheduleWithFixedDelay(new Runnable() {
        @Override
        public void run() {
            // Check and connect to the registry
            try {
                //重试
                retry();
            } catch (Throwable t) { // Defensive fault tolerance
                logger.error("Unexpected error occur at failed retry, cause: " + t.getMessage(), t);
            }
        }
    }, retryPeriod, retryPeriod, TimeUnit.MILLISECONDS);
}
```

构造函数主要是创建了失败重试的定时器，重试频率从URL取，如果没有设置，则默认为5000ms。

■ 3.register && unregister && subscribe && unsubscribe

这四个方法就是注册、取消注册、订阅、取消订阅的具体实现，因为代码逻辑极其相似，所以为放在一起，下面为只贴出注册的源码：

```
            // Sending a registration request to the server side
            // 向注册中心发送一个注册请求
            doRegister(url);
        } catch (Exception e) {
            Throwable t = e;

            // If the startup detection is opened, the Exception is thrown directly.
            // 如果开启了启动时检测，则直接抛出异常
            boolean check = getUrl().getParameter(Constants.CHECK_KEY, true)
                    && url.getParameter(Constants.CHECK_KEY, true)
                    && !Constants.CONSUMER_PROTOCOL.equals(url.getProtocol());
            boolean skipFailback = t instanceof SkipFailbackWrapperException;
            if (check || skipFailback) {
                if (skipFailback) {
                    t = t.getCause();
                }
                throw new IllegalStateException("Failed to register " + url + " to registry " + getUrl().getAddress()
+ ", cause: " + t.getMessage(), t);
            } else {
                logger.error("Failed to register " + url + ", waiting for retry, cause: " + t.getMessage(), t);
            }

            // Record a failed registration request to a failed list, retry regularly
            // 把这个注册失败的url放入缓存，并且定时重试。
            failedRegistered.add(url);
        }
    }
```

可以看到，逻辑很清晰，就是做了一个doRegister的操作，如果失败抛出异常，则加入到失败的缓存中进行重试。为这里要解释的是doRegister，与之对应的还有doUnregister、doSubscribe、doUnsubscribe三个方法，是FailbackRegistry抽象出来的方法，意图在于每种实现注册中心的方法不一样，相对应的注册、订阅等操作也会有所区别，而把这四个方法抽象出现，为了让子类只去关注这四个的实现，比如说redis实现的注册中心跟zookeeper实现的注册中心方式肯定不一样，那么对应的注册订阅等操作也有所不同，那么各自只要去实现该抽象方法即可。

其他的三个方法有需要的可以查看github上的我写的注释。

■ 4.notify

```
@Override
protected void notify(URL url, NotifyListener listener, List<URL> urls) {
    if (url == null) {
        throw new IllegalArgumentException("notify url == null");
    }
    if (listener == null) {
        throw new IllegalArgumentException("notify listener == null");
    }
    try {
        // 通知 url 数据变化
        doNotify(url, listener, urls);
    } catch (Exception t) {
        // Record a failed registration request to a failed list, retry regularly
        // 放入失败的缓存中，重试
        Map<NotifyListener, List<URL>> listeners = failedNotified.get(url);
        if (listeners == null) {
            failedNotified.putIfAbsent(url, new ConcurrentHashMap<NotifyListener, List<URL>>());
            listeners = failedNotified.get(url);
        }
        listeners.put(listener, urls);
        logger.error("Failed to notify for subscribe " + url + ", waiting for retry, cause: " + t.getMessage(),
t);
    }
}

protected void doNotify(URL url, NotifyListener listener, List<URL> urls) {
```

可以看到notify不一样，他还是又回去调用了父类AbstractRegistry的notify，与上述四个方法不一样。

■ 5.revocer

```
@Override
protected void recover() throws Exception {
    // register
    // register 恢复注册，添加到 `failedRegistered` ，定时重试
    Set<URL> recoverRegistered = new HashSet<URL>(getRegistered());
    if (!recoverRegistered.isEmpty()) {
        if (logger.isInfoEnabled()) {
            logger.info("Recover register url " + recoverRegistered);
        }
        for (URL url : recoverRegistered) {
            failedRegistered.add(url);
        }
    }
    // subscribe
    // subscribe 恢复订阅，添加到 `failedSubscribed` ，定时重试
    Map<URL, Set<NotifyListener>> recoverSubscribed = new HashMap<URL, Set<NotifyListener>>(getSubscribed());
    if (!recoverSubscribed.isEmpty()) {
        if (logger.isInfoEnabled()) {
            logger.info("Recover subscribe url " + recoverSubscribed.keySet());
        }
        for (Map.Entry<URL, Set<NotifyListener>> entry : recoverSubscribed.entrySet()) {
            URL url = entry.getKey();
            for (NotifyListener listener : entry.getValue()) {
                addFailedSubscribed(url, listener);
            }
        }
```

重写了父类的recover，将注册和订阅放入到对应的失败缓存中，然后定时重试。

■ 6.retry

该方法中实现了重试的逻辑，分别对注册失败failedRegistered、取消注册失败failedUnregistered、订阅失败failedSubscribed、取消订阅失败failedUnsubscribed、通知监听器失败failedNotified这五个缓存中的元素进行重试，重试的逻辑就是调用了相关的方法，然后从缓存中删除，例如重试注册，先进行doRegister，然后把该url从failedRegistered移除。具体的注释请到GitHub查看。

（七）support包下的AbstractRegistryFactory

该类实现了RegistryFactory接口，抽象了createRegistry方法，它实现了Registry的容器管理。

■ 1.属性

```
// Log output
// 日志记录
private static final Logger LOGGER = LoggerFactory.getLogger(AbstractRegistryFactory.class);

// The lock for the acquisition process of the registry
// 锁，对REGISTRIES访问对竞争控制
private static final ReentrantLock LOCK = new ReentrantLock();

// Registry Collection Map<RegistryAddress, Registry>
// Registry 集合
private static final Map<String, Registry> REGISTRIES = new ConcurrentHashMap<String, Registry>();
```

■ 2.destroyAll

```
public static void destroyAll() {
    if (LOGGER.isInfoEnabled()) {
        LOGGER.info("Close all registries " + getRegistries());
    }
    // Lock up the registry shutdown process
    // 获得锁
    LOCK.lock();
    try {
        for (Registry registry : getRegistries()) {
            try {
                // 销毁
                registry.destroy();
            } catch (Throwable e) {
                LOGGER.error(e.getMessage(), e);
            }
        }
        // 清空缓存
        REGISTRIES.clear();
    } finally {
        // Release the lock
        // 释放锁
        LOCK.unlock();
    }
}
```

该方法作用是销毁所有的Registry对象，并且清除内存缓存，逻辑比较简单，关键就是对REGISTRIES进行同步的操作。

■ 3.getRegistry

```
            .addParameter(Constants.INTERFACE_KEY, RegistryService.class.getName())
            .removeParameters(Constants.EXPORT_KEY, Constants.REFER_KEY);
    // 计算key值
    String key = url.toServiceString();
    // Lock the registry access process to ensure a single instance of the registry
    // 获得锁
    LOCK.lock();
    try {
        Registry registry = REGISTRIES.get(key);
        if (registry != null) {
            return registry;
        }
        // 创建Registry对象
        registry = createRegistry(url);
        if (registry == null) {
            throw new IllegalStateException("Can not create registry " + url);
        }
        // 添加到缓存。
        REGISTRIES.put(key, registry);
        return registry;
    } finally {
        // Release the lock
        // 释放锁
        LOCK.unlock();
    }
}
```

该方法是实现了RegistryFactory接口中的方法，关于key值的计算我会在后续讲解URL的文章中讲到，这里最要注意的是createRegistry，因为AbstractRegistryFactory类把这个方法抽象出来，为了让子类只要关注该方法，比如说redis实现的注册中心和zookeeper实现的注册中心创建方式肯定不同，而他们相同的一些操作都已经在AbstractRegistryFactory中实现。所以只要关注并且实现该抽象方法即可。

（八）support包下的ConsumerInvokerWrapper && ProviderInvokerWrapper

这两个类实现了Invoker接口，分别是服务消费者和服务提供者的Invoker的包装器，其中就包装了一些属性，我们来看看源码：

■ 1.ConsumerInvokerWrapper属性

```
// Invoker 对象
private Invoker<T> invoker;
// 原始url
private URL originUrl;
// 注册中心url
private URL registryUrl;
// 消费者url
private URL consumerUrl;
// 注册中心 Directory
private RegistryDirectory registryDirectory;
```

■ 2.ProviderInvokerWrapper属性

```
// Invoker对象
private Invoker<T> invoker;
// 原始url
private URL originUrl;
// 注册中心url
private URL registryUrl;
// 服务提供者url
private URL providerUrl;
// 是否注册
private volatile boolean isReg;
```

这两个类都被运用在Dubbo QOS中，需要了解Dubbo QOS的可以到官方文档里面查看

> QOS网址：http://dubbo.apache.org/zh-cn...

（九）support包下的ProviderConsumerRegTable

服务提供者和消费者注册表，存储JVM进程中服务提供者和消费者的Invoker，该类也是被运用在QOS中，包括上面的两个类，都跟QOS中的Offline下线服务命令和ls列出消费者和提供者逻辑实现有关系。我们可以看看它的属性：

```
// 服务提供者Invoker集合，key 为服务提供者的url 计算的key，就是url.toServiceString()方法得到的
public static ConcurrentHashMap<String, Set<ProviderInvokerWrapper>> providerInvokers = new ConcurrentHashMap<String,
Set<ProviderInvokerWrapper>>();
// 服务消费者的Invoker集合，key 为服务消费者的url 计算的key，url.toServiceString()方法得到的
public static ConcurrentHashMap<String, Set<ConsumerInvokerWrapper>> consumerInvokers = new ConcurrentHashMap<String,
Set<ConsumerInvokerWrapper>>();
```

可以看到，其实记录的服务提供者、消费者、注册中心中间的调用链，为了从一方出发能够很直观的找到跟它相关联的所有调用链。

该类中的其他方法请自行查看，这部分跟运维命令的实现相关，所以为不在这里讲解。

（十）support包下的SkipFailbackWrapperException

该类是一个dubbo单独创建的异常，在FailbackRegistry中被使用到，自定义的是一个跳过失败重试的异常。

（十一）status包下的RegistryStatusChecker

该类实现了StatusChecker，StatusChecker是一个状态校验的接口，RegistryStatusChecker是它的扩展类，做了一些跟注册中心有关的状态检查和设置。我们来看看源码：

```
@Activate
public class RegistryStatusChecker implements StatusChecker {

    @Override
    public Status check() {
        // 获得所有的注册中心对象
        Collection<Registry> registries = AbstractRegistryFactory.getRegistries();
        if (registries.isEmpty()) {
            return new Status(Status.Level.UNKNOWN);
        }
        Status.Level level = Status.Level.OK;
        StringBuilder buf = new StringBuilder();
        // 拼接注册中心url中的地址
        for (Registry registry : registries) {
            if (buf.length() > 0) {
                buf.append(",");
            }
            buf.append(registry.getUrl().getAddress());
            // 如果注册中心的节点不可用，则拼接disconnected，并且状态设置为error
            if (!registry.isAvailable()) {
                level = Status.Level.ERROR;
                buf.append("(disconnected)");
            } else {
                buf.append("(connected)");
            }
        }
```

第一个关注点就是@Activate注解，也就是RegistryStatusChecker类会自动激活加载。该类就实现了接口的check方法，作用就是给注册中心进行状态检查，并且返回检查结果。

下面讲的是integration下面的两个类RegistryProtocol和RegistryDirectory，这两个类与注册中心核心的逻辑关系没有那么强。RegistryProtocol是对dubbo-rpc-api的依赖集成，RegistryDirectory是对dubbo-cluster的依赖集成。如果看了下面的解析有点糊涂，可以先跳过这部分，等我出了rpc和cluster相关的文章后再回来看就会比较清晰。

（十二）integration包下的RegistryProtocol && RegistryDirectory

- RegistryProtocol实现了Protocol接口，也是Protocol接口等扩展类，但是它可以认为并不是一个真正的协议，他是实际的协议（dubbo . rmi）包装者，这样客户端的请求在一开始如果没有服务端的信息，会先从注册中心拉取服务的注册信息，然后再和服务端直连。RegistryProtocol是基于注册中心发现服务提供者的实现协议。
- RegistryDirectory：注册中心服务，维护着所有可用的远程Invoker或者本地的Invoker。 它的Invoker集合是从注册中心获取的， 它实现了NotifyListener接口实现了回调接口notify方法。比如消费方要调用某远程服务，会向注册中心订阅这个服务的所有服务提供方，订阅时和服务提供方数据有变动时回调消费方的NotifyListener服务的notify方法，回调接口传入所有服务的提供方的url地址然后将urls转化为invokers, 也就是refer应用远程服务。

这两个类等我讲解完rpc和cluster模块之后再进行补充源码解析。

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的注册中心关于服务注册、订阅、服务变更通知等内部逻辑实现，接下来四篇文章我将会讲解dubbo、multicast、zookeeper、redis四种实现注册中心策略的逻辑实现。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。