Dubbo注册中心

注册中心在微服务架构中的作用举足轻重，有了它服务提供者Provider和注册者Consumer就能感知彼此。从下述Dubbo架构图示中可知：①Provider 从容器启动后的初始化阶段便会向注册中心完成注册操作；②Consumer启动初始化阶段完成对所需Provider的订阅操作；③另外在Provider发生变化，需要 通知对应注册了监听此变化的Consumer。

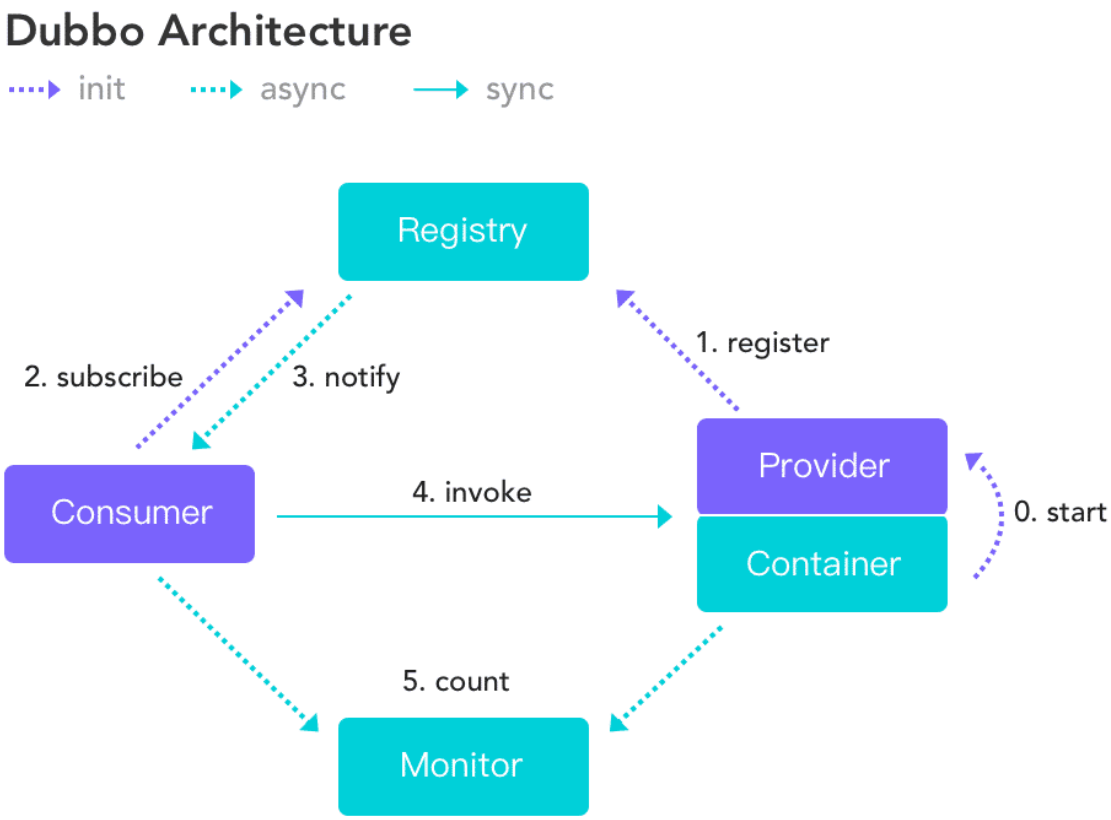

图 1: Dubbo架构

> Registry只是Consumer和Provider能感知彼此状态变化的一种便捷途径而已，彼此的实际通讯交互过程是直接进行的，于Registry是透明无感的。Provider 状态发生变化了，会由Registry主动推送订阅了该Provider的Consumer们，这保证了Consumer感知Provider状态变化的及时性，和具体务需求逻辑交互解耦，也提升了系统的稳定性。

> Dubbo中存在很多概念，有些理解起来就觉得特别费劲。如本文的 `Registry` ，翻译过来的意思是 注册中心 ，但它是应用本地的，真正的注册中心是其他独立部署的进程，或进程组成的集群，比如 `Zookeeper` 。本地的 `Registry` 通过和 `Zookeeper` 等进行实时的信息同步，维持这些内容的一致性，从而实现了 注册中心 这个特性。另外就 `Registry` 而言， `Consumer` 和 `Provider` 只是个用户视觉的概念，他们一律被看做是一份 URL 数据。

Registry定义

注册中心实现目前在业界是一们比较成熟的技术，有多种方案，为了搞清楚Dubbo的注册中心到底是如何运作的，本文将根据 `AbstractRegistry` 源码进行 逐步解析。在进一步阐述之前先看看其实现接口的定义

JAVA

```java
public interface RegistryService {

    /**
     * 注册数据，比如：提供者地址，消费者地址，路由规则，覆盖规则，等数据。
     *
     * 注册需处理契约：<br>
     * 1. 当URL设置了check=false时，注册失败后不报错，在后台定时重试，否则抛出异常。<br>
     * 2. 当URL设置了dynamic=false参数，则需持久存储，否则，当注册者出现断电等情况异常退出时，需自动删除。<br>
     * 3. 当URL设置了category=routers时，表示分类存储，缺省类别为providers，可按分类部分通知数据。<br>
     * 4. 当注册中心重启，网络抖动，不能丢失数据，包括断线自动删除数据。<br>
     * 5. 允许URI相同但参数不同的URL并存，不能覆盖。<br>
     *
     * @param url 注册信息，不允许为空，如：dubbo://10.20.153.10/com.alibaba.foo.BarService?version=1.0.0&application=kylin
     */
    void register(URL url);

    /**
     * 取消注册.
     *
     * 取消注册需处理契约：<br>
     * 1. 如果是dynamic=false的持久存储数据，找不到注册数据，则抛IllegalStateException，否则忽略。<br>
     * 2. 按全URL匹配取消注册。<br>
     *
     * @param url 注册信息，不允许为空，如：dubbo://10.20.153.10/com.alibaba.foo.BarService?version=1.0.0&application=kylin
     */
    void unregister(URL url);

    /**
     * 订阅符合条件的已注册数据，当有注册数据变更时自动推送.
     *
     * 订阅需处理契约：<br>
     * 1. 当URL设置了check=false时，订阅失败后不报错，在后台定时重试。<br>
     * 2. 当URL设置了category=routers，只通知指定分类的数据，多个分类用逗号分隔，并允许星号通配，表示订阅所有分类数据。<br>
     * 3. 允许以interface,group,version,classifier作为条件查询，如：interface=com.alibaba.foo.BarService&version=1.0.0<br>
     * 4. 并且查询条件允许星号通配，订阅所有接口的所有分组的所有版本，或：interface=*&group=*&version=*&classifier=*<br>
     * 5. 当注册中心重启，网络抖动，需自动恢复订阅请求。<br>
     * 6. 允许URI相同但参数不同的URL并存，不能覆盖。<br>
     * 7. 必须阻塞订阅过程，等第一次通知完后再返回。<br>
     *
     * @param url 订阅条件，不允许为空，如：consumer://10.20.153.10/com.alibaba.foo.BarService?version=1.0.0&application=kylin
     * @param listener 变更事件监听器，不允许为空
     */
    void subscribe(URL url, NotifyListener listener);

    /**
     * 取消订阅.
     *
```

```
     * 取消订阅需处理契约：<br>
     * 1. 如果没有订阅，直接忽略。<br>
     * 2. 按全URL匹配取消订阅。<br>
     *
     * @param url 订阅条件，不允许为空，如：
consumer://10.20.153.10/com.alibaba.foo.BarService?version=1.0.0&application=kylin
     * @param listener 变更事件监听器，不允许为空
     */
    void unsubscribe(URL url, NotifyListener listener);

    /**
     * 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。
     *
     * @see com.alibaba.dubbo.registry.NotifyListener#notify(List)
     * @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=kylin
     * @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。
     */
    List<URL> lookup(URL url);

}
```

具体实现

在Dubbo中注册中心的实现方式有多种，包括：①Zookeeper；②Etcd；③Consul；④Redis；⑤Multicast。如上文所示，他们提供的最基础的功能就是 "注册、订阅、通知"这三项，有着很强的共性。最后在微服务这种分布式架构系统中，容错处理不可或缺，注册中心使用本地缓存文件作为容错机制。以下将 从这4方面分开阐述。

> Dubbo中，URL的有着很强的通用性，它**可以完全用于表征某类型的节点**，比如Consumer、Provider

本文中Consumer和Provider代表并不完全就是微服务中所指的服务消费端和服务提供端，是相对于注册、订阅和通知这三个操作而言的，任意节点Node都可以 根据自身需要在Registry注册成Provider或者订阅它所感兴趣的由其它Provider触发的事件。

扫盲——URL、Unmodifiable View、Node

1. URL，统一资源定位符，顾名思义是一个在系统框架中唯一界定资源的标识符可以简单理解为字符串，在Dubbo中URL是一个复杂的存在，作为公共契约的 承载体或称：统一配置模型、配置总线，具体参考： URL 统一模型 @ **Dubbo** (https://dubbo.apache.org/zh-cn/blog/introduction-to-dubbo-url.html)
2. 多线程环境下，即便线程安全的容器，简单的通过获取其引用，后续对其迭代，迭代过程中，其所含元素包括个数及内容可能随时会改变，为了获得所在线程 的当下视图，需要使用到Java集合框架提供的 `unmodifiableXXX` 辅助方法取得当下非可变视图，如下：

```java
public Set<URL> getRegistered() {
    return Collections.unmodifiableSet(registered);
}

public Map<URL, Set<NotifyListener>> getSubscribed() {
    return Collections.unmodifiableMap(subscribed);
}

public Map<URL, Map<String, List<URL>>> getNotified() {
    return Collections.unmodifiableMap(notified);
}
```

3.在Dubbo中，Registry、Consumer、Provider等能够独立部署的节点，均被表示为Node节点，各个具体实现节点需要向往提供获取自身URL、可用状态 检查的方法，以及销毁操作，如下：

```java
public interface Node {

    URL getUrl();

    boolean isAvailable();

    void destroy();
}
```

扫盲——Consumer 和 Provider 的匹配

Registry 实现中，使用 UrlUtils.isMatch(consumerUrl, providerUrl) 来检查 Consumer 和 Provider 的匹配问题，其代码实现虽然很短，但没那么容易理解，和订阅、通知密切相关，有必要在这边剖析一下其细节。

首先是关于 Service Key 的取值，下述两个URL对应的值分别为 com.dubbo.interfaceName① 和 org.dubbo.interfaceName② ，也就是说在 path 和参数 interface 二者中，优先取后者。于Consumer来说，可以使用 interface 作为被引用微服务的标识，而 path 留作它用。

```java
//①
dubbo://admin:hello1234@10.20.130.230:20880/org.dubbo.interfaceName?
interface=com.dubbo.interfaceName&group=group1&version=1.0.0
//②
dubbo://admin:hello1234@10.20.130.230:20880/org.dubbo.interfaceName?
group=group1&version=1.0.0
```

另外方法中还用到了 isMatchCategory(category, categories) ，根据其定义的规则，下述情况下 category 是和 categories 匹配的：

1. categories 值为空， category 值为 "providers" ；
2. categories 值为 "*" ；

3. categories 不包含 "_" + category （含"_"的情况下)；
4. categories 包含 或 等于 category ；

总体而言，按优先顺序，二者需要在 Service Key 、 category 、 enabled 、 group 、 version 、 classifier 这几项均匹配。

JAVA

```java
public static boolean isMatch(URL consumerUrl, URL providerUrl) {
    String consumerInterface = consumerUrl.getServiceInterface();
    String providerInterface = providerUrl.getServiceInterface();

    //等价于 consumerI != * && providerI != * && consumerI != providerI
    //也就是consumerI和providerI都没有设置为通配符时，二者又不相等的情况，肯定不匹配
    if (!(ANY_VALUE.equals(consumerInterface)
            || ANY_VALUE.equals(providerInterface)
            || StringUtils.isEquals(consumerInterface, providerInterface))) {
        return false;
    }

    //若配置的consumer配置的category范围不包含provider所配置，也不匹配
    if (!isMatchCategory(providerUrl.getParameter(CATEGORY_KEY, DEFAULT_CATEGORY),
            consumerUrl.getParameter(CATEGORY_KEY, DEFAULT_CATEGORY))) {
        return false;
    }

    //若provider配置了enabled=false，而consumer没有配置enabled=*，则也不匹配
    if (!providerUrl.getParameter(ENABLED_KEY, true)
            && !ANY_VALUE.equals(consumerUrl.getParameter(ENABLED_KEY))) {
        return false;
    }

    String consumerGroup = consumerUrl.getParameter(GROUP_KEY);
    String consumerVersion = consumerUrl.getParameter(VERSION_KEY);
    String consumerClassifier = consumerUrl.getParameter(CLASSIFIER_KEY, ANY_VALUE);

    String providerGroup = providerUrl.getParameter(GROUP_KEY);
    String providerVersion = providerUrl.getParameter(VERSION_KEY);
    String providerClassifier = providerUrl.getParameter(CLASSIFIER_KEY, ANY_VALUE);

    //最后这里要求group、version、还有consumerClassifier均能匹配
    //在consumer一端，三者均能使用通配符，通配符表示匹配任何值
    return (ANY_VALUE.equals(consumerGroup)
                || StringUtils.isEquals(consumerGroup, providerGroup)
                || StringUtils.isContains(consumerGroup, providerGroup))
            && (ANY_VALUE.equals(consumerVersion)
                || StringUtils.isEquals(consumerVersion, providerVersion))
            && (consumerClassifier == null
                || ANY_VALUE.equals(consumerClassifier)
                || StringUtils.isEquals(consumerClassifier, providerClassifier));
}


public static boolean isMatchCategory(String category, String categories) {
    if (categories == null || categories.length() == 0) {
        return DEFAULT_CATEGORY.equals(category);
    } else if (categories.contains(ANY_VALUE)) {
        return true;
    } else if (categories.contains(REMOVE_VALUE_PREFIX)) {
        return !categories.contains(REMOVE_VALUE_PREFIX + category);
    } else {
        return categories.contains(category);
    }
}
```

注册

Provider在启动后的初始化阶段，会主动向注册中心提交注册信息，同样，需要下线处理时，也会主动发出注销申请。 Provider注册相关的逻辑其实很简单，如下：

JAVA

```java
public abstract class AbstractRegistry implements Registry {

    private final Set<URL> registered = new ConcurrentHashSet<>();
    ...

    public void register(URL url) {
        if (url == null) {
            throw new IllegalArgumentException("register url == null");
        }
        registered.add(url);
    }

    public void unregister(URL url) {
        if (url == null) {
            throw new IllegalArgumentException("unregister url == null");
        }
        registered.remove(url);
    }

    public void destroy() {

        Set<URL> destroyRegistered = new HashSet<>(getRegistered());
        if (!destroyRegistered.isEmpty()) {
            for (URL url : destroyRegistered) {
                //当URL设置了dynamic=false参数，则需持久存储，否则，当注册者出现断电等情况异常退出
时，需自动删除。
                if (url.getParameter(Constants.DYNAMIC_KEY, true)) {
                    try {
                        unregister(url);
                    } catch (Throwable t) {
                        logger.warn("Failed to unregister url " + url + " to registry "
                            + getUrl() + " on destroy, cause: "
                             + t.getMessage(), t);
                    }
                }
            }
        }
        ...
    }

    //恢复方法，在注册中心断开，重连成功的时候
    protected void recover() throws Exception {
        //把内存缓存中的registered取出来遍历进行注册
        Set<URL> recoverRegistered = getRegistered();
        if (!recoverRegistered.isEmpty()) {
            for (URL url : recoverRegistered) {
                register(url);
            }
        }
        ...
    }
    ...
}
```

从上述源码可知，Dubbo使用了并发包下的 `ConcurrentHashSet` 作为所有Provider的注册信息容器，确保了线程安全和Provider注册资源URL的全局唯 一性。

> `recover()` *源码疑惑解析*
>
> 上述关于recover的代码片段，单看起来会觉得很蹊跷，在试图恢复时，仅仅简单地从内存缓存中获取了所有已注册provider的URL视图，然后逐个调用 `register()` 重新注册，然而 `register()` 也仅仅是将URL加入到 `registered` 集合中。
>
> 总体而言，Java是一门纯OOP的编程语言，其继承和多态特性，决定了一个对象的某个方法的具体行为最终取决于该方法的覆写轨迹。也就是说，该方法实际的执行操作由 运行时对象决定的，如果其对应类覆写了父类方法时，调用了 `super.()` ，则父类的行为得到保留，否则会被无情地擦除。
>
> 最终的注册中心实现类并不是直接继承于 `AbstractRegistry` 的，Dubbo要求相应注册中心能够提供基本的重试机制以保证注册中心的可用性。 `FailbackRegistry` 作为其直接继承类，覆写了 `register()` 方法，包含了一序列重试相关逻辑，乃至调用最终在Zookeeper、Etcd集群完成实际注册的 `doRegister(URL url)` 方法。

订阅

Consumer在启动后的初始化阶段，会主动向注册中心提交订阅请求，同样，需要下线处理时，也会主动发出注销申请。每一个Consumer都有唯一的URL，它可以同时订阅 多个感兴趣的事件，具体参考下述源码：

JAVA

```java
public abstract class AbstractRegistry implements Registry {

    private final ConcurrentMap<URL, Set<NotifyListener>> subscribed = new
ConcurrentHashMap<>();

    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");
        }
        Set<NotifyListener> listeners = subscribed.computeIfAbsent(url, n -> new
ConcurrentHashSet<>());
        listeners.add(listener);
    }

    public void unsubscribe(URL url, NotifyListener listener) {
        if (url == null) {
            throw new IllegalArgumentException("unsubscribe url == null");
        }
        if (listener == null) {
            throw new IllegalArgumentException("unsubscribe listener == null");
        }
        Set<NotifyListener> listeners = subscribed.get(url);
        if (listeners != null) {
            listeners.remove(listener);
        }
    }


    //恢复方法，在注册中心断开，重连成功的时候
    protected void recover() throws Exception {
        ...
        //把内存缓存中的subscribed取出来遍历进行订阅
        Map<URL, Set<NotifyListener>> recoverSubscribed = new HashMap<>
(getSubscribed());
        if (!recoverSubscribed.isEmpty()) {
            for (Map.Entry<URL, Set<NotifyListener>> entry :
recoverSubscribed.entrySet()) {
                URL url = entry.getKey();
                for (NotifyListener listener : entry.getValue()) {
                    subscribe(url, listener);
                }
            }
        }
    }

    public void destroy() {
        ...
        //把内存缓存中的subscribed取出来遍历进行取消订阅
        Map<URL, Set<NotifyListener>> destroySubscribed = new HashMap<>
(getSubscribed());
        if (!destroySubscribed.isEmpty()) {
            for (Map.Entry<URL, Set<NotifyListener>> entry :
destroySubscribed.entrySet()) {
```

```
                URL url = entry.getKey();
                for (NotifyListener listener : entry.getValue()) {
                    try {
                        unsubscribe(url, listener);
                    } catch (Throwable t) {
                        logger.warn("Failed to unsubscribe url " + url + " to registry
"
                                + getUrl() + " on destroy, cause: " + t.getMessage(), t);
                    }
                }
            }
        }
    }
    ...
}
```

通过上述源码发现Consumer在订阅感兴趣的事件时，传入的参数只包含*URL*和*NotifyListener*两个参数，这和直觉有些冲突，订阅的事件应该来自 提供服务某些候选的Provider，那Dubbo怎么确定有哪些候选项呢？这得回到上文中提到 `RegistryService` 接口定义及URL实现语义上，实际上接口定义 讲得很清楚，*URL*的*Path部分*标识订阅服务的*Consumer*，而*Query参数部分*则是用于匹配候选*Provider*的。

通知

Consumer和Provider向Registry提供了订阅和注册数据后，Registry会在Provider的状态发生了变化时，根据Consumer的订阅情况，触发相对应事件，将 Consumer所感兴趣的Provider数据 `notify()` 给Consumer。同时Consumer也可以主动 `lookup()` 获取查询所匹配的Provider数据。想进一步搞清楚 3者间关系，先看看其数据定义：

JAVA

```
private final ConcurrentMap<URL, Set<NotifyListener>> subscribed
    = new ConcurrentHashMap<>();

private final Set<URL> registered
    = new ConcurrentHashSet<>();

//key【URL】：表征Consumer的URL
//value【Map<String, List<URL>>】：
//  键为分类标识，值为该分类下所有对应Provider的URL
private final ConcurrentMap<URL, Map<String, List<URL>>> notified
    = new ConcurrentHashMap<>();
```

Provider向Registry注册，提供自身的表征信息URL，Consumer则向Registry订阅其所关注的事件 `NotifyListener` ，在发生相应事件时，Registry会 将所有Provider表征信息URL按Category分组 `notify()` 给Consumer。然而这并不是全貌， `subscribed` 集合的Key中还潜藏着另外一层含义，其URL 携带的参数，用于明确告知Registry，符合哪些特征的Provider表征信息才是它所真正所关注的。也就是说有了这两组数据，就大致知道Provider状态有变动时 该通知哪些Consumer了。

`notified` 用于按Category分组装填那些最近已经通知过Consumer的所有Provider表征信息，这里记录的信息是便于实现 `lookup()` 的。

JAVA

```java
public interface NotifyListener {
    void notify(List<URL> urls);
}

public abstract class AbstractRegistry implements Registry {

    /**
     * Notify changes from the Provider side.
     *
     * @param url      consumer side url
     * @param listener listener
     * @param urls     provider latest urls
     */
    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");
        }
        if ((CollectionUtils.isEmpty(urls))
                //意即Consumer不是匹配任意Interface
                && !Constants.ANY_VALUE.equals(url.getServiceInterface())) {
            logger.warn("Ignore empty notify urls for subscribe url " + url);
            return;
        }

        //按CATEGORY进行分组，将表征Provider的urls中Consumer感兴趣的那些装进result中
        Map<String, List<URL>> result = new HashMap<>();
        for (URL u : urls) {
            //检测Consumer对当前Provider是否感兴趣
            if (UrlUtils.isMatch(url, u)) {
                String category = u.getParameter(Constants.CATEGORY_KEY,
Constants.DEFAULT_CATEGORY);
                List<URL> categoryList = result.computeIfAbsent(category, k -> new
ArrayList<>());
                categoryList.add(u);
            }
        }
        if (result.size() == 0) {
            return;
        }

        //使用表征Consumer的url获得对应感兴趣的
        Map<String, List<URL>> categoryNotified = notified.computeIfAbsent(url, u ->
new ConcurrentHashMap<>());
        for (Map.Entry<String, List<URL>> entry : result.entrySet()) {
            String category = entry.getKey();
            List<URL> categoryList = entry.getValue();

            //将上述按CATEGORY分组的所有Provider置入notified中
            categoryNotified.put(category, categoryList);

            //针对当前CATEGORY分组下的所有Provider回调listener
            listener.notify(categoryList);
            // We will update our cache file after each notification.
```

```
            // When our Registry has a subscribe failure due to network jitter, we can
return at least the existing cache URL.
            saveProperties(url);
        }
    }

    //当参数中所有providers信息有变动时，通知所有订阅他们的consumber们这一变动
    //这些providers属于同一个微服务部署的多个不同实例
    protected void notify(List<URL> urls) {
        if (CollectionUtils.isEmpty(urls)) {
            return;
        }

        for (Map.Entry<URL, Set<NotifyListener>> entry : getSubscribed().entrySet()) {
            URL url = entry.getKey();
            //这里说明传入的一组URL是类似的，要么都能匹配到Consumer，否则全不
            if (!UrlUtils.isMatch(url, urls.get(0))) {
                continue;
            }

            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);
                    }
                }
            }
        }
    }

    @Override
    public List<URL> lookup(URL url) {
        List<URL> result = new ArrayList<>();

        //将所有按Category分组好的Provider的URL信息装入到一维的result中
        Map<String, List<URL>> notifiedUrls = getNotified().get(url);
        if (notifiedUrls != null && notifiedUrls.size() > 0) {
            for (List<URL> urls : notifiedUrls.values()) {
                for (URL u : urls) {
                    if (!Constants.EMPTY_PROTOCOL.equals(u.getProtocol())) {
                        result.add(u);
                    }
                }
            }
        } else {
            //使用原子引用类型保存Provider的URL信息，封装在List中，最初其内容为空
            final AtomicReference<List<URL>> reference = new AtomicReference<>();
            //生成NotifyListener，Dubbo会在Providers状态发生变化时notify给根据URL特征能匹配的
Consumer
            //listener的回调仅仅是将获取到Providers(List<URL>)设置到reference中而已
            NotifyListener listener = reference::set;
            //完成订阅操作，确保listener能被回调到
            subscribe(url, listener); // Subscribe logic guarantees the first notify to
```

```
return
            //获取reference中的内容，并将获取到值设到result返回值中，感觉这里大多数情况下是无用的，
            //除非多线程环境下，刚好要执行这语句的时候，CPU资源已经让渡给其它线程notify操作了
            List<URL> urls = reference.get();
            if (CollectionUtils.isNotEmpty(urls)) {
                for (URL u : urls) {
                    if (!Constants.EMPTY_PROTOCOL.equals(u.getProtocol())) {
                        result.add(u);
                    }
                }
            }
        }
        return result;
    }
    ...
}
```

容错设计

We will update our cache file after each notification. When our Registry has a subscribe failure due to network jitter, we can return at least the existing cache URL.

在 `notify()` 实现中，有上述注释，大意是因为网络抖动导致订阅失败时，为保证服务的可靠性，作为最次的方案，订阅者可向注册中心调用 `public List<URL> getCacheUrls(URL url)` 获取所有匹配到的最近注册在Registry的Provider的URL。

在分布式架构中，作为担纲PRC通讯框架的Dubbo，它解决的是微服务间协作的难题，大多数情况下，仅仅作为Provider和Consumer的一套基础依赖和应用一起打包 部署，Dubbo自身并没有单独部署，本文所述的Registry也仅仅是其中一个依赖模块，由其完成到Zookeeper、Etcd、Consul等Server的注册订阅操作。

大致的设计方案如下： Registry在每次 `notify()` 通知时，均将当前被*notify*的Consumer能匹配到所有Provider的URL组成的List写入到*Properties*中，它的Key值为 Consumer的URL中获取到的*ServiceKey*，根据需求同步或异步地将这些内容在文件锁的辅助下互斥地保存到对应的 `/.dubbo/dubbo-registry-[当前应用名]-[当前 Registry所在的IP地址].cache` 文件中。为了确保在多线程环境下文件的保存不发生冲突，Dubbo使用了基于版本号的乐观锁，只有获取到了最新的版本号， 才能执行。

相关代码如下：

JAVA

```java
public abstract class AbstractRegistry implements Registry {

    private static final int MAX_RETRY_TIMES_SAVE_PROPERTIES = 3;

    private final AtomicInteger savePropertiesRetryTimes = new AtomicInteger();

    // Local disk cache, where the special key value.registries records the list of
registry centers, and the others are the list of notified service providers
    private final Properties properties = new Properties();

    private final AtomicLong lastCacheChanged = new AtomicLong();

    //将文件中缓存的信息恢复到缓存properties中
    private void loadProperties() {
        if (file != null && file.exists()) {
            InputStream in = null;
            try {
                in = new FileInputStream(file);
                properties.load(in);
                if (logger.isInfoEnabled()) {
                    logger.info("Load registry cache file " + file + ", data: " +
properties);
                }
            } catch (Throwable e) {
                logger.warn("Failed to load registry cache file " + file, e);
            } finally {
                if (in != null) {
                    try {
                        in.close();
                    } catch (IOException e) {
                        logger.warn(e.getMessage(), e);
                    }
                }
            }
        }
    }


    private void saveProperties(URL url) {
        if (file == null) {
            return;
        }

        try {
            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());
                    }
                }
            }
```

```
            properties.setProperty(url.getServiceKey(), buf.toString());
            long version = lastCacheChanged.incrementAndGet();
            if (syncSaveFile) {
                doSaveProperties(version);
            } else {
                registryCacheExecutor.execute(new SaveProperties(version));
            }
        } catch (Throwable t) {
            logger.warn(t.getMessage(), t);
        }
    }

    public void doSaveProperties(long version) {
        //只有获得最新版本号才能执行保存操作
        if (version < lastCacheChanged.get()) {
            return;
        }
        if (file == null) {
            return;
        }
        // Save
        try {
            //获取*.lock文件，不存在则新建
            File lockfile = new File(file.getAbsolutePath() + ".lock");
            if (!lockfile.exists()) {
                lockfile.createNewFile();
            }
            //由*.lock文件获取到其持有的锁
            try (RandomAccessFile raf = new RandomAccessFile(lockfile, "rw");
                 FileChannel channel = raf.getChannel()) {
                FileLock lock = channel.tryLock();
                if (lock == null) {
                    throw new IOException("Can not lock the registry cache file " +
file.getAbsolutePath() + ", ignore and retry later, maybe multi java process use the
file, please config: dubbo.registry.file=xxx.properties");
                }
                // Save
                try {
                    if (!file.exists()) {
                        file.createNewFile();
                    }

                    //使用store()操作保存Properties到文件中
                    try (FileOutputStream outputFile = new FileOutputStream(file)) {
                        properties.store(outputFile, "Dubbo Registry Cache");
                    }
                } finally {
                    //释放文件锁
                    lock.release();
                }
            }
        } catch (Throwable e) {
            //如果因锁获取失败等原因导致的异常，对当前操作进行重试处理
            savePropertiesRetryTimes.incrementAndGet();
            if (savePropertiesRetryTimes.get() >= MAX_RETRY_TIMES_SAVE_PROPERTIES) {
                logger.warn("Failed to save registry cache file after retrying " +
MAX_RETRY_TIMES_SAVE_PROPERTIES + " times, cause: " + e.getMessage(), e);
```

```
                saveProperties.set(0);
                return;
            }
            if (version < lastCacheChanged.get()) {
                saveProperties.set(0);
                return;
            } else {
                registryCacheExecutor.execute(new
SaveProperties(lastCacheChanged.incrementAndGet()));
            }
            logger.warn("Failed to save registry cache file, will retry, cause: " +
e.getMessage(), e);
        }
    }
    ...
}
```

假如已经有一个线程整在执行 doSaveProperties() 操作，已经执行到"// Save"这个位置，另外一个线程也视图发起该操作，由于它会获得最新的 version，因此它也能继续往下执行，这时就很有可能发生共享资源的争用，接下来使用的文件锁刚好保证了这种互斥性。

Dubbo为了防止ookeeper等的注册中心出现网络抖动情况而导致Consumer订阅操作无法顺利进行需要以文件的方式缓存 最新Consumer匹配到的Provider的URL信息，在每次 `notify()` 都会调用 `saveProperties()` 将最新数据保存起来。 总所周知，IO是比较费时的，这势必降低效率，因而另外提供一个异步保存这些数据到*.properties文件的操作。

JAVA

```
public abstract class AbstractRegistry implements Registry {
    // File cache timing writing
    private final ExecutorService registryCacheExecutor =
Executors.newFixedThreadPool(1, new NamedThreadFactory("DubboSaveRegistryCache",
true));
    // Is it synchronized to save the file
    private final boolean syncSaveFile;

    private class SaveProperties implements Runnable {
        private long version;

        private SaveProperties(long version) {
            this.version = version;
        }

        @Override
        public void run() {
            doSaveProperties(version);
        }
    }
    ...
}
```

失败重试处理

作为分布式服务的注册模块，其稳定性和容错性的要求会比较苛刻，由于真正负责注册数据处理的是部署在另一台主机的Zookeeper、etcd等网络节点，跨越网络IO 的操作的失败概率很高，因此对应动作相应也会有着比较高的概率会失败，作为服务可靠性保障，重试机制的重要性不言而喻。

从上述代码中我们知道，注册中心有 `notify`、`subscribe`、`unsubscribe`、`register`、`unregister` 5个主要操作，重试也就是针对这几个动作。Dubbo并没有 把这部分实现在基类 `AbstractRegistry` 中，做了扩展实现—— `FailbackRegistry` ，无论是接入Zookeeper还是诸如etcd等其他作为注册中心的分布式协作间 ，*Dubbo*都要求能提供失败重试机制。

`FailbackRegistry` 中定义了如下几个待子类实现向其他分布式中间件实现 `subscribe`、`unsubscribe`、`register`、`unregister` 这4个操作的模板抽象方法：

JAVA

```java
public abstract void doRegister(URL url);

public abstract void doUnregister(URL url);

public abstract void doSubscribe(URL url, NotifyListener listener);

public abstract void doUnsubscribe(URL url, NotifyListener listener);
```

参考定时轮算法 · HashedWheelTimer，在充分理解了定时轮算法后，重试实现的原理其实比较容易理解。

failedRegistered\failedUnregistered

实现比较简单，仅以 `failedRegistered` 为例：

JAVA

```java
public abstract class FailbackRegistry extends AbstractRegistry {
    ...
    private final ConcurrentMap<URL, FailedRegisteredTask> failedRegistered = new
ConcurrentHashMap<URL, FailedRegisteredTask>();

    public void removeFailedRegisteredTask(URL url) {
        failedRegistered.remove(url);
    }

    private void addFailedRegistered(URL url) {
        FailedRegisteredTask oldOne = failedRegistered.get(url);
        if (oldOne != null) {
            return;
        }
        FailedRegisteredTask newTask = new FailedRegisteredTask(url, this);
        oldOne = failedRegistered.putIfAbsent(url, newTask);
        if (oldOne == null) {
            // never has a retry task. then start a new task for retry.
            retryTimer.newTimeout(newTask, retryPeriod, TimeUnit.MILLISECONDS);
        }
    }

    //在Provider发起或者重试register操作时均会调用
    private void removeFailedRegistered(URL url) {
        FailedRegisteredTask f = failedRegistered.remove(url);
        if (f != null) {
            f.cancel();
        }
    }

    ConcurrentMap<URL, FailedRegisteredTask> getFailedRegistered() {
        return failedRegistered;
    }

    @Override
    public void register(URL url) {
        super.register(url);
        removeFailedRegistered(url);
        removeFailedUnregistered(url);
        try {
            // Sending a registration request to the server side
            doRegister(url);
        } catch (Exception e) {
            Throwable t = e;

            // If the startup detection is opened, the Exception is thrown directly.
            boolean check = getUrl().getParameter(Constants.CHECK_KEY, true)
                    && url.getParameter(Constants.CHECK_KEY, true)
                    && !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
            addFailedRegistered(url);
        }
    }

    public abstract void doRegister(URL url);
    ...
}
```

从以上代码不难发现，当Provider向Registry发起 `register` 时，如果该操作失败，若未开启启动检测特性，则会给定时轮添加一个重试任务——`FailedRegisteredTask`， 后者在若干时间获得某个滴答运行时机时会重新执行 `doRegister`，以完成到Zookeeper等的注册操作。

`FailedRegisteredTask` 中重试逻辑如下，回调 `registry()`，再将自身这个任务从 `failedRegistered` 这个任务容器中移除：

JAVA

```
registry.doRegister(url);
registry.removeFailedRegisteredTask(url);
```

failedSubscribed\failedUnsubscribed

这三者的重试逻辑和上述基本相同，在进一步了解前，先看看下述源码。从上文分析得知，*Consumer*订阅事件，是由*Registry*根据其URL参数判别有哪些注册了的Provider 匹配该*Consumer*的，因而其自身（由URL表征）和其订阅的NotifyListener是强绑定关系，再根据重试需要和相应重试Task形成形如 `<<URL,NotifyListener>,*Task>` 的绑定关系。

JAVA

```java
public abstract class FailbackRegistry extends AbstractRegistry {
    ...
    private final ConcurrentMap<Holder, FailedSubscribedTask> failedSubscribed =
        new ConcurrentHashMap<Holder, FailedSubscribedTask>();

    private final ConcurrentMap<Holder, FailedUnsubscribedTask> failedUnsubscribed =
        new ConcurrentHashMap<Holder, FailedUnsubscribedTask>();

    private final ConcurrentMap<Holder, FailedNotifiedTask> failedNotified =
        new ConcurrentHashMap<Holder, FailedNotifiedTask>();

    static class Holder {

        private final URL url;

        private final NotifyListener notifyListener;

        Holder(URL url, NotifyListener notifyListener) {
            if (url == null || notifyListener == null) {
                throw new IllegalArgumentException();
            }
            this.url = url;
            this.notifyListener = notifyListener;
        }

        @Override
        public int hashCode() {
            return url.hashCode() + notifyListener.hashCode();
        }

        @Override
        public boolean equals(Object obj) {
            if (obj instanceof Holder) {
                Holder h = (Holder) obj;
                return this.url.equals(h.url) &&
this.notifyListener.equals(h.notifyListener);
            } else {
                return false;
            }
        }
    }
    ...
}
```

Dubbo认为在执行 `subscribe` 操作时，如果发生异常，那么说明负责承担注册中心角色的Zookeeper等中间件出现了网络抖动，这时会调用 `getCacheUrls(url)` 获取最近缓存的所有匹配当前*Consumer*关注的所有*Providers*，有值则会调用 `notify()` 通知该*Consumer*，否则才发起重试逻辑。如下所示：

JAVA

```java
public abstract class FailbackRegistry extends AbstractRegistry {
    ...
    public void subscribe(URL url, NotifyListener listener) {
        super.subscribe(url, listener);
        removeFailedSubscribed(url, listener);
        try {
            // Sending a subscription request to the server side
            doSubscribe(url, listener);
        } catch (Exception e) {
            Throwable t = e;

            List<URL> urls = getCacheUrls(url);
            if (CollectionUtils.isNotEmpty(urls)) {
                notify(url, listener, urls);
                logger.error("Failed to subscribe " + url + ", Using cached list: " +
urls + " from cache file: "
                        + getUrl().getParameter(FILE_KEY, System.getProperty("user.home") +
"/dubbo-registry-" + url.getHost() + ".cache") + ", cause: " + t.getMessage(), t);
            } else {
                // If the startup detection is opened, the Exception is thrown
directly.
                boolean check = getUrl().getParameter(Constants.CHECK_KEY, true)
                        && url.getParameter(Constants.CHECK_KEY, true);
                boolean skipFailback = t instanceof SkipFailbackWrapperException;
                if (check || skipFailback) {
                    if (skipFailback) {
                        t = t.getCause();
                    }
                    throw new IllegalStateException("Failed to subscribe " + url + ",
cause: " + t.getMessage(), t);
                } else {
                    logger.error("Failed to subscribe " + url + ", waiting for retry,
cause: " + t.getMessage(), t);
                }
            }

            // Record a failed registration request to a failed list, retry regularly
            addFailedSubscribed(url, listener);
        }
    }
    ...
}
```

failedNotified

从上文分析可知 `notify` 实际完成的操作是，就对应Consumer所关注的Provider回调NotifyListener事件，而被关注对象列表时动态变化的，因而也 导致对应的 `notify` 操作实现比较特殊，以下是其所有相关源码。

JAVA

```java
public abstract class FailbackRegistry extends AbstractRegistry {
    ...
    private void removeFailedSubscribed(URL url, NotifyListener listener) {
        Holder h = new Holder(url, listener);
        FailedSubscribedTask f = failedSubscribed.remove(h);
        if (f != null) {
            f.cancel();
        }
        removeFailedUnsubscribed(url, listener);
        removeFailedNotified(url, listener);
    }

    private void removeFailedNotified(URL url, NotifyListener listener) {
        Holder h = new Holder(url, listener);
        FailedNotifiedTask f = failedNotified.remove(h);
        if (f != null) {
            f.cancel();
        }
    }

    private void addFailedNotified(URL url, NotifyListener listener, List<URL> urls) {
        Holder h = new Holder(url, listener);
        FailedNotifiedTask newTask = new FailedNotifiedTask(url, listener);
        FailedNotifiedTask f = failedNotified.putIfAbsent(h, newTask);
        if (f == null) {
            // never has a retry task. then start a new task for retry.
            newTask.addUrlToRetry(urls);
            retryTimer.newTimeout(newTask, retryPeriod, TimeUnit.MILLISECONDS);
        } else {
            // just add urls which needs retry.
            newTask.addUrlToRetry(urls);
        }
    }


    @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 {
            doNotify(url, listener, urls);
        } catch (Exception t) {
            // Record a failed registration request to a failed list, retry regularly
            addFailedNotified(url, listener, urls);
            logger.error("Failed to notify for subscribe " + url + ", waiting for
retry, cause: " + t.getMessage(), t);
        }
    }

    ...
}
```

```
public final class FailedNotifiedTask extends AbstractRetryTask {

    private static final String NAME = "retry subscribe";

    private final NotifyListener listener;

    private final List<URL> urls = new CopyOnWriteArrayList<>();

    public FailedNotifiedTask(URL url, NotifyListener listener) {
        super(url, null, NAME);
        if (listener == null) {
            throw new IllegalArgumentException();
        }
        this.listener = listener;
    }

    public void addUrlToRetry(List<URL> urls) {
        if (CollectionUtils.isEmpty(urls)) {
            return;
        }
        this.urls.addAll(urls);
    }

    public void removeRetryUrl(List<URL> urls) {
        this.urls.removeAll(urls);
    }

    @Override
    protected void doRetry(URL url, FailbackRegistry registry, Timeout timeout) {
        if (CollectionUtils.isNotEmpty(urls)) {
            listener.notify(urls);
            urls.clear();
        }
        //在下一个周期重试当前Task
        reput(timeout, retryPeriod);
    }
}
```

由其对应的Task定义可以看出，如果该Task没有因为 `subscribe/unsubscribe` 操作而调用 `removeFailedSubscribed` 被移除，那么该Task会 一直周期性的运行下去，由 `doRetry()` 的逻辑——只有匹配Provider的urls容器内容不为空的时候，才会回调 `listener.notify()` 执行实际的通知 操作，并随后清理该容器，也就是说某个固定 `<URL,NotifyListener>` 绑定所对应的 `FailedNotifiedTask` 被设计成周期重试任务，并且一旦添加 就会长期缓存在内存中，后续的*notify*重试，仅仅是将对应的匹配*Provider*的*urls*加入到其容器中以等待下一个滴答运行时刻。

recover

最后有个比较特殊的地方是， `recover()` 方法也被覆写了，改为调用 `addFailedRegistered(url)` 和 `addFailedSubscribed(url, listener)` ， 由定时轮驱动异步完成相应的恢复工作。

完结