

Dubbo的服务治理和监听机制源码

1、服务上线下线监听

1.1、监听注册

1.1.1、说明

以cn.enjoy.service.UserService接口的上线下线为例。

服务的上线会在/dubbo/cn.enjoy.service.UserService/providers节点下写入相应的dubbo协议的节点数据，下线就会删除该节点的数据，那么是如何注册对该节点的监听的呢？我们来分析一下源码

1.1.2、源码分析

学过zookeeper的同学应该都知道，如果要监听一个节点的数据变更，就只要客户端注册一个对该节点的监听就可以了，我们来看看dubbo是如何注册监听的。首先我们应该知道，当providers节点发生了数据变更应该通知给谁，肯定是要通知给消费方的，所以这里的客户端指的就是消费方，那么我们只要看看消费方的protocol.refer方法，在服务引用的时候注册了监听。代码最终来到了RegistryProtocol中的doCreateInvoker方法。

```
protected <T> ClusterInvoker<T> doCreateInvoker(DynamicDirectory<T> directory, Cluster
cluster, Registry registry, Class<T> type) {
    directory.setRegistry(registry);
    directory.setProtocol(protocol);
    // all attributes of REFER_KEY
    Map<String, String> parameters = new HashMap<String, String>
(directory.getConsumerUrl().getParameters());
    URL urlToRegistry = new ServiceConfigURL(
        parameters.get(PROTOCOL_KEY) == null ? DUBBO : parameters.get(PROTOCOL_KEY),
        parameters.remove(REGISTER_IP_KEY), 0, getPath(parameters, type), parameters);
    if (directory.isShouldRegister()) {
        directory.setRegisteredConsumerUrl(urlToRegistry);
        //把协议注册到 /dubbo/cn.enjoy.userService/consumers节点下面
        registry.register(directory.getRegisteredConsumerUrl());
    }
    //创建路由链
    directory.buildRouterChain(urlToRegistry);
    //订阅事件, 对 configurations, providers, routes节点建立监听
    directory.subscribe(toSubscribeUrl(urlToRegistry));

    //返回默认的 FailoverClusterInvoker对象
    return (ClusterInvoker<T>) cluster.join(directory);
}
```

directory.subscribe(toSubscribeUrl(urlToRegistry));就是对providers节点注册了监听。

```

@Override
public void subscribe(URL url) {
    setSubscribeUrl(url);
    CONSUMER_CONFIGURATION_LISTENER.addNotifyListener(this);
    //订阅事件 对 config中的 xxx.xx.xx.xx::configurations
    referenceConfigurationListener = new ReferenceConfigurationListener(this, url);
    //订阅其他事件, configurations routes, providers
    registry.subscribe(url, this);
}

```

registry.subscribe(url, this);这行代码就是注册监听的代码，其实这里注册的监听目录有三个，configurations routes, providers。代码最终来到了ZookeeperRegistry中的doSubscribe方法。

```

@Override
public void doSubscribe(final URL url, final NotifyListener listener) {
    try {
        if (ANY_VALUE.equals(url.getServiceInterface())) {
            String root = toRootPath();
            ConcurrentMap<NotifyListener, ChildListener> listeners =
            zkListeners.computeIfAbsent(url, k -> new ConcurrentHashMap<>());
            ChildListener zkListener = listeners.computeIfAbsent(listener, k ->
            (parentPath, currentChlds) -> {
                for (String child : currentChlds) {
                    child = URL.decode(child);
                    if (!anyServices.contains(child)) {
                        anyServices.add(child);
                        subscribe(url.setPath(child).addParameters(INTERFACE_KEY, child,
                            Constants.CHECK_KEY, String.valueOf(false)), k);
                    }
                }
            });
            zkClient.create(root, false);
            List<String> services = zkClient.addChildListener(root, zkListener);
            if (CollectionUtils.isEmpty(services)) {
                for (String service : services) {
                    service = URL.decode(service);
                    anyServices.add(service);
                    subscribe(url.setPath(service).addParameters(INTERFACE_KEY, service,
                        Constants.CHECK_KEY, String.valueOf(false)), listener);
                }
            }
        } else {
            CountDownLatch latch = new CountDownLatch(1);
            List<URL> urls = new ArrayList<>();
            //这里对应 configurators, providers, routes目录
            for (String path : toCategoriesPath(url)) {
                ConcurrentMap<NotifyListener, ChildListener> listeners =
                zkListeners.computeIfAbsent(url, k -> new ConcurrentHashMap<>());
                //RegistryChildListenerImpl 事件回调类, zookeeper事件回调到它
                ChildListener zkListener = listeners.computeIfAbsent(listener, k -> new
                RegistryChildListenerImpl(url, path, k, latch));
                if (zkListener instanceof RegistryChildListenerImpl) {

```

```

        ((RegistryChildListenerImpl) zkListener).setLatch(latch);
    }
    zkClient.create(path, false);
    //这里会注册zookeeper事件, 并且把zookeeper事件和RegistryChildListenerImpl做映射
    List<String> children = zkClient.addChildListener(path, zkListener);
    if (children != null) {
        //弄一个empty协议, 做初始化工作, 比如清空集合容器
        urls.addAll(toUrlsWithEmpty(url, path, children));
    }
}
//启动后做初始化式的触发监听
notify(url, listener, urls);
// tells the listener to run only after the sync notification of main thread
finishes.
    latch.countDown();
}
} catch (Throwable e) {
    throw new RpcException("Failed to subscribe " + url + " to zookeeper " + getUrl() +
        ", cause: " + e.getMessage(), e);
}
}

```

toCategoriesPath(url)该方法根据dubbo协议中的category属性的值来得到需要监听的目录, 目录就有三个configurators, providers, routes。

RegistryChildListenerImpl是zookeeper的实例监听类回调的逻辑类

```
List<String> children = zkClient.addChildListener(path, zkListener);
```

在这行代码这里注册了zookeeper的监听, 并且把RegistryChildListenerImpl实例传递过去了。

```

@Override
public List<String> addChildListener(String path, final ChildListener listener) {
    ConcurrentMap<ChildListener, TargetChildListener> listeners =
childListeners.computeIfAbsent(path, k -> new ConcurrentHashMap<>());
    //ChildListener和zookeeper事件做了映射
    TargetChildListener targetListener = listeners.computeIfAbsent(listener, k ->
createTargetChildListener(path, k));
    //添加zookeeper事件监听
    return addTargetChildListener(path, targetListener);
}

```

监听实例

```

@Override
public CuratorZookeeperClient.CuratorWatcherImpl createTargetChildListener(String path,
ChildListener listener) {
    return new CuratorZookeeperClient.CuratorWatcherImpl(client, listener, path);
}

```

```
//注册zookeeper的监听,
@Override
public List<String> addTargetChildListener(String path, CuratorWatcherImpl listener) {
    try {
        return client.getChildren().usingWatcher(listener).forPath(path);
    } catch (NoNodeException e) {
        return null;
    } catch (Exception e) {
        throw new IllegalStateException(e.getMessage(), e);
    }
}
```

上面的分析我们知道了，其实我们是对某个接口的三个节点注册了监听，configurators，providers，routes这三个节点都注册上了监听了。

1.2、监听触发

1.2.1、说明

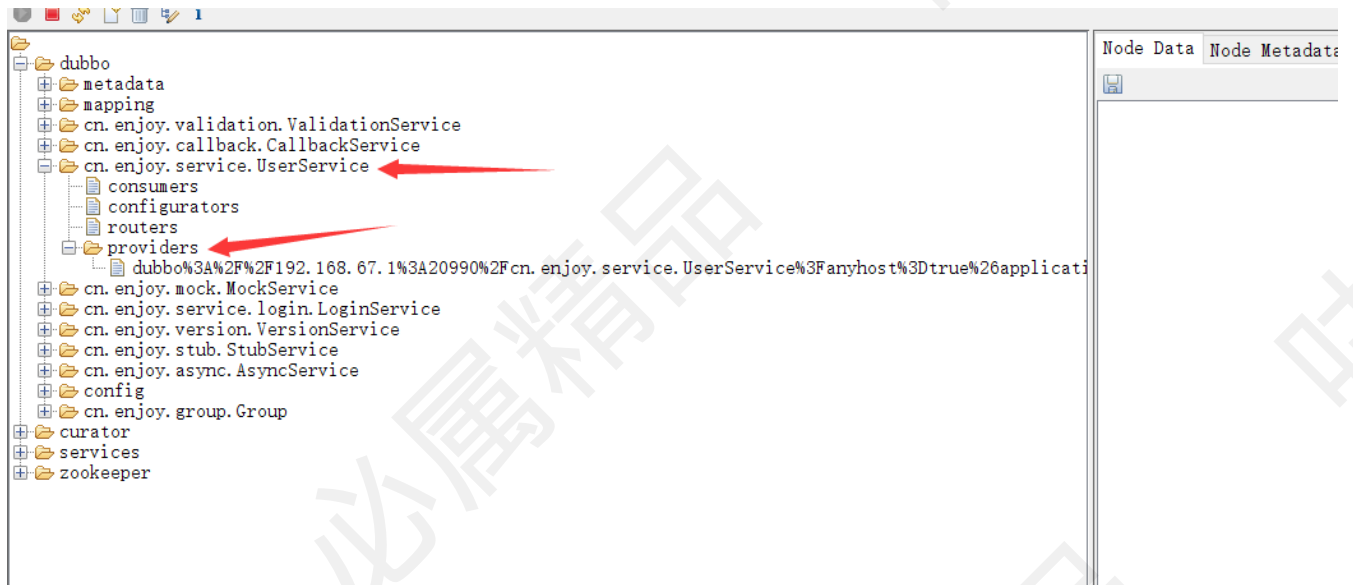
学过zookeeper的同学应该都知道，监听的触发只有我们有对监听的节点发生数据的变更就会触发监听的客户端，比如服务的上线下线就会对providers节点进行数据新增和删除操作就会触发事件。

1.2.2、源码分析

前面我们分析了监听的注册，下面我们来分析一下监听的触发源码，比如我们往providers节点写入一个数据，那么监听应该要触发，往providers写数据，我们采用dubbo api的方式去写一个dubbo协议的数据写到providers节点下，如下：

```
@Test
public void providerReg() {
    String url =
        "dubbo%3A%2F%2F192.168.67.3%3A20990%2Fcn.enjoy.service.UserService%3Fanyhost%3Dtrue%26application%3Ddubbo_provider%26deprecated%3Dfalse%26dubbo%3D2.0.2%26dynamic%3Dtrue%26generic%3Dfalse%26interface%3Dcn.enjoy.service.UserService%26metadata-type%3Dremote%26methods%3DdoKill%2CqueryUser%26pid%3D14092%26release%3D3.0.2.1%26retries%3D7%26revision%3D1.0-SNAPSHOT%26service-name-mapping%3Dtrue%26side%3Dprovider%26threadpool%3Dfixed%26threads%3D100%26timeout%3D5000%26timestamp%3D1635058443480";
    RegistryFactory registryFactory =
        ExtensionLoader.getExtensionLoader(RegistryFactory.class).getAdaptiveExtension();
    Registry registry =
        registryFactory.getRegistry(URL.valueOf("zookeeper://127.0.0.1:2181"));
    registry.register(URL.valueOf(URL.decode(url)));
}
```

往dubbo/cn.enjoy.service.UserService/providers/节点写入dubbo协议



当写入时，会触发客户端的监听代码，我们来看看监听类，zookeeper会回调到process方法来触发事件。childListener就是前面我们分析的RegistryChildListenerImpl类。

```
static class CuratorWatcherImpl implements CuratorWatcher {

    private CuratorFramework client;
    private volatile ChildListener childListener;
    private String path;

    public CuratorWatcherImpl(CuratorFramework client, ChildListener listener, String path)
    {
        this.client = client;
        this.childListener = listener;
        this.path = path;
    }

    protected CuratorWatcherImpl() {
    }

    public void unwatch() {
        this.childListener = null;
    }

    @Override
    public void process(WatchedEvent event) throws Exception {
        // if client connect or disconnect to server, zookeeper will queue
        // watched event(Watcher.Event.EventType.None, ..., path = null).
        if (event.getType() == Watcher.Event.EventType.None) {
            return;
        }

        if (childListener != null) {
            childListener.childChanged(path,
            client.getChildren().usingWatcher(this).forPath(path));
        }
    }
}
```

```
}
```

接下来我们看一下事件触发后都干了些什么。

```
private class RegistryChildListenerImpl implements ChildListener {
    private RegistryNotifier notifier;
    private long lastExecuteTime;
    private volatile CountDownLatch latch;

    public RegistryChildListenerImpl(URL consumerUrl, String path, NotifyListener listener,
    CountDownLatch latch) {
        this.latch = latch;
        notifier = new RegistryNotifier(ZookeeperRegistry.this.getDelay()) {
            @Override
            public void notify(Object rawAddresses) {
                long delayTime = getDelayTime();
                if (delayTime <= 0) {
                    this.doNotify(rawAddresses);
                } else {
                    long interval = delayTime - (System.currentTimeMillis() -
lastExecuteTime);
                    if (interval > 0) {
                        try {
                            Thread.sleep(interval);
                        } catch (InterruptedException e) {
                            // ignore
                        }
                    }
                    lastExecuteTime = System.currentTimeMillis();
                    this.doNotify(rawAddresses);
                }
            }
        };

        @Override
        protected void doNotify(Object rawAddresses) {
            ZookeeperRegistry.this.notify(consumerUrl, listener,
            ZookeeperRegistry.this.toUrlsWithEmpty(consumerUrl, path, (List<String>) rawAddresses));
        }
    };
}

public void setLatch(CountDownLatch latch) {
    this.latch = latch;
}

@Override
public void childChanged(String path, List<String> children) {
    try {
        latch.await();
    } catch (InterruptedException e) {
        logger.warn("Zookeeper children listener thread was interrupted unexpectedly,
may cause race condition with the main thread.");
    }
}
```

```

        notifier.notify(children);
    }
}

```

在这里触发了逻辑

```

protected void doNotify(Object rawAddresses) {
    ZookeeperRegistry.this.notify(consumerUrl, listener,
    ZookeeperRegistry.this.toUrlsWithEmpty(consumerUrl, path, (List<String>) rawAddresses));
}

```

listener.notify(categoryList);触发了RegistryDirectory的notify逻辑，我们重点看看这里。

```

for (Map.Entry<String, List<URL>> entry : result.entrySet()) {
    String category = entry.getKey();
    List<URL> categoryList = entry.getValue();
    categoryNotified.put(category, categoryList);
    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.
    if (localCacheEnabled) {
        saveProperties(url);
    }
}
}

```

RegistryDirectory的notify逻辑

```

//事件监听回调
@Override
public synchronized void notify(List<URL> urls) {
    if (isDestroyed()) {
        return;
    }

    //对回调的协议分组
    // routes://
    // override://
    //dubbo://
    Map<String, List<URL>> categoryUrls = urls.stream()
        .filter(Objects::nonNull)
        .filter(this::isValidCategory)
        .filter(this::isNotCompatibleFor26x)
        .collect(Collectors.groupingBy(this::judgeCategory));

    List<URL> configuratorURLs = categoryUrls.getDefault(CONFIGURATORS_CATEGORY,
    Collections.emptyList());
    this.configurators =
    Configurator.toConfigurators(configuratorURLs).orElse(this.configurators);
}

```

```

List<URL> routerURLs = categoryURLs.getDefault(ROUTERS_CATEGORY,
Collections.emptyList());
//生成路由规则，加入到规则链中
toRouters(routerURLs).ifPresent(this::addRouters);

// providers
List<URL> providerURLs = categoryURLs.getDefault(PROVIDERS_CATEGORY,
Collections.emptyList());
/**
 * 3.x added for extend URL address
 */
ExtensionLoader<AddressListener> addressListenerExtensionLoader =
ExtensionLoader.getExtensionLoader(AddressListener.class);
List<AddressListener> supportedListeners =
addressListenerExtensionLoader.getActivateExtension(getUrl(), (String[]) null);
if (supportedListeners != null && !supportedListeners.isEmpty()) {
    for (AddressListener addressListener : supportedListeners) {
        providerURLs = addressListener.notify(providerURLs, getConsumerUrl(), this);
    }
}
//刷本地服务列表
refreshOverrideAndInvoker(providerURLs);
}

```

如果触发的是一个providers协议，那么configuratorURLs和routerURLs都是空的。那么直接走到了refreshOverrideAndInvoker(providerURLs);

```

private synchronized void refreshOverrideAndInvoker(List<URL> urls) {
    // mock zookeeper://xxx?mock=return null
    overrideDirectoryUrl();
    //刷本地列表
    refreshInvoker(urls);
}

```

直接走到了refreshInvoker(urls);这个方法的大概意思就是刷本地列表，就是刷新invokers变量的值。

```

//刷本地服务列表
private void refreshInvoker(List<URL> invokerURLs) {
    Assert.notNull(invokerURLs, "invokerURLs should not be null");

    if (invokerURLs.size() == 1
        && invokerURLs.get(0) != null
        && EMPTY_PROTOCOL.equals(invokerURLs.get(0).getProtocol())) {
        this.forbidden = true; // Forbid to access
        this.invokers = Collections.emptyList();
        routerChain.setInvokers(this.invokers);
        destroyAllInvokers(); // Close all invokers
    } else {
        this.forbidden = false; // Allow to access
        Map<URL, Invoker<T>> oldUrlInvokerMap = this.urlInvokerMap; // local reference
        if (invokerURLs == Collections.<URL>emptyList()) {
            invokerURLs = new ArrayList<>();

```



```

    }
    if (invokerUrls.isEmpty() && this.cachedInvokerUrls != null) {
        invokerUrls.addAll(this.cachedInvokerUrls);
    } else {
        this.cachedInvokerUrls = new HashSet<>();
        this.cachedInvokerUrls.addAll(invokerUrls); //Cached invoker urls, convenient
for comparison
    }
    if (invokerUrls.isEmpty()) {
        return;
    }
    //创建url和invoker对象的映射关系
    Map<URL, Invoker<T>> newUrlInvokerMap = toInvokers(invokerUrls); // Translate url
list to Invoker map

    /**
     * If the calculation is wrong, it is not processed.
     *
     * 1. The protocol configured by the client is inconsistent with the protocol of
the server.
     *     eg: consumer protocol = dubbo, provider only has other protocol
services(rest).
     * 2. The registration center is not robust and pushes illegal specification data.
     *
     */
    if (CollectionUtils.isEmptyMap(newUrlInvokerMap)) {
        logger.error(new IllegalStateException("urls to invokers error
.invokerUrls.size : " + invokerUrls.size() + ", invoker.size : 0. urls : " + invokerUrls
.toString()));
        return;
    }

    //所有的invoker对象
    List<Invoker<T>> newInvokers = Collections.unmodifiableList(new ArrayList<>
(newUrlInvokerMap.values()));
    // pre-route and build cache, notice that route cache should build on original
Invoker list.
    // toMergeMethodInvokerMap() will wrap some invokers having different groups, those
wrapped invokers not should be routed.
    routerChain.setInvokers(newInvokers);
    this.invokers = multiGroup ? toMergeInvokerList(newInvokers) : newInvokers;
    this.urlInvokerMap = newUrlInvokerMap;

    try {
        destroyUnusedInvokers(oldUrlInvokerMap, newUrlInvokerMap); // close the unused
Invoker
    } catch (Exception e) {
        logger.warn("destroyUnusedInvokers error. ", e);
    }

    // notify invokers refreshed
    this.invokersChanged();
}

```

```
}
```

会在

```
//创建url和invoker对象的映射关系 Map<URL, Invoker> newUrlInvokerMap = toInvokers(invokerUrls);
```

这行代码中根据传递进来的dubbo协议去生成invoker对象，其实就是调用了protocol.refer方法生成了invoker对象了。

```
private Map<URL, Invoker<T>> toInvokers(List<URL> urls) {
    Map<URL, Invoker<T>> newUrlInvokerMap = new ConcurrentHashMap<>();
    if (urls == null || urls.isEmpty()) {
        return newUrlInvokerMap;
    }
    String queryProtocols = this.queryMap.get(PROTOCOL_KEY);
    for (URL providerUrl : urls) {
        // If protocol is configured at the reference side, only the matching protocol is
        // selected
        if (queryProtocols != null && queryProtocols.length() > 0) {
            boolean accept = false;
            String[] acceptProtocols = queryProtocols.split(",");
            for (String acceptProtocol : acceptProtocols) {
                if (providerUrl.getProtocol().equals(acceptProtocol)) {
                    accept = true;
                    break;
                }
            }
            if (!accept) {
                continue;
            }
        }
        if (EMPTY_PROTOCOL.equals(providerUrl.getProtocol())) {
            continue;
        }
        if
        (!ExtensionLoader.getExtensionLoader(Protocol.class).hasExtension(providerUrl.getProtocol()
        )) {
            logger.error(new IllegalStateException("Unsupported protocol " +
            providerUrl.getProtocol() +
                " in notified url: " + providerUrl + " from registry " +
            getUrl().getAddress() +
                " to consumer " + NetUtils.getLocalHost() + ", supported protocol: " +
            ExtensionLoader.getExtensionLoader(Protocol.class).getSupportedExtensions()));
            continue;
        }
        URL url = mergeUrl(providerUrl);

        // Cache key is url that does not merge with consumer side parameters, regardless
        // of how the consumer combines parameters, if the server url changes, then refer again
        Map<URL, Invoker<T>> localUrlInvokerMap = this.urlInvokerMap; // local reference
        Invoker<T> invoker = localUrlInvokerMap == null ? null :
        localUrlInvokerMap.remove(url);
```

```

        if (invoker == null) { // Not in the cache, refer again
            try {
                boolean enabled = true;
                if (url.getParameter(DISABLED_KEY)) {
                    enabled = !url.getParameter(DISABLED_KEY, false);
                } else {
                    enabled = url.getParameter(ENABLED_KEY, true);
                }
                if (enabled) {
                    //生成invoker对象
                    invoker = protocol.refer(serviceType, url);
                }
            } catch (Throwable t) {
                logger.error("Failed to refer invoker for interface:" + serviceType +
                    ",url:(" + url + ") " + t.getMessage(), t);
            }
            if (invoker != null) { // Put new invoker in cache
                newUrlInvokerMap.put(url, invoker);
            }
        } else {
            newUrlInvokerMap.put(url, invoker);
        }
    }
    return newUrlInvokerMap;
}

```

然后在这行代码

```

this.invokers = multiGroup ? toMergeInvokerList(newInvokers) : newInvokers;

```

刷新了服务列表。

我们从上面的分析中可以看到，其实触发的事件最终的目的就是为了刷新客户端本地的服务列表，把新注册的dubbo协议通过protocol.refer生成了一个新的invoker对象，然后加入到了本地服务列表中。

2、动态配置的监听

在dubbo中我们可以对配置文件进行分布式管理，同时我们也可以在运行时去修改dubbo协议中的参数让它在运行时生效，这些都属性配置的动态修改，是服务治理的一种，dubbo有一个同一的服务治理后台就是dubbo-admin。dubbo-admin的下载在第一节课的文档中有，只要从GitHub上下载就可以了，我们去修改接口的属性看看。比如修改UserService接口的retries属性，我们看看zookeeper有什么变化。

服务名	组	版本	应用	操作
cn.enjoy.async.AsyncService			dubbo_provider	详情 测试 更多
cn.enjoy.callback.CallbackService			dubbo_provider	详情 测试 更多
cn.enjoy.group.Group	groupImpl1		dubbo_provider	详情 测试 更多
cn.enjoy.group.Group	groupImpl2		dubbo_provider	详情 测试 更多
cn.enjoy.mock.MockService			dubbo_provider	详情 测试 更多
cn.enjoy.service.UserService			dubbo_provider	详情 测试 更多
cn.enjoy.stub.StubService			dubbo_provider	详情 测试 更多
cn.enjoy.validation.ValidationService			dubbo_provider	详情 测试 更多
cn.enjoy.version.VersionService		1.0.0	dubbo_provider	详情 测试 更多
cn.enjoy.version.VersionService		2.0.0	dubbo_provider	详情 测试 更多
每页行数: 10 1-10 共 10 条				<div><div>条件路由</div><div>标签路由</div><div>动态配置</div><div>黑白名单</div><div>权重调整</div><div>负载均衡</div></div>

动态配置

服务治理 / 动态配置

搜索动态配置

cn.enjoy.service.UserService

按服务名

Version

Group

搜索

查询结果

服务名

操作

无可用数据

创建新动态配置规则

Service class

cn.enjoy.service.UserService

Version

Group

Application Name

规则内容

```
1 configVersion: v2.7
2 enabled: true
3 configs:
4   - addresses: [0.0.0.0] # 0.0.0.0 for all addresses
5     side: consumer # effective side, consumer or addresses
6     parameters:
7       retries: 5 # dynamic config parameter
8
```

关闭

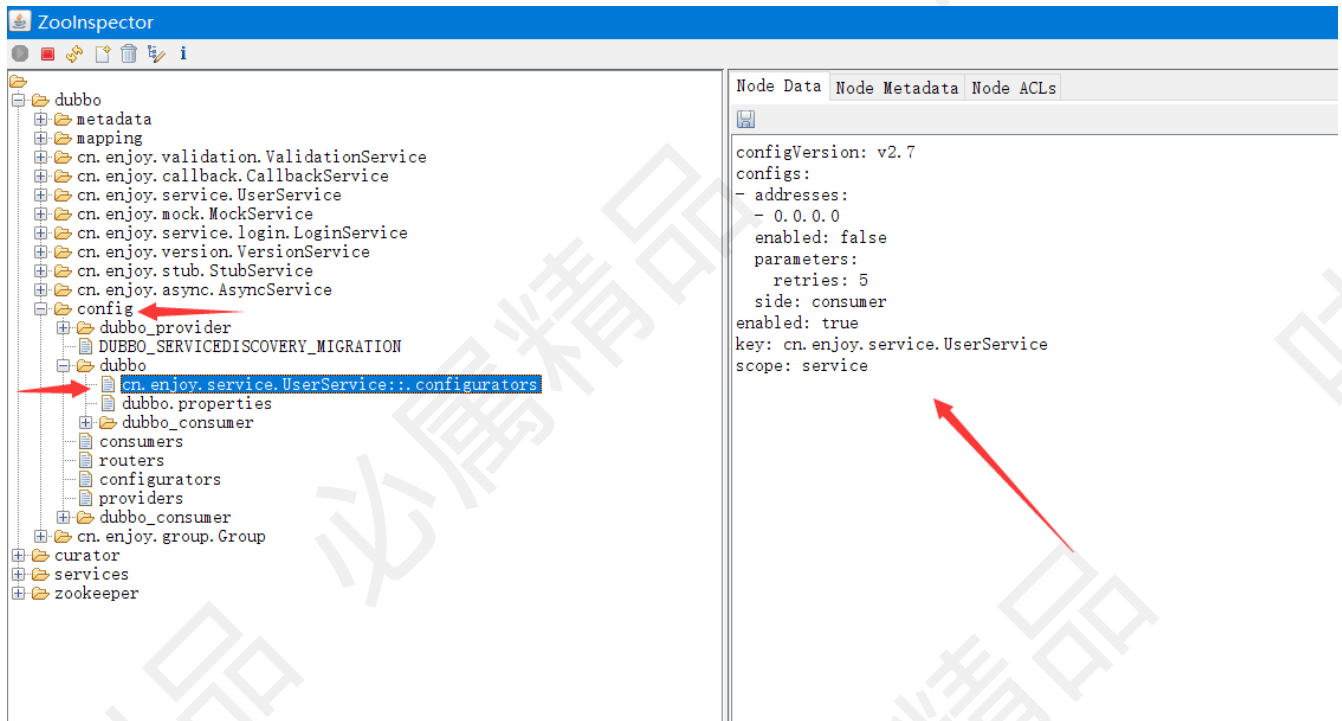
保存

通过上面的步骤就可以完成对retries属性的动态修改，dubbo-admin里面修改了属性后，我们来看看zookeeper的变化。

1、在configurators节点多了一个override协议节点



2、在分布式配置节点多了一个接口配置数据



从上面的变化来看看，我们在dubbo-admin中修改一个数据其实对应zookeeper有两个节点新增了，那么这两个节点新增会不会触发客户端的事件呢？答案是肯定的。我们接下来来看看这两个新增节点的事件注册流程。

2.1、监听注册

2.1.1、说明

之前在分析providers节点注册的时候我们就分析了客户端也会对configurators节点进行监听，这块的监听注册逻辑我就不分析了，是跟providers节点的注册监听逻辑是一模一样的，我们重点来分析一下对/dubbo/config节点下的节点进行注册的逻辑。

2.1.2、源码分析

源码还是来看看RegistryProtocol中的refer，最终会走到RegistryDirectory的subscribe方法

```
@Override
public void subscribe(URL url) {
    setSubscribeUrl(url);
    CONSUMER_CONFIGURATION_LISTENER.addNotifyListener(this);
    //订阅事件 对 config中的 xxx.xx.xx.xx::configurations
    referenceConfigurationListener = new ReferenceConfigurationListener(this, url);
    //订阅其他事件, configurations routes, providers
    registry.subscribe(url, this);
}
```

config节点的注册就在referenceConfigurationListener = new ReferenceConfigurationListener(this, url);这行代码中。

```

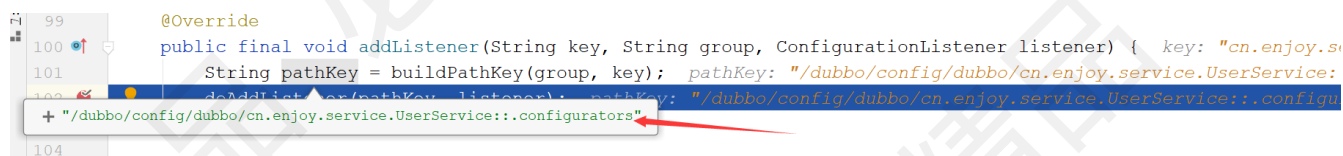
ReferenceConfigurationListener(RegistryDirectory directory, URL url) {
    this.directory = directory;
    this.url = url;
    this.initWith(DynamicConfiguration.getRuleKey(url) + CONFIGURATORS_SUFFIX);
}

```

```

protected final void initWith(String key) {
    ruleRepository.addListener(key, this);
    String rawConfig = ruleRepository.getRule(key, DynamicConfiguration.DEFAULT_GROUP);
    if (!StringUtils.isEmpty(rawConfig)) {
        genConfiguratorsFromRawRule(rawConfig);
    }
}

```



```

99      @Override
100     public final void addListener(String key, String group, ConfigurationListener listener) { key: "cn.enjoy.s
101         String pathKey = buildPathKey(group, key); pathKey: "/dubbo/config/dubbo/cn.enjoy.service.UserService:
102         doAddListener(pathKey, listener); pathKey: "/dubbo/config/dubbo/cn.enjoy.service.UserService::configu
+ "/dubbo/config/dubbo/cn.enjoy.service.UserService::configurators"
104

```

可以看到对应的路径就是zookeeper中新增的这个路径，是对这个路径进行了监听的。

```

@Override
protected void doAddListener(String pathKey, ConfigurationListener listener) {
    cacheListener.addListener(pathKey, listener);
    zkClient.addDataListener(pathKey, cacheListener, executor);
}

```

其实这里的逻辑跟之前分析的注册监听的逻辑差不多，也是有一个CacheListener，里面建立了路径和监听逻辑的映射关系，然后zookeeper的监听实现类持有了CacheListener逻辑，由zookeeper的监听实现类掉到了CacheListener中去了。

```

@Override
public void addDataListener(String path, DataListener listener, Executor executor) {
    ConcurrentMap<DataListener, TargetDataListener> dataListenerMap =
    listeners.computeIfAbsent(path, k -> new ConcurrentHashMap<>());
    TargetDataListener targetListener = dataListenerMap.computeIfAbsent(listener, k ->
    createTargetDataListener(path, k));
    addTargetDataListener(path, targetListener, executor);
}

```

这里就创建了zookeeper的监听实现类和注册了zookeeper的监听了。

监听实现类

```

@Override
protected CuratorZookeeperClient.NodeCacheListenerImpl createTargetDataListener(String
path, DataListener listener) {
    return new NodeCacheListenerImpl(client, listener, path);
}

```

注册监听

```

@Override
protected void addTargetDataListener(String path,
CuratorZookeeperClient.NodeCacheListenerImpl nodeCacheListener, Executor executor) {
    try {
        NodeCache nodeCache = new NodeCache(client, path);
        if (nodeCacheMap.putIfAbsent(path, nodeCache) != null) {
            return;
        }
        if (executor == null) {
            nodeCache.getListenable().addListener(nodeCacheListener);
        } else {
            nodeCache.getListenable().addListener(nodeCacheListener, executor);
        }

        nodeCache.start();
    } catch (Exception e) {
        throw new IllegalStateException("Add nodeCache listener for path:" + path, e);
    }
}

```

从上面的分析我们可以看出，已经注册了对分布式配置中心的某个节点的监听了。

2.2、监听触发

2.2.1、说明

前面我们分析了监听的注册，下面我们来看看监听的触发逻辑，看看监听触发后到底是干了些什么。

前面我们通过dubbo-admin修改了一个属性，已经写了数据到相应的zookeeper节点中了，那么事件一定会触发的。我们来看看触发的逻辑。下面是监听实现类

2.2.2、源码分析

```

static class NodeCacheListenerImpl implements NodeCacheListener {

    private CuratorFramework client;

    private volatile DataListener dataListener;

    private String path;

    protected NodeCacheListenerImpl() {
    }
}

```



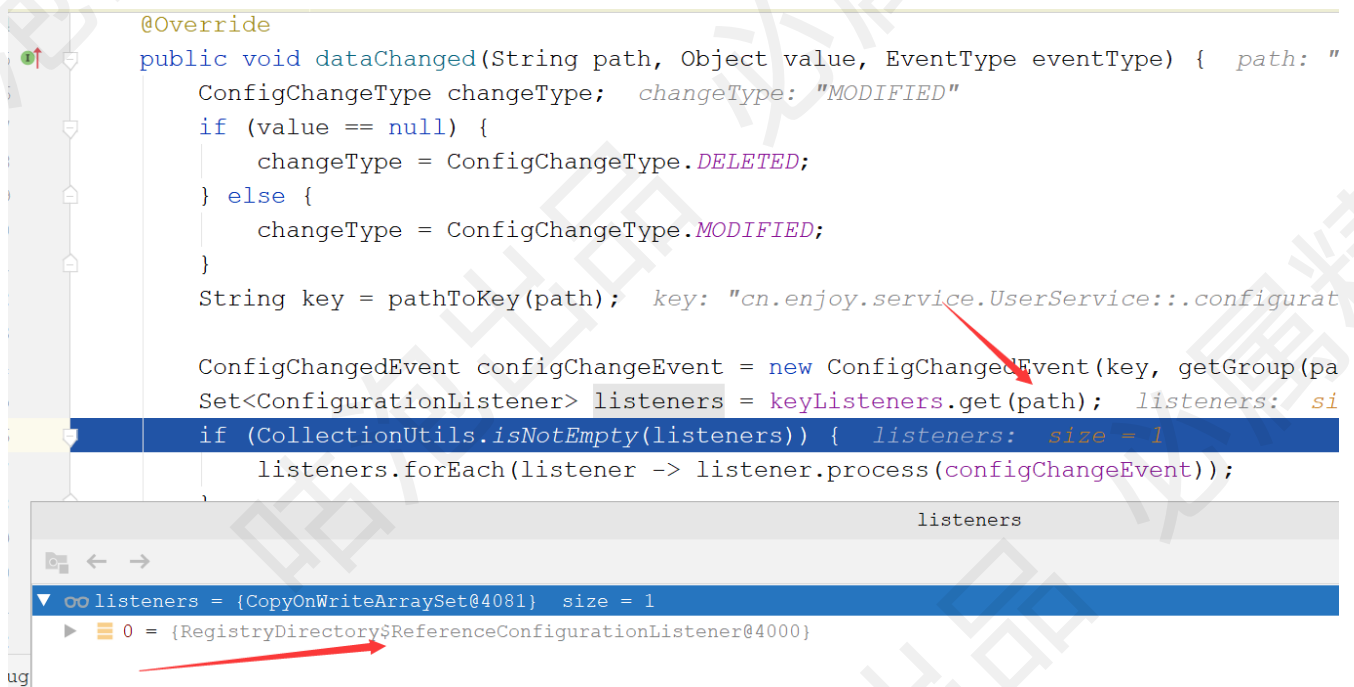
```

public NodeCacheListenerImpl(CuratorFramework client, DataListener dataListener, String
path) {
    this.client = client;
    this.dataListener = dataListener;
    this.path = path;
}

@Override
public void nodeChanged() throws Exception {
    ChildData childData = nodeCacheMap.get(path).getCurrentData();
    String content = null;
    EventType eventType;
    if (childData == null) {
        eventType = EventType.NodeDeleted;
    } else {
        content = new String(childData.getData(), CHARSET);
        eventType = EventType.NodeDataChanged;
    }
    dataListener.dataChanged(path, content, eventType);
}
}

```

当事件触发后就会掉到nodeChanged()方法。



```

@Override
public void dataChanged(String path, Object value, EventType eventType) {
    path: "
    ConfigChangeType changeType; changeType: "MODIFIED"
    if (value == null) {
        changeType = ConfigChangeType.DELETED;
    } else {
        changeType = ConfigChangeType.MODIFIED;
    }
    String key = pathToKey(path); key: "cn.enjoy.service.UserService::configurat
    ConfigChangeEvent configChangeEvent = new ConfigChangeEvent(key, getGroup(pa
    Set<ConfigurationListener> listeners = keyListeners.get(path); listeners: si
    if (CollectionUtils.isEmpty(listeners)) { listeners: size = 1
        listeners.forEach(listener -> listener.process(configChangeEvent));
}

```

listeners

listeners = {CopyOnWriteArraySet@4081} size = 1

0 = {RegistryDirectory\$ReferenceConfigurationListener@4000}

可以看到根据path，path就是新增的config节点的路径，根据path从CacheListener中后去到了一个客户端的监听类。那么就会掉到该监听类的process方法中。

```

@Override
public void dataChanged(String path, Object value, EventType eventType) {
    ConfigChangeType changeType;
    if (value == null) {
        changeType = ConfigChangeType.DELETED;
    } else {
        changeType = ConfigChangeType.MODIFIED;
    }
}

```

```

}
String key = pathToKey(path);

ConfigChangeEvent configChangeEvent = new ConfigChangeEvent(key, getGroup(path),
(String) value, changeType);
Set<ConfigurationListener> listeners = keyListeners.get(path);
if (CollectionUtils.isNotEmpty(listeners)) {
    listeners.forEach(listener -> listener.process(configChangeEvent));
}
}
}

```

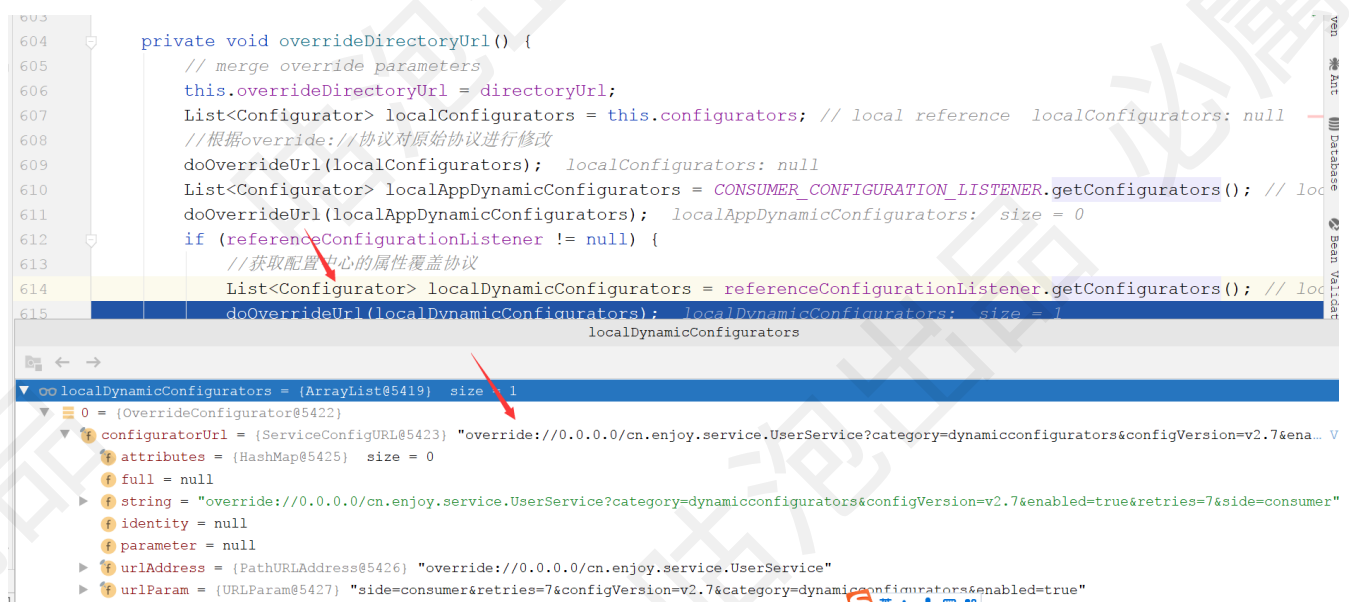
其实最终也是掉到了RegistryDirectory的refreshOverrideAndInvoker逻辑，这里我们上面分析过就是为了刷新服务列表的，那么我们看看

这里是从zookeeper的分布式配置中心中获取配置，然后根据override协议生成配置类

```

private void overrideDirectoryUrl() {
    // merge override parameters
    this.overrideDirectoryUrl = directoryUrl;
    List<Configurator> localConfigurators = this.configurators; // local reference
    //根据override://协议对原始协议进行修改
    doOverrideUrl(localConfigurators);
    List<Configurator> localAppDynamicConfigurators =
    CONSUMER_CONFIGURATION_LISTENER.getConfigurators(); // local reference
    doOverrideUrl(localAppDynamicConfigurators);
    if (referenceConfigurationListener != null) {
        //获取配置中心的属性覆盖协议
        List<Configurator> localDynamicConfigurators =
        referenceConfigurationListener.getConfigurators(); // local reference
        doOverrideUrl(localDynamicConfigurators);
    }
}

```



然后根据配置类的configure方法把原始dubbo协议然后根据override协议来修改原始的dubbo协议，比如把override协议中的retries=8替换掉之前dubbo协议中的retries=3，这就是配置类的作用，一个override协议会获取一个Configurator的实例，根据这个实例修改原始的dubbo协议。其实就是属性替换或者属性新增。

```
private void doOverrideUrl(List<Configurator> configurators) {
    if (CollectionUtils.isNotEmpty(configurators)) {
        for (Configurator configurator : configurators) {
            this.overrideDirectoryUrl = configurator.configure(overrideDirectoryUrl);
        }
    }
}
```

现在原始的dubbo协议修改了，那么接下来就是服务列表刷新了，其实刷新逻辑跟之前是一样的，也是会在toInvokers方法中去根据protocol.refer方法根据新的dubbo协议生成一个invoker对象，然后刷新invokers服务列表，这里就不贴代码了，跟之前的一样。

3、动态路由的监听

3.1、监听注册

3.1.1、说明

动态路由这块我们只分析条件动态路由，也是要根据dubbo-admin服务治理的方式去修改路由配置，然后在运行时作用到调用流程中去。我们去看看dubbo-admin中如何去修改一个动态路由。

查询结果				
服务名	组	版本	应用	操作
cn.enjoy.async.AsyncService			dubbo_provider	详情 测试 更多
cn.enjoy.callback.CallbackService			dubbo_provider	详情 测试 更多
cn.enjoy.group.Group	groupImpl1		dubbo_provider	详情 测试 更多
cn.enjoy.group.Group	groupImpl2		dubbo_provider	详情 测试 更多
cn.enjoy.mock.MockService			dubbo_provider	详情 测试 更多
cn.enjoy.service.UserService			dubbo_provider	详情 测试 更多
cn.enjoy.stub.StubService			dubbo_provider	详情 测试 更多
cn.enjoy.validation.ValidationService			dubbo_provider	详情 测试 更多
cn.enjoy.version.VersionService		1.0.0	dubbo_provider	详情 测试 更多
cn.enjoy.version.VersionService		2.0.0	dubbo_provider	详情 测试 更多
每页行数: 10 1-10 共 10 条				<div> <div>条件路由</div> <div>标签路由</div> <div>动态配置</div> <div>黑白名单</div> <div>权重调整</div> <div>负载均衡</div> </div>

条件路由

服务治理 / 条件路由

搜索路由规则

cn.enjoy.service.UserService

按服务名

Version

Group

搜索

查询结果

服务名

组

版本

开启

操作

无可用数据

创建

我们以读写分离路由为例去配置，关于动态路由配置规则，请参照dubbo官网的路由配置规则：

<https://dubbo.apache.org/zh/docs/advanced/routing-rule/>

```
method = find*,list*,get*,is* => host = 172.22.3.94,172.22.3.95,172.22.3.96
method != find*,list*,get*,is* => host = 192.168.67.1,172.22.3.98
```

创建新路由规则

Service class

cn.enjoy.service.UserService

Version

Group

Application Name

规则内容

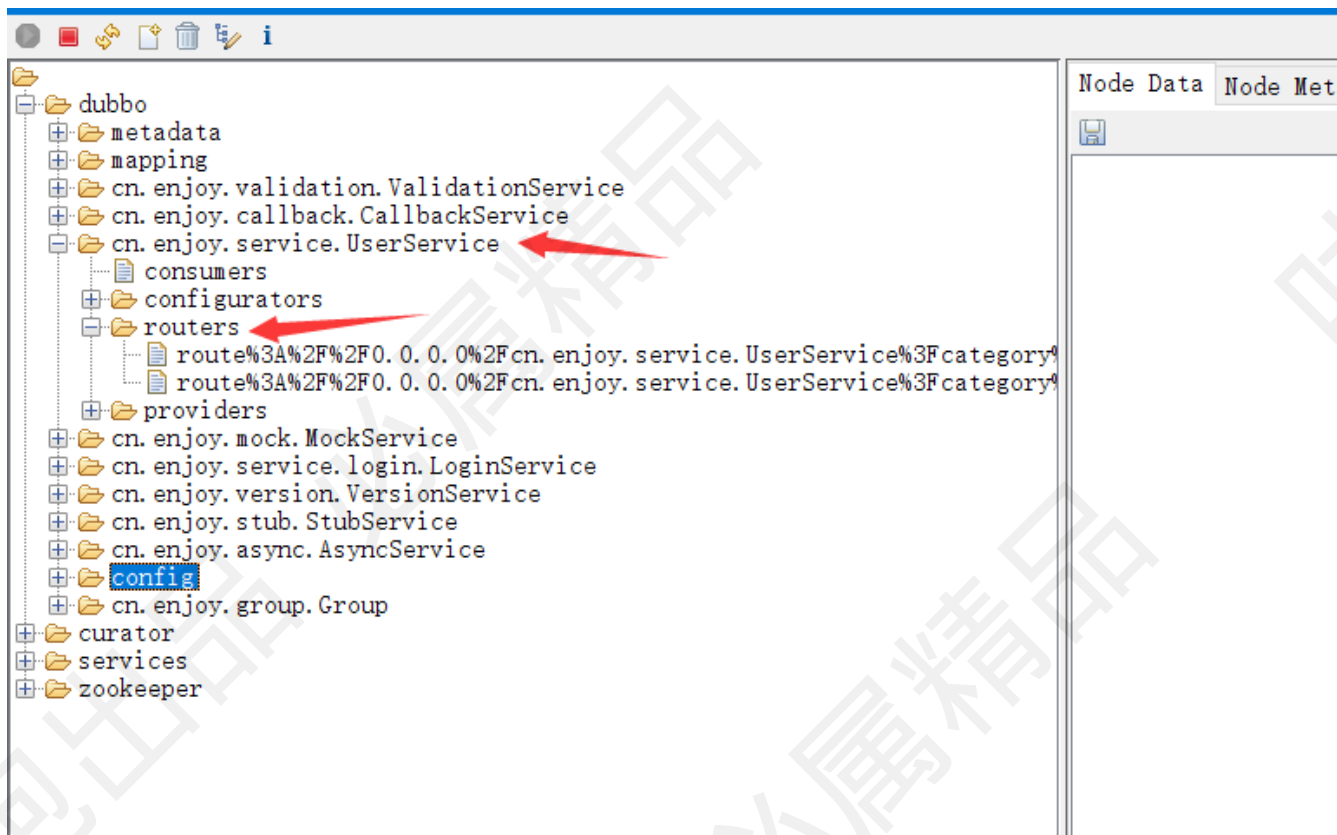
```
1 enabled: true
2 runtime: false
3 force: true
4 conditions:
5   - 'method = find*,list*,get*,is* => host = 172.22.3.94,172.22.3.95,172.22.3.96'
6   - 'method != find*,list*,get*,is* => host = 192.168.67.1,172.22.3.98'
```

关闭

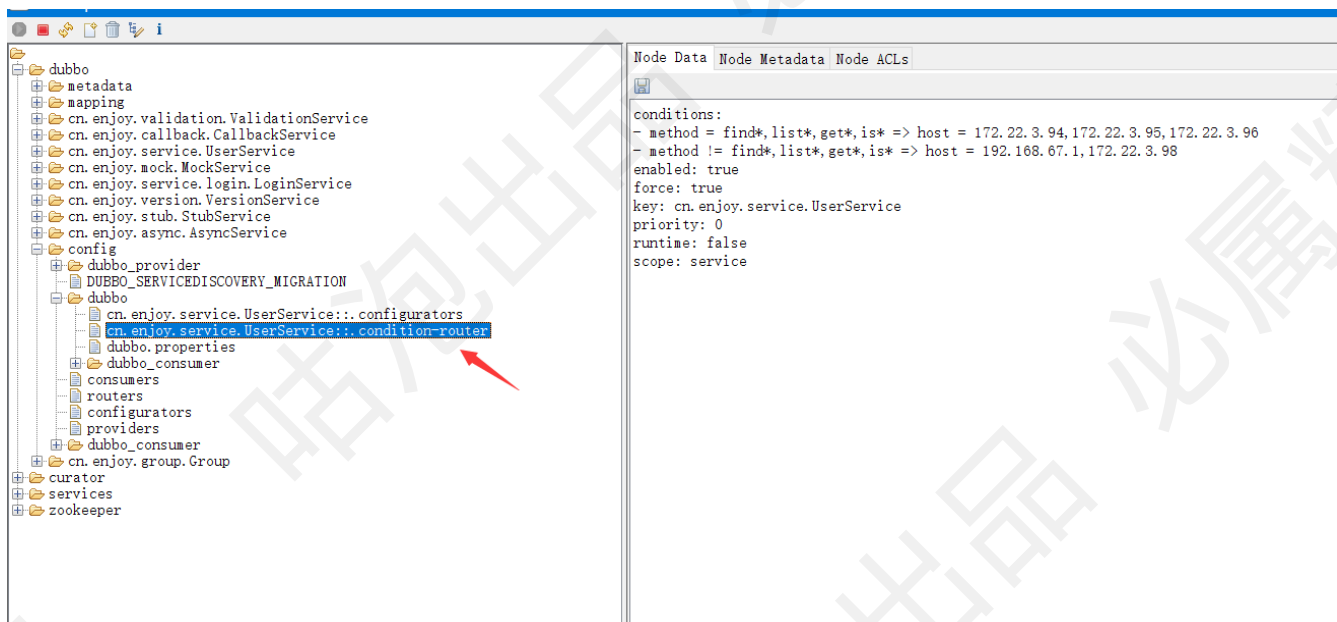
保存

保存后我们看看zookeeper的变化

1、在routers节点多了一个route协议节点



2、分布式配置config节点下新增了节点



3.1.2、源码分析

routers节点下的监听之前已经分析过，我这里就不分析了，我们主要来分析一下config节点下的监听注册逻辑。

还是在RegistryProtocol的refer作为入口，最终来到

```

protected <T> ClusterInvoker<T> doCreateInvoker(DynamicDirectory<T> directory, Cluster
cluster, Registry registry, Class<T> type) {
    directory.setRegistry(registry);
    directory.setProtocol(protocol);
    // all attributes of REFER_KEY
    Map<String, String> parameters = new HashMap<String, String>
(directory.getConsumerUrl().getParameters());
    URL urlToRegistry = new ServiceConfigURL(
        parameters.get(PROTOCOL_KEY) == null ? DUBBO : parameters.get(PROTOCOL_KEY),
        parameters.remove(REGISTER_IP_KEY), 0, getPath(parameters, type), parameters);
    if (directory.isShouldRegister()) {
        directory.setRegisteredConsumerUrl(urlToRegistry);
        //把协议注册到 /dubbo/cn.enjoy.userService/consumers节点下面
        registry.register(directory.getRegisteredConsumerUrl());
    }
    //创建路由链
    directory.buildRouterChain(urlToRegistry);
    //订阅事件, 对 configurations, providers, routes节点建立监听
    directory.subscribe(toSubscribeUrl(urlToRegistry));

    //返回默认的 FailoverClusterInvoker对象
    return (ClusterInvoker<T>) cluster.join(directory);
}

```

directory.buildRouterChain(urlToRegistry);在这行代码这里完成了路由的监听

```

private RouterChain(URL url) {
    loopPool = executorRepository.nextExecutorExecutor();
    List<RouterFactory> extensionFactories =
ExtensionLoader.getExtensionLoader(RouterFactory.class)
        .getActivateExtension(url, ROUTER_KEY);

    List<Router> routers = extensionFactories.stream()
        .map(factory -> factory.getRouter(url))
        .collect(Collectors.toList());

    initWithRouters(routers);

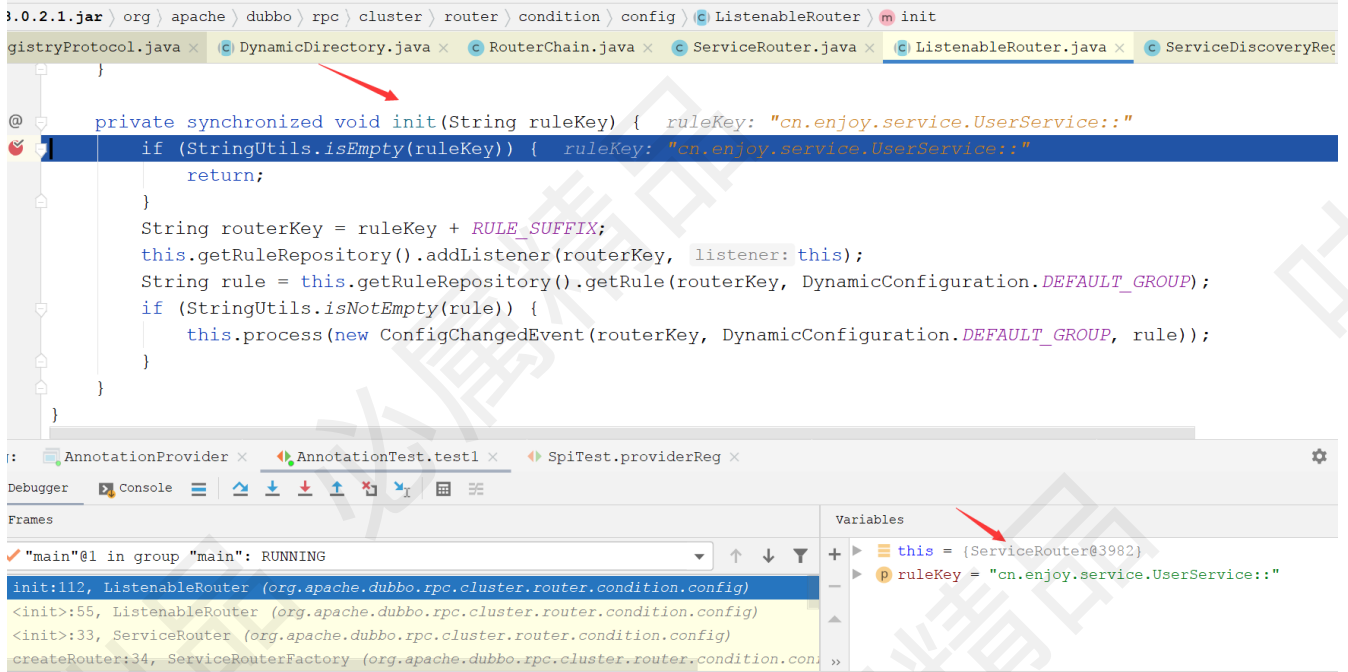
    List<StateRouterFactory> extensionStateRouterFactories =
ExtensionLoader.getExtensionLoader(
    StateRouterFactory.class)
        .getActivateExtension(url, STATE_ROUTER_KEY);

    List<StateRouter> stateRouters = extensionStateRouterFactories.stream()
        .map(factory -> factory.getRouter(url, this))
        .sorted(StateRouter::compareTo)
        .collect(Collectors.toList());

    // init state routers
    initWithStateRouters(stateRouters);
}

```

最终代码会走到



```
this.getRuleRepository().addListener(routerKey, this);
```

在这行代码这里完成了注册流程，注册跟之前是一样的就不细看了

```
default void addListener(String key, ConfigurationListener listener) {
    addListener(key, DEFAULT_GROUP, listener);
    + "cn.enjoy.service.UserService::condition-router"
}
```

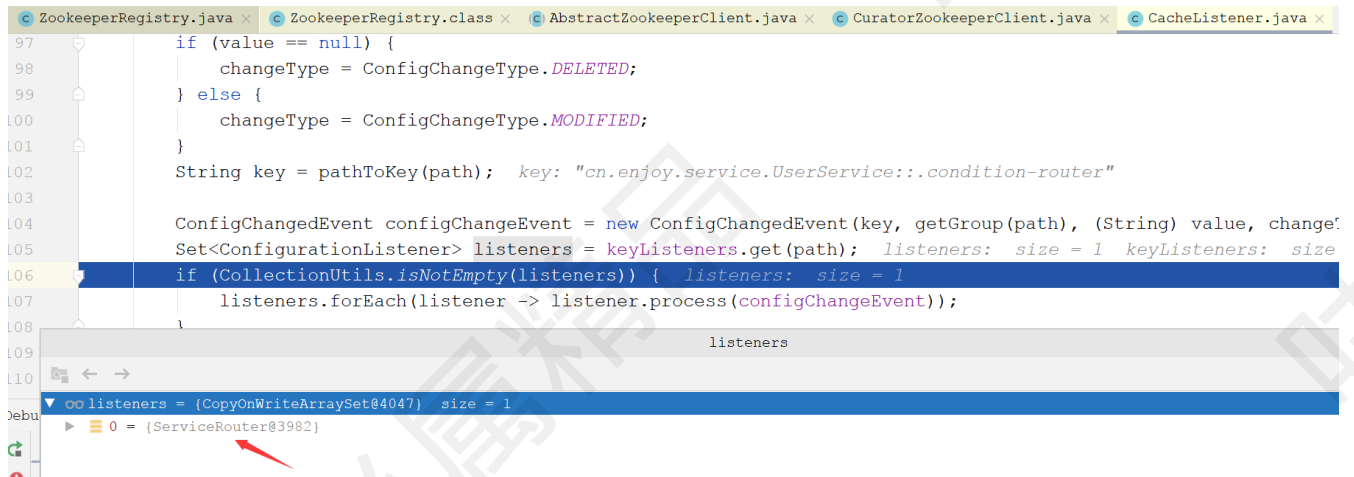
3.2、监听触发

3.2.1、说明

当我们写节点时就会触发事件

3.2.2、源码分析

当触发了事件时，一样的会走到`CacheListener`类中，然后根据`path`后去到`path`对应的监听回调类，该类就是`ServiceRouter`



The screenshot shows an IDE with several tabs: ZookeeperRegistry.java, ZookeeperRegistry.class, AbstractZookeeperClient.java, CuratorZookeeperClient.java, and CacheListener.java. The ZookeeperRegistry.java file is open, showing lines 97 to 110. The code defines a process method that handles configuration changes. A red arrow points to the debugger state, which shows a variable 'listeners' of type 'CopyOnWriteArraySet' with a size of 1. The single element in the set is a 'ServiceRouter' object with ID 3982.

```
97         if (value == null) {
98             changeType = ConfigChangeType.DELETED;
99         } else {
100             changeType = ConfigChangeType.MODIFIED;
101         }
102         String key = pathToKey(path); key: "cn.enjoy.service.UserService::condition-router"
103
104         ConfigChangedEvent configChangeEvent = new ConfigChangedEvent(key, getGroup(path), (String) value, changeType);
105         Set<ConfigurationListener> listeners = keyListeners.get(path); listeners: size = 1 keyListeners: size = 1
106         if (CollectionUtils.isEmpty(listeners)) { listeners: size = 1
107             listeners.forEach(listener -> listener.process(configChangeEvent));
108         }
109     }
110
```

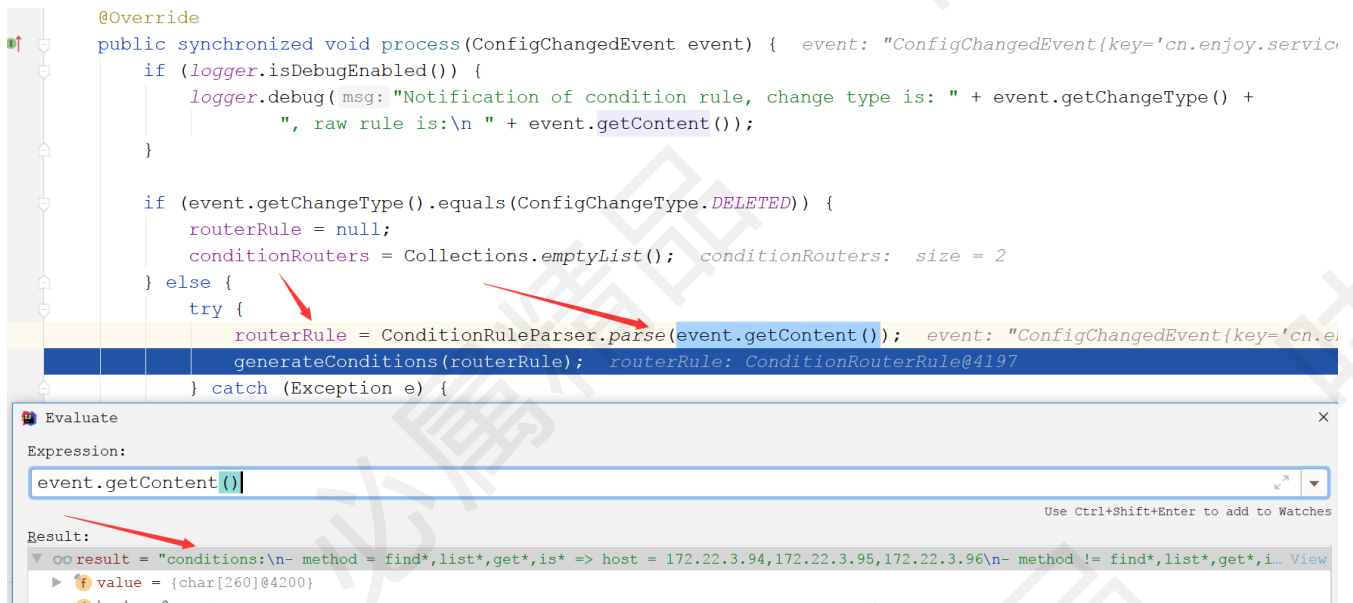
Debugger state:

```
listeners = {CopyOnWriteArraySet@4047} size = 1
  0 = {ServiceRouter@3982}
```

我们看看该类的process逻辑

```
@Override
public synchronized void process(ConfigChangedEvent event) {
    if (logger.isDebugEnabled()) {
        Logger.debug("Notification of condition rule, change type is: " +
            event.getChangeType() +
            ", raw rule is:\n " + event.getContent());
    }

    if (event.getChangeType().equals(ConfigChangeType.DELETED)) {
        routerRule = null;
        conditionRouters = Collections.emptyList();
    } else {
        try {
            routerRule = ConditionRuleParser.parse(event.getContent());
            generateConditions(routerRule);
        } catch (Exception e) {
            logger.error("Failed to parse the raw condition rule and it will not take effect, please check " +
                "if the condition rule matches with the template, the raw rule is:\n " +
                event.getContent(), e);
        }
    }
}
```

这里是解析配置规则，然后根据配置规则生成路由规则类，这里就生成了两个条件路由规则类



那么这个规则类最终是如何进行路由的呢？

其实说白了，就是在调用过程中，当我们获取到服务列表的时候，对这个服务列表进行条件路由规则的过滤，过滤剩下的invoker对象才是我们需要调用的invoker对象。

那我们来看看调用流程

在doList方法中就会根据路由规则来路由了，我们看看doList逻辑

```
@Override
public List<Invoker<T>> doList(Invocation invocation) {
    if (forbidden) {
        // 1. No service provider 2. Service providers are disabled
        throw new RpcException(RpcException.FORBIDDEN_EXCEPTION, "No provider available
from registry " +
            getUrl().getAddress() + " for service " + getConsumerUrl().getServiceKey()
+ " on consumer " +
            NetUtils.getLocalHost() + " use dubbo version " + Version.getVersion() +
            ", please check status of providers(disabled, not registered or in
blacklist).");
    }
}
```

```

if (multiGroup) {
    return this.invokers == null ? Collections.emptyList() : this.invokers;
}

List<Invoker<T>> invokers = null;
try {
    // Get invokers from cache, only runtime routers will be executed.
    invokers = routerChain.route(getConsumerUrl(), invocation);
} catch (Throwable t) {
    logger.error("Failed to execute router: " + getUrl() + ", cause: " +
t.getMessage(), t);
}

return invokers == null ? Collections.emptyList() : invokers;
}

```

invokers = routerChain.route(getConsumerUrl(), invocation);这行代码就是根据路由规则路由的。

最终会走到ServiceRouter中，根据事件触发后生成的两个条件路由规则来进行路由匹配。

```

@Override
public <T> List<Invoker<T>> route(List<Invoker<T>> invokers, URL url, Invocation invocation) throws
if (CollectionUtils.isEmpty(invokers) || conditionRouters.size() == 0) {
    return invokers; invokers: size = 1
}

// We will check enabled status inside each router.
for (Router router : conditionRouters) { conditionRouters: size = 2
    invokers = router.route(invokers, url, invocation);
}

```



```

@Override
public <T> List<Invoker<T>> route(List<Invoker<T>> invokers, URL url, Invocation
invocation)
    throws RpcException {
    if (!enabled) {
        return invokers;
    }

    if (CollectionUtils.isEmpty(invokers)) {
        return invokers;
    }
    try {
        if (!matchWhen(url, invocation)) {
            return invokers;
        }
        List<Invoker<T>> result = new ArrayList<Invoker<T>>();
        if (thenCondition == null) {

```

```

        logger.warn("The current consumer in the service blacklist. consumer: " +
NetUtils.getLocalHost() + ", service: " + url.getServiceKey());
        return result;
    }
    for (Invoker<T> invoker : invokers) {
        if (matchThen(invoker.getUrl(), url)) {
            result.add(invoker);
        }
    }
    if (!result.isEmpty()) {
        return result;
    } else if (this.isForce()) {
        logger.warn("The route result is empty and force execute. consumer: " +
NetUtils.getLocalHost() + ", service: " + url.getServiceKey() + ", router: " +
url.getParameterAndDecoded(RULE_KEY));
        return result;
    }
} catch (Throwable t) {
    logger.error("Failed to execute condition router rule: " + getUrl() + ", invokers:
" + invokers + ", cause: " + t.getMessage(), t);
}
return invokers;
}

```

路由匹配规则就是根据你的配置信息来进行一一匹配，这块逻辑同学们可以自己去看，这里就不分析了，路由还算是比较容易看懂的。