crazyhzm 265

Dubbo源码解析（五）注册中心——multicast

java dubbo 阅读约 35 分钟

注册中心——multicast

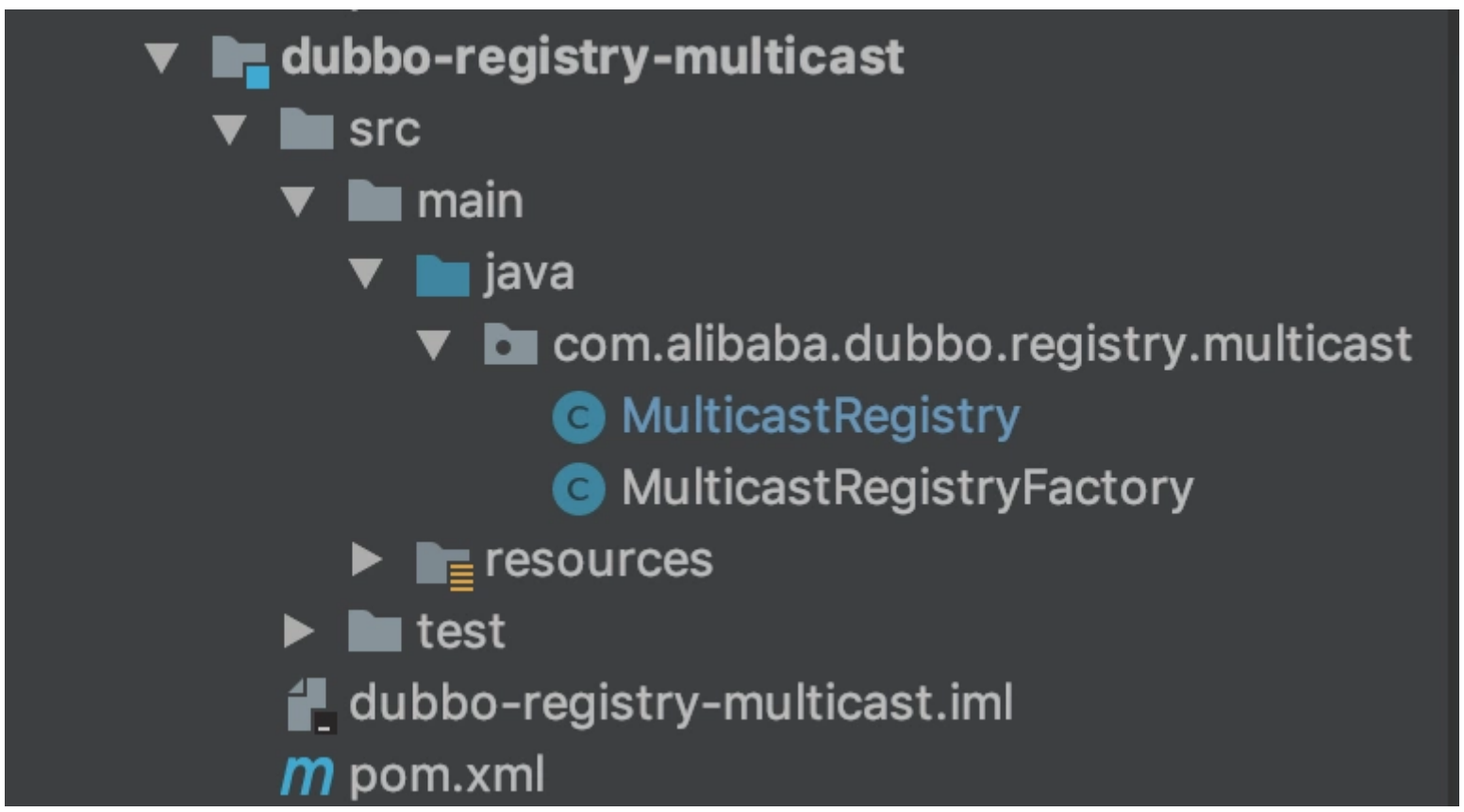
> 目标：解释以为multicast实现的注册中心原理，理解单播、广播、多播区别，解读duubo-registry-multicast的源码

这是dubbo实现注册中心的第二种方式，也是dubbo的demo模块中用的注册中心实现方式。multicast其实是用到了MulticastSocket来实现的。

我这边稍微补充一点关于多点广播，也就是MulticastSocket的介绍。MulticastSocket类是继承了DatagramSocket类，DatagramSocket只允许把数据报发送给一个指定的目标地址，而MulticastSocket可以将数据报以广播的形式发送给多个客户端。它的思想是MulticastSocket会把一个数据报发送给一个特定的多点广播地址，这个多点广播地址是一组特殊的网络地址，当客户端需要发送或者接收广播信息时，只要加入该组就好。IP协议为多点广播提供了一批特殊的IP地址，地址范围是224.0.0.0至239.255.255.255。MulticastSocket类既可以将数据报发送到多点广播地址，也可以接收其他主机的广播信息。

以上是对multicast背景的简略介绍，接下来让我们具体的来看dubbo怎么把MulticastSocket运用到注册中心的实现中。

我们先来看看包下面有哪些类：



可以看到跟默认的注册中心的包结构非常类似。接下来我们就来解读一下这两个类。

（一）MulticastRegistry

该类继承了FailbackRegistry类，该类就是针对注册中心核心的功能注册、订阅、取消注册、取消订阅，查询注册列表进行展开，利用广播的方式去实现。

1.属性

```
// 默认的多点广播端口
private static final int DEFAULT_MULTICAST_PORT = 1234;

// 多点广播的地址
private final InetAddress mutilcastAddress;

// 多点广播
private final MulticastSocket mutilcastSocket;

// 多点广播端口
private final int mutilcastPort;

//收到的URL
private final ConcurrentMap<URL, Set<URL>> received = new ConcurrentHashMap<URL, Set<URL>>();

// 任务调度器
private final ScheduledExecutorService cleanExecutor = Executors.newScheduledThreadPool(1, new
NamedThreadFactory("DubboMulticastRegistryCleanTimer", true));

// 定时清理执行器，一定时间清理过期的url
private final ScheduledFuture<?> cleanFuture;

// 清理的间隔时间
private final int cleanPeriod;

// 管理员权限
private volatile boolean admin = false;
```

看上面的属性，需要关注以下几个点：

1. mutilcastSocket，该类是muticast注册中心实现的关键，这里补充一下单播、广播、以及多播的区别，因为下面会涉及到。单播是每次只有两个实体相互通信，发送端和接收端都是唯一确定的；广播目的地址为网络中的全体目标，而多播的目的地址是一组目标，加入该组的成员均是数据包的目的地。
2. 关注任务调度器和清理计时器，该类封装了定时清理过期的服务的策略。

2.构造方法

```
public MulticastRegistry(URL url) {
    super(url);
    if (url.isAnyHost()) {
        throw new IllegalStateException("registry address == null");
    }
    if (!isMulticastAddress(url.getHost())) {
        throw new IllegalArgumentException("Invalid multicast address " + url.getHost() + ", scope: 224.0.0.0 -
239.255.255.255");
    }
    try {
        mutilcastAddress = InetAddress.getByName(url.getHost());
        // 如果url携带的配置中没有端口号，则使用默认端口号
        mutilcastPort = url.getPort() <= 0 ? DEFAULT_MULTICAST_PORT : url.getPort();
        mutilcastSocket = new MulticastSocket(mutilcastPort);
        // 禁用多播数据报的本地环回
        mutilcastSocket.setLoopbackMode(false);
        // 加入同一组广播
        mutilcastSocket.joinGroup(mutilcastAddress);
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                byte[] buf = new byte[2048];
                // 实例化数据报
                DatagramPacket recv = new DatagramPacket(buf, buf.length);
                while (!mutilcastSocket.isClosed()) {
                    try {
```

这个构造器最关键的就是一个线程和一个定时清理任务。

1. 线程中做的工作是根据接收到的消息来判定是什么请求，作出对应的操作，只要mutilcastSocket没有断开，就一直接收消息，内部的实现体现在receive方法中，下文会展开讲述。

2. 定时清理任务是清理过期的注册的服务。通过两次socket的尝试来判定是否过期。clean方法下文会展开讲述

3.isMulticastAddress

```
private static boolean isMulticastAddress(String ip) {
    int i = ip.indexOf('.');
    if (i > 0) {
        String prefix = ip.substring(0, i);
        if (StringUtils.isInteger(prefix)) {
            int p = Integer.parseInt(prefix);
            return p >= 224 && p <= 239;
        }
    }
    return false;
}
```

该方法很简单，为也没写注释，就是判断是否为多点广播地址，地址范围是224.0.0.0至239.255.255.255。

4.clean

```
private void clean() {
    // 当url中携带的服务接口配置为是*时候，才可以执行清理
    if (admin) {
        for (Set<URL> providers : new HashSet<Set<URL>>(received.values())) {
            for (URL url : new HashSet<URL>(providers)) {
                // 判断是否过期
                if (isExpired(url)) {
                    if (logger.isWarnEnabled()) {
                        logger.warn("Clean expired provider " + url);
                    }
                    //取消注册
                    doUnregister(url);
                }
            }
        }
    }
}
```

该方法也比较简单，关机的是如何判断过期以及做的取消注册的操作。下面会展开讲解这几个方法。

5.isExpired

```
private boolean isExpired(URL url) {
    // 如果为非动态管理模式或者协议是consumer、route或者override，则没有过期
    if (!url.getParameter(Constants.DYNAMIC_KEY, true)
            || url.getPort() <= 0
            || Constants.CONSUMER_PROTOCOL.equals(url.getProtocol())
            || Constants.ROUTE_PROTOCOL.equals(url.getProtocol())
            || Constants.OVERRIDE_PROTOCOL.equals(url.getProtocol())) {
        return false;
    }
    Socket socket = null;
    try {
        // 利用url携带的主机地址和端口号实例化socket
        socket = new Socket(url.getHost(), url.getPort());
    } catch (Throwable e) {
        // 如果实例化失败，等待100ms重试第二次，如果还失败，则判定已过期
        try {
            // 等待100ms
            Thread.sleep(100);
        } catch (Throwable e2) {
        }
        Socket socket2 = null;
        try {
            socket2 = new Socket(url.getHost(), url.getPort());
        } catch (Throwable e2) {
            return true;
        } finally {
```

这个方法就是判断服务是否过期，有两次尝试socket的操作，如果尝试失败，则判断为过期。

6.receive

```
private void receive(String msg, InetSocketAddress remoteAddress) {
    if (logger.isInfoEnabled()) {
        logger.info("Receive multicast message: " + msg + " from " + remoteAddress);
    }
    // 如果这个消息是以register、unregister、subscribe开头的，则进行相应的操作
    if (msg.startsWith(Constants.REGISTER)) {
        URL url = URL.valueOf(msg.substring(Constants.REGISTER.length()).trim());
        // 注册服务
        registered(url);
    } else if (msg.startsWith(Constants.UNREGISTER)) {
        URL url = URL.valueOf(msg.substring(Constants.UNREGISTER.length()).trim());
        // 取消注册服务
        unregistered(url);
    } else if (msg.startsWith(Constants.SUBSCRIBE)) {
        URL url = URL.valueOf(msg.substring(Constants.SUBSCRIBE.length()).trim());
        // 获得以及注册的url集合
        Set<URL> urls = getRegistered();
        if (urls != null && !urls.isEmpty()) {
            for (URL u : urls) {
                // 判断是否合法
                if (UrlUtils.isMatch(url, u)) {
                    String host = remoteAddress != null && remoteAddress.getAddress() != null
                            ? remoteAddress.getAddress().getHostAddress() : url.getIp();
                    // 建议服务提供者和服务消费者在不同机器上运行，如果在同一机器上，需设置unicast=false
                    // 同一台机器中的多个进程不能单播单播，或者只有一个进程接收信息，发给消费者的单播消息可能被提供者抢
占，两个消费者在同一台机器也一样，
```

可以很清楚的看到，根据接收到的消息开头的数据来判断需要做什么类型的操作，重点在于订阅，可以选择单播订阅还是广播订阅，这个取决于url携带的配置是什么。

7.broadcast

```
private void broadcast(String msg) {
    if (logger.isInfoEnabled()) {
        logger.info("Send broadcast message: " + msg + " to " + mutilcastAddress + ":" + mutilcastPort);
    }
    try {
        byte[] data = (msg + "\n").getBytes();
        // 实例化数据报,重点是目的地址是mutilcastAddress
        DatagramPacket hi = new DatagramPacket(data, data.length, mutilcastAddress, mutilcastPort);
        // 发送数据报
        mutilcastSocket.send(hi);
    } catch (Exception e) {
        throw new IllegalStateException(e.getMessage(), e);
    }
}
```

这是广播的实现方法，重点是数据报的目的地址是mutilcastAddress。代表着一组地址

8.unicast

```
private void unicast(String msg, String host) {
    if (logger.isInfoEnabled()) {
        logger.info("Send unicast message: " + msg + " to " + host + ":" + mutilcastPort);
    }
    try {
        byte[] data = (msg + "\n").getBytes();
        // 实例化数据报,重点是目的地址是只是单个地址
        DatagramPacket hi = new DatagramPacket(data, data.length, InetAddress.getByName(host), mutilcastPort);
        // 发送数据报
        mutilcastSocket.send(hi);
    } catch (Exception e) {
        throw new IllegalStateException(e.getMessage(), e);
    }
}
```

这是单播的实现，跟广播的区别就只是目的地址不一样，单播的目的地址就只是一个地址，而广播的是一组地址。

9.doRegister && doUnregister && doSubscribe && doUnsubscribe

```
@Override
protected void doRegister(URL url) {
    broadcast(Constants.REGISTER + " " + url.toFullString());
}
@Override
protected void doUnregister(URL url) {
    broadcast(Constants.UNREGISTER + " " + url.toFullString());
}
@Override
protected void doSubscribe(URL url, NotifyListener listener) {
    // 当url中携带的服务接口配置为是*时候，才可以执行清理，类似管理员权限
    if (Constants.ANY_VALUE.equals(url.getServiceInterface())) {
        admin = true;
    }
    broadcast(Constants.SUBSCRIBE + " " + url.toFullString());
    // 对监听器进行同步锁
    synchronized (listener) {
        try {
            listener.wait(url.getParameter(Constants.TIMEOUT_KEY, Constants.DEFAULT_TIMEOUT));
        } catch (InterruptedException e) {
        }
    }
}
@Override
protected void doUnsubscribe(URL url, NotifyListener listener) {
    if (!Constants.ANY_VALUE.equals(url.getServiceInterface())
```

这几个方法就是实现了父类FailbackRegistry的抽象方法。都是调用了broadcast方法。

10.destroy

```
@Override
public void destroy() {
    super.destroy();
    try {
        // 取消清理任务
        if (cleanFuture != null) {
            cleanFuture.cancel(true);
        }
    } catch (Throwable t) {
        logger.warn(t.getMessage(), t);
    }
    try {
        // 把该地址从组内移除
        mutilcastSocket.leaveGroup(mutilcastAddress);
        // 关闭mutilcastSocket
        mutilcastSocket.close();
    } catch (Throwable t) {
        logger.warn(t.getMessage(), t);
    }
    // 关闭线程池
    ExecutorUtil.gracefulShutdown(cleanExecutor, cleanPeriod);
}
```

该方法的逻辑跟dubbo注册中心的destroy方法类似，就多了把该地址从组内移除的操作。gracefulShutdown方法我在[《dubbo源码解析（四）注册中心——dubbo》]中已经讲到。

11.register

```
@Override
public void register(URL url) {
    super.register(url);
    registered(url);
}
```

```
protected void registered(URL url) {
    // 遍历订阅的监听器集合
    for (Map.Entry<URL, Set<NotifyListener>> entry : getSubscribed().entrySet()) {
        URL key = entry.getKey();
        // 判断是否合法
        if (UrlUtils.isMatch(key, url)) {
            // 通过消费者url获得接收到的服务url集合
            Set<URL> urls = received.get(key);
            if (urls == null) {
                received.putIfAbsent(key, new ConcurrentHashSet<URL>());
                urls = received.get(key);
            }
            // 加入服务url
            urls.add(url);
            List<URL> list = toList(urls);
            for (NotifyListener listener : entry.getValue()) {
                // 把服务url的变化通知监听器
                notify(key, listener, list);
                synchronized (listener) {
                    listener.notify();
                }
            }
        }
    }
}
```

可以看到该类重写了父类的register方法，不过逻辑没有过多的变化，就是把需要注册的url放入缓存中，如果通知监听器url的变化。

12.unregister

```
@Override
public void unregister(URL url) {
    super.unregister(url);
    unregistered(url);
}
```

```
protected void unregistered(URL url) {
    // 遍历订阅的监听器集合
    for (Map.Entry<URL, Set<NotifyListener>> entry : getSubscribed().entrySet()) {
        URL key = entry.getKey();
        if (UrlUtils.isMatch(key, url)) {
            Set<URL> urls = received.get(key);
            // 缓存中移除
            if (urls != null) {
                urls.remove(url);
            }
            if (urls == null || urls.isEmpty()){
                if (urls == null){
                    urls = new ConcurrentHashSet<URL>();
                }
                // 设置携带empty协议的url
                URL empty = url.setProtocol(Constants.EMPTY_PROTOCOL);
                urls.add(empty);
            }
            List<URL> list = toList(urls);
            // 通知监听器 服务url变化
            for (NotifyListener listener : entry.getValue()) {
                notify(key, listener, list);
            }
        }
    }
}
```

这个逻辑也比较清晰，把需要取消注册的服务url从缓存中移除，然后如果没有接收的服务url了，就加入一个携带empty协议的url，然后通知监听器服务变化。

13.lookup

```
@Override
public List<URL> lookup(URL url) {
    List<URL> urls = new ArrayList<URL>();
    // 通过消费者url获得订阅的服务的监听器
    Map<String, List<URL>> notifiedUrls = getNotified().get(url);
    // 获得注册的服务url集合
    if (notifiedUrls != null && notifiedUrls.size() > 0) {
        for (List<URL> values : notifiedUrls.values()) {
            urls.addAll(values);
        }
    }
    // 如果为空，则从内存缓存properties获得相关value，并且返回为注册的服务
    if (urls.isEmpty()) {
        List<URL> cacheUrls = getCacheUrls(url);
        if (cacheUrls != null && !cacheUrls.isEmpty()) {
            urls.addAll(cacheUrls);
        }
    }
    // 如果还是为空则从缓存registered中获得已注册 服务URL 集合
    if (urls.isEmpty()) {
        for (URL u : getRegistered()) {
            if (UrlUtils.isMatch(url, u)) {
                urls.add(u);
            }
        }
    }
```

该方法是返回注册的服务url列表，可以看到有很多种获得的方法这些缓存都保存在AbstractRegistry类中，相关的介绍可以查看《dubbo源码解析（三）注册中心——开篇》。

14.subscribe && unsubscribe

```
@Override
public void subscribe(URL url, NotifyListener listener) {
    super.subscribe(url, listener);
    subscribed(url, listener);
}

@Override
public void unsubscribe(URL url, NotifyListener listener) {
    super.unsubscribe(url, listener);
    received.remove(url);
}
```

```
protected void subscribed(URL url, NotifyListener listener) {
    // 查询注册列表
    List<URL> urls = lookup(url);
    // 通知url
    notify(url, listener, urls);
}
```

这两个重写了父类的方法，分别是订阅和取消订阅。逻辑很简单。

（二）MulticastRegistryFactory

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

```
public class MulticastRegistryFactory extends AbstractRegistryFactory {

    @Override
    public Registry createRegistry(URL url) {
        return new MulticastRegistry(url);
    }

}
```

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

后记

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

该文章讲解了dubbo利用multicast来实现注册中心，其中关键的是需要弄明白MulticastSocket以及单播、广播、多播的概念，其他的逻辑并不复杂。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。

crazyhzm
265

关注作者

0 条评论　得票 • 时间

撰写评论 ...