

Dubbo源码解析（十八）远程通信——Zookeeper

 java  zookeeper  dubbo 阅读约 24 分钟

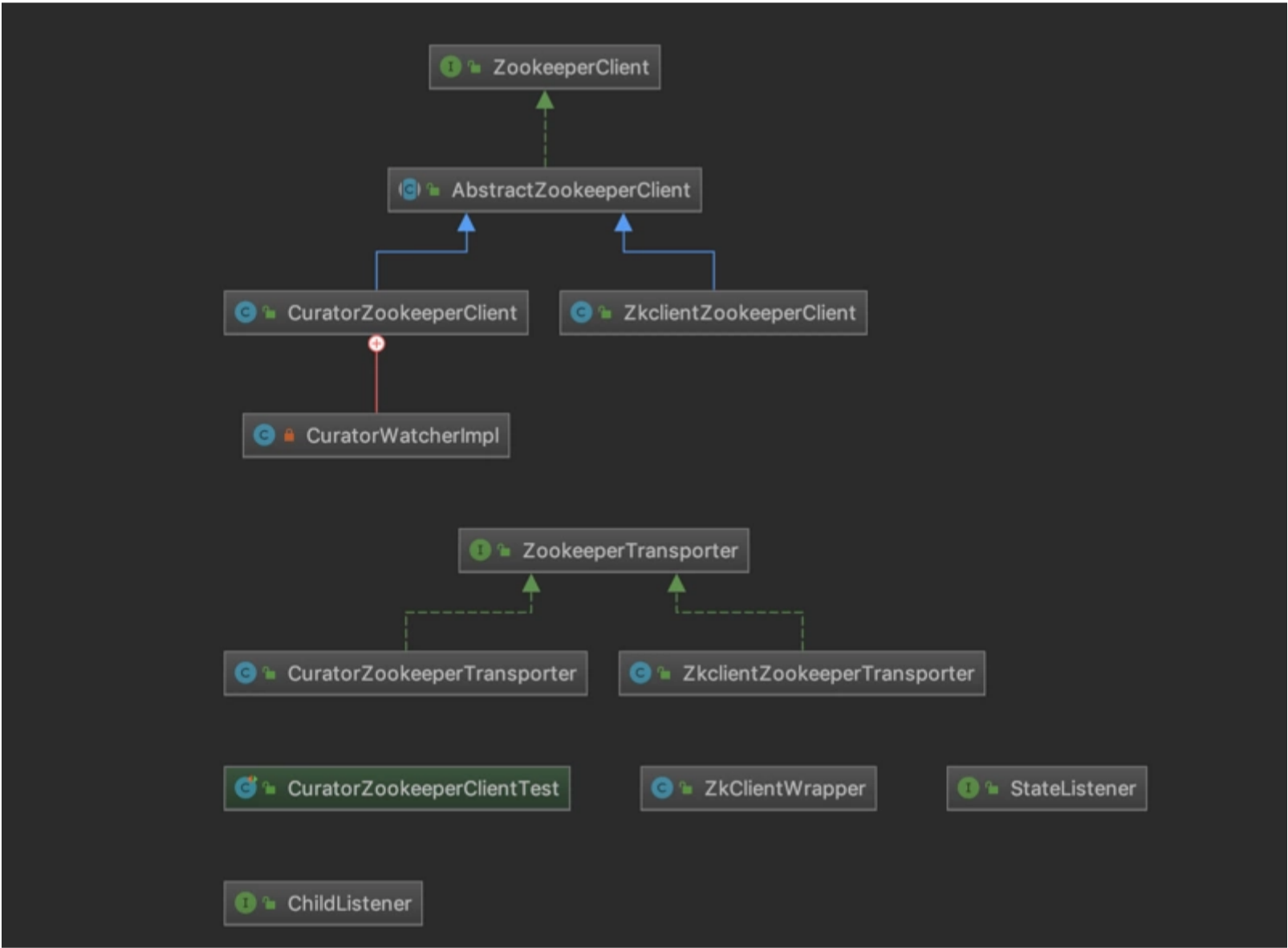
远程通讯——Zookeeper

目标：介绍基于zookeeper的来实现的远程通信、介绍dubbo-remoting-zookeeper内的源码解析。

前言

对于zookeeper我相信肯定不陌生，在之前的文章里面也有讲到zookeeper来作为注册中心。在这里，基于zookeeper来实现远程通讯，duubo封装了zookeeper client，来和zookeeper server通讯。

下面是类图：



源码分析

（一）ZookeeperClient

```
public interface ZookeeperClient {

    /**
     * 创建client
     * @param path
     * @param ephemeral
     */
    void create(String path, boolean ephemeral);

    /**
     * 删除client
     * @param path
     */
    void delete(String path);

    /**
     * 获得子节点集合
     * @param path
     * @return
     */
    List<String> getChildren(String path);

    /**
     * 向zookeeper的该节点发起订阅，获得该节点所有
     * @param path
     * @param listener
     */
}
```

该接口是基于zookeeper的客户端接口，其中封装了客户端的一些方法。

(二) AbstractZookeeperClient

该类实现了ZookeeperClient接口，是客户端的抽象类，它实现了一些公共逻辑，把具体的doClose、createPersistent等方法抽象出来，留给子类来实现。

1.属性

```
/**
 * url 对象
 */
private final URL url;

/**
 * 状态监听器集合
 */
private final Set<StateListener> stateListeners = new CopyOnWriteArraySet<StateListener>();

/**
 * 客户端监听器集合
 */
private final ConcurrentMap<String, ConcurrentMap<ChildListener, TargetChildListener>> childListeners = new
ConcurrentHashMap<String, ConcurrentMap<ChildListener, TargetChildListener>>();

/**
 * 是否关闭
 */
private volatile boolean closed = false;
```

2.create

```

@Override
public void create(String path, boolean ephemeral) {
    // 如果不是临时节点
    if (!ephemeral) {
        // 判断该客户端是否存在
        if (checkExists(path)) {
            return;
        }
    }
    // 获得/的位置
    int i = path.lastIndexOf('/');
    if (i > 0) {
        // 创建客户端
        create(path.substring(0, i), false);
    }
    // 如果是临时节点
    if (ephemeral) {
        // 创建临时节点
        createEphemeral(path);
    } else {
        // 递归创建节点
        createPersistent(path);
    }
}
}

```

该方法是创建客户端的方法，其中createEphemeral和createPersistent方法都被抽象出来。具体看下面的类的介绍。

3.addStateListener

```

@Override
public void addStatelister(StateListener listener) {
    // 状态监听器加入集合
    statelisters.add(listener);
}

```

该方法就是增加状态监听器。

4.close

```

@Override
public void close() {
    if (closed) {
        return;
    }
    closed = true;
    try {
        // 关闭
        doClose();
    } catch (Throwable t) {
        logger.warn(t.getMessage(), t);
    }
}
}

```

该方法是关闭客户端，其中doClose方法也被抽象出。

```
/**
 * 关闭客户端
 */
protected abstract void doClose();

/**
 * 递归创建节点
 * @param path
 */
protected abstract void createPersistent(String path);

/**
 * 创建临时节点
 * @param path
 */
protected abstract void createEphemeral(String path);

/**
 * 检测该节点是否存在
 * @param path
 * @return
 */
protected abstract boolean checkExists(String path);

/**
 * 创建子节点监听器
```

上述的方法都是被抽象的，又它的两个子类来实现。

（三）ZkclientZookeeperClient

该类继承了AbstractZookeeperClient，是zk客户端的实现类。

1.属性

```
/**
 * zk 客户端包装类
 */
private final ZkClientWrapper client;

/**
 * 连接状态
 */
private volatile KeeperState state = KeeperState.SyncConnected;
```

该类有两个属性，其中client就是核心所在，几乎所有方法都调用了client的方法。

2.构造函数

```
public ZkclientZookeeperClient(URL url) {
    super(url);
    // 新建一个zkclient包装类
    client = new ZkClientWrapper(url.getBackupAddress(), 30000);
    // 增加状态监听
    client.addListener(new IZkStateListener() {
        /**
         * 如果状态改变
         * @param state
         * @throws Exception
         */
        @Override
        public void handleStateChanged(KeeperState state) throws Exception {
            ZkclientZookeeperClient.this.state = state;
            // 如果状态变为了断开连接
            if (state == KeeperState.Disconnected) {
                // 则修改状态
                stateChanged(StateListener.DISCONNECTED);
            } else if (state == KeeperState.SyncConnected) {
                stateChanged(StateListener.CONNECTED);
            }
        }

        @Override
        public void handleNewSession() throws Exception {
            // 状态变为重连
        }
    });
}
```

该方法是构造方法，同时在里面也做了创建客户端和启动客户端的操作。其他方法都是实现了父类抽象的方法，并且调用的是client方法，为举个例子：

```
@Override
public void createPersistent(String path) {
    try {
        // 递归创建节点
        client.createPersistent(path);
    } catch (ZkNodeExistsException e) {
    }
}
```

该方法是递归场景节点，调用的就是client.createPersistent(path)。

（四）CuratorZookeeperClient

该类是Curator框架提供的一套高级API，简化了ZooKeeper的操作，从而对客户端的实现。

1.属性

```
/**
 * 框架式客户端
 */
private final CuratorFramework client;
```

2.构造方法

```
public CuratorZookeeperClient(URL url) {
    super(url);
    try {
        // 工厂创建者
        CuratorFrameworkFactory.Builder builder = CuratorFrameworkFactory.builder()
            .connectString(url.getBackupAddress())
            .retryPolicy(new RetryNTimes(1, 1000))
            .connectionTimeoutMs(5000);

        String authority = url.getAuthority();
        if (authority != null && authority.length() > 0) {
            builder = builder.authorization("digest", authority.getBytes());
        }
        // 创建客户端
        client = builder.build();
        // 添加监听器
        client.getConnectionStateListenable().addListener(new ConnectionStateListener() {
            @Override
            public void stateChanged(CuratorFramework client, ConnectionState state) {
                // 如果为状态为Lost，则改变为未连接
                if (state == ConnectionState.LOST) {
                    CuratorZookeeperClient.this.stateChanged(StateListener.DISCONNECTED);
                } else if (state == ConnectionState.CONNECTED) {
                    // 改变状态为连接
                    CuratorZookeeperClient.this.stateChanged(StateListener.CONNECTED);
                } else if (state == ConnectionState.RECONNECTED) {
                    // 改变状态为未连接

```

该方法是构造方法，同样里面也包含了客户端创建和启动的逻辑。

其他的方法也一样是实现了父类的抽象方法，举个例子：

```
@Override
public void createPersistent(String path) {
    try {
        client.create().forPath(path);
    } catch (NodeExistsException e) {
    } catch (Exception e) {
        throw new IllegalStateException(e.getMessage(), e);
    }
}
```

（五）ZookeeperTransporter

```
@SPI("curator")
public interface ZookeeperTransporter {

    /**
     * 连接服务器
     * @param url
     * @return
     */
    @Adaptive({Constants.CLIENT_KEY, Constants.TRANSPORTER_KEY})
    ZookeeperClient connect(URL url);

}
```

该方法是zookeeper的信息交换接口。同样也是一个可扩展接口，默认实现CuratorZookeeperTransporter类。

（六）ZkclientZookeeperTransporter

```
public class ZkclientZookeeperTransporter implements ZookeeperTransporter {

    @Override
    public ZookeeperClient connect(URL url) {
        // 新建ZkclientZookeeperClient实例
        return new ZkclientZookeeperClient(url);
    }

}
```

该类实现了ZookeeperTransporter，其中就是创建了ZkclientZookeeperClient实例。

(七) CuratorZookeeperTransporter

```
public class CuratorZookeeperTransporter implements ZookeeperTransporter {

    @Override
    public ZookeeperClient connect(URL url) {
        // 创建CuratorZookeeperClient实例
        return new CuratorZookeeperClient(url);
    }

}
```

该接口实现了ZookeeperTransporter，是ZookeeperTransporter默认的实现类，同样也是创建了；对应的CuratorZookeeperClient实例。

(八) ZkClientWrapper

该类是zk客户端的包装类。

1.属性

```
/**
 * 超时事件
 */
private long timeout;
/**
 * zk客户端
 */
private ZkClient client;
/**
 * 客户端状态
 */
private volatile KeeperState state;
/**
 * 客户端线程
 */
private ListenableFutureTask<ZkClient> listenableFutureTask;
/**
 * 是否开始
 */
private volatile boolean started = false;
```

2.构造方法

```
public ZkClientWrapper(final String serverAddr, long timeout) {
    this.timeout = timeout;
    listenableFutureTask = ListenableFutureTask.create(new Callable<ZkClient>() {
        @Override
        public ZkClient call() throws Exception {
            // 创建zk客户端
            return new ZkClient(serverAddr, Integer.MAX_VALUE);
        }
    });
}
```

设置了超时时间和客户端线程。

3.start

```
public void start() {
    // 如果客户端没有开启
    if (!started) {
        // 创建连接线程
        Thread connectThread = new Thread(listenableFutureTask);
        connectThread.setName("DubboZkclientConnector");
        connectThread.setDaemon(true);
        // 开启线程
        connectThread.start();
        try {
            // 获得zk客户端
            client = listenableFutureTask.get(timeout, TimeUnit.MILLISECONDS);
        } catch (Throwable t) {
            logger.error("Timeout! zookeeper server can not be connected in : " + timeout + "ms!", t);
        }
        started = true;
    } else {
        logger.warn("Zkclient has already been started!");
    }
}
```

该方法是客户端启动方法。

4.addListener

```
public void addListener(final IZkStateListener listener) {
    // 增加监听器
    listenableFutureTask.addListener(new Runnable() {
        @Override
        public void run() {
            try {
                client = listenableFutureTask.get();
                // 增加监听器
                client.subscribeStateChanges(listener);
            } catch (InterruptedException e) {
                logger.warn(Thread.currentThread().getName() + " was interrupted unexpectedly, which may cause unpredictable exception!");
            } catch (ExecutionException e) {
                logger.error("Got an exception when trying to create zkclient instance, can not connect to zookeeper server, please check!", e);
            }
        }
    });
}
```

该方法是为客户端添加监听器。

其他方法都是对于 客户端是否还连接的检测，可自行查看代码。

(九) ChildListener

```
public interface ChildListener {

    /**
     * 子节点修改
     * @param path
     * @param children
     */
    void childChanged(String path, List<String> children);

}
```

该接口是子节点的监听器，当子节点变化的时候会用到。

(十) StateListener

```
public interface StateListener {

    int DISCONNECTED = 0;

    int CONNECTED = 1;

    int RECONNECTED = 2;

    /**
     * 状态修改
     * @param connected
     */
    void stateChanged(int connected);

}
```

该接口是状态监听器，其中定义了一个状态更改的方法以及三种状态。

后记

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

该文章讲解了基于zookeeper的来实现的远程通信、介绍dubbo-remoting-zookeeper内的源码解析，关键需要对zookeeper有所了解。该篇之后，远程通讯的源码解析就先到这里了，其实大家会发现，如果能够对讲解api系列的文章了解透了，那么后面的文章九很简单，就好像轨道铺好，可以直接顺着轨道往后，根本没有阻碍。接下来我将开始对rpc模块进行讲解。

阅读 710 • 更新于 11月8日


👍 赞 1

🔖 收藏 1



¥ 赞赏

🔗 分享

本作品系 原创 ， 作者保留所有权利，未经作者允许，禁止转载和演绎




crazyhzm

 265 

关注作者

0 条评论

得票 · 时间



撰写评论 ...

提交评论