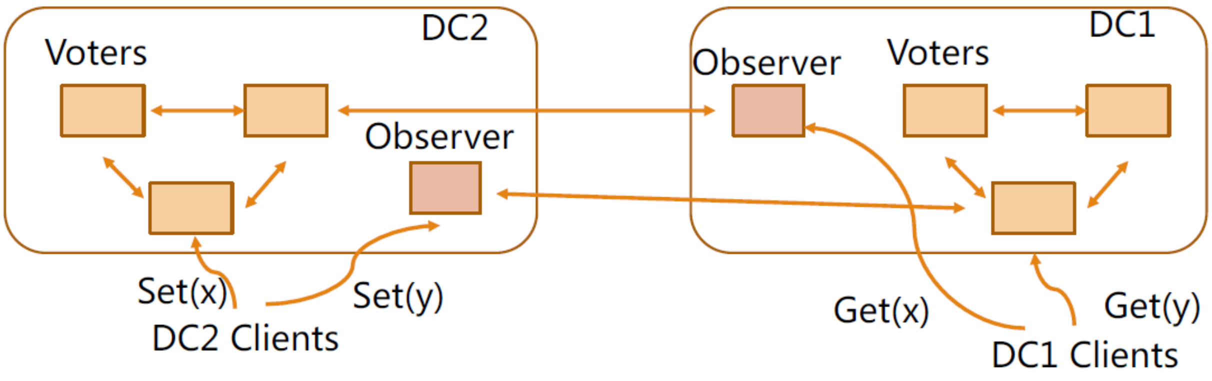
zookeeper本身具有容错机制，当一台机器宕机后会通过自身的leader选举机制重新选出master保证整个集群是可用状态，满足选举机制的前提条件是宕机的机器数量不能超过整个集群机器数量的一半，否则整个集群将不可用，在部署集群时往往采用奇数机器个数部署比较节省资源。

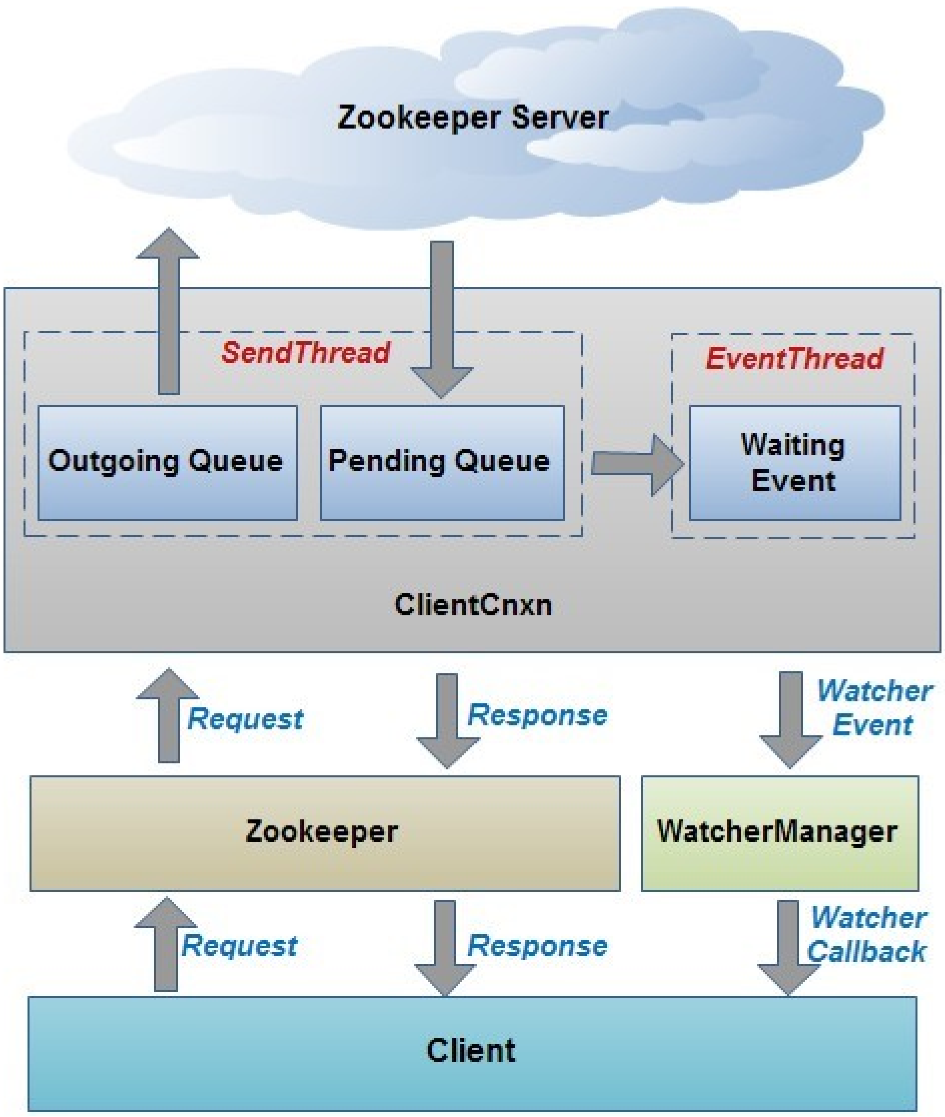
独立集群

跨数据中心方案采用独立集群方案，读写性能都很高，但对于全局数据唯一性并不能保证。

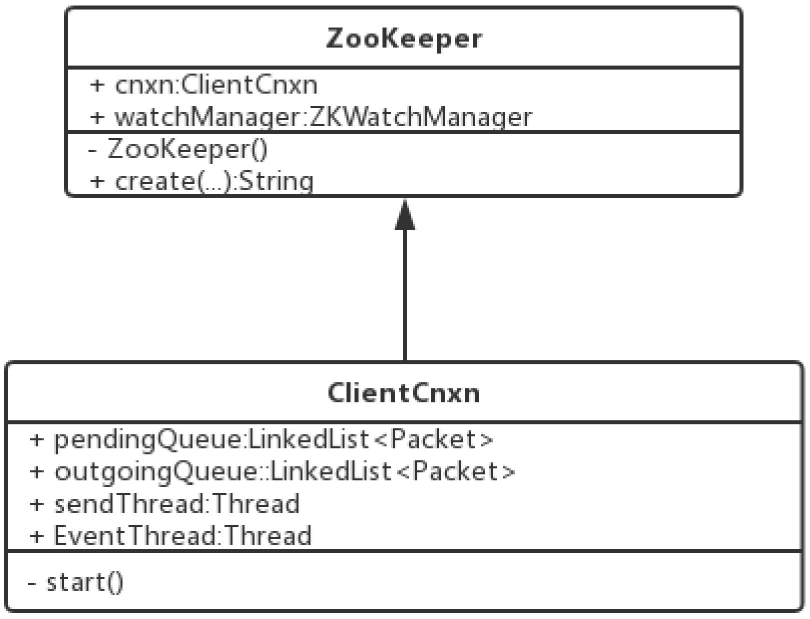


客户端连接

客户端的整体流程如下：



客户端发送请求到zookeeper对象，zookeeper对象将请求封装成Packt对象放入Outgoing队列中，连接就绪后会放入pending队列，最终会被消费生成Event放入EventThread。



其中客户端连接过程具体如下：

0：在构造Zookeeper对象时，会执行cnxn的start方法：

```
    public ZooKeeper(String connectString, int sessionTimeout, Watcher watcher,
             boolean canBeReadOnly)
         throws IOException
     {
         HostProvider hostProvider = new StaticHostProvider(
                 connectStringParser.getServerAddresses());
         cnxn = new ClientCnxn(connectStringParser.getChrootPath(),
                 hostProvider, sessionTimeout, this, watchManager,
                 getClientCnxnSocket(), canBeReadOnly);
         cnxn.start();
     }
```

1：start方法启动了SendThread和EventThread线程：

```
   public void start() {
         sendThread.start();
         eventThread.start();
     }
```

2：SendThread线程会有对服务端连接的操作(源码有删减)：

```
public void run() {
            while (state.isAlive()) {
                try {
                    if (!clientCnxnSocket.isConnected()) {
                        // 如果不是第一次连接，前一次可能连接失败就随机休眠不超过1000毫
                        if(!isFirstConnect){
                            try {
                                Thread.sleep(r.nextInt(1000));
                            } catch (InterruptedException e) {
                                LOG.warn("Unexpected exception", e);
                            }
                        }
                        // don't re-establish connection if we are closing
                        if (closing || !state.isAlive()) {
                            break;
                        }
                    // 执行连接操作
                    startConnect();
                    // 执行其他请求操作
                    clientCnxnSocket.doTransport(to, pendingQueue, outgoingQueue,
                }
                // getIdleRecv = 当前时间减去最后一次心跳时间
                 to = readTimeout - clientCnxnSocket.getIdleRecv();
                 // 判断是否连接超时
                  if (to <= 0) {
                        throw new SessionTimeoutException(
                                "Client session timed out, have not heard from se
                                        + clientCnxnSocket.getIdleRecv() + "ms"
                                        + " for sessionid 0x"
                                        + Long.toHexString(sessionId));
                }
            }catch (Throwable e) {
                //关闭当前socket连接并处理还未处理的客户端请求
                cleanup();
                //更新当前时间
                clientCnxnSocket.updateNow();
                //更新最后一次(预想)的发送和心跳时间
                clientCnxnSocket.updateLastSendAndHeard();
            }
        }
    }
```

3：startConnect方法从给的连接主机列表中按照顺序获取一个：

```
    private void startConnect() throws IOException {
            state = States.CONNECTING;

            InetSocketAddress addr;
            if (rwServerAddress != null) {
                addr = rwServerAddress;
                rwServerAddress = null;
            } else {
                // 获取连接的主机地址
                addr = hostProvider.next(1000);
            }
            // 创建socket通道并注册到selector（NIO）
            clientCnxnSocket.connect(addr);
        }
```

4：下标递增的方式获取连接主机：

```
   public InetSocketAddress next(long spinDelay) {
         ++currentIndex; //当前下标
         // 等于主机列表数量就置0，也就是会循环从第一个开始获取
         if (currentIndex == serverAddresses.size()) {
             currentIndex = 0;
         }
         // 如果是主机列表的最后一次且延迟连接参数大于0，那么进行休眠操作
         if (currentIndex == lastIndex && spinDelay > 0) {
             try {
                 Thread.sleep(spinDelay);
             } catch (InterruptedException e) {
                 LOG.warn("Unexpected exception", e);
             }
         } else if (lastIndex == -1) {
             // We don't want to sleep on the first ever connect attempt.
             lastIndex = 0;
         }
         // 获取主机列表中的某一个元素
         return serverAddresses.get(currentIndex);
     }
```

5：其中核心方法是doTransport方法里面的doIO方法，负责相关数据的读写操作(源码有删减)：

```
  void doIO(List<Packet> pendingQueue, LinkedList<Packet> outgoingQueue, ClientCnx
       throws InterruptedException, IOException {
         SocketChannel sock = (SocketChannel) sockKey.channel();
         if (sockKey.isReadable()) {
             // 这里如果连接断开会直接抛异常
             int rc = sock.read(incomingBuffer);
             // 更新心跳时间
             updateLastHeard();
         }
         if (sockKey.isWritable()) {
             synchronized(outgoingQueue) {
                 Packet p = findSendablePacket(outgoingQueue,
                         cnxn.sendThread.clientTunneledAuthenticationInProgress())
                 // 写入心跳包或者请客户端请求包
                 sock.write(p.bb);
             }
         }
     }
```

dubbo相关

dubbo在填写注册中心时会写上主用地址和备用地址：

```
<dubbo:registry address="zookeeper://172.22.23.120:2181?backup=172.22.23.121:2181
```

实际上在操作zk时会把两个地址都发给zookeeper：

```
         public ZkclientZookeeperClient(URL url) {
         super(url);
         client = new ZkClient(
                 url.getBackupAddress(),
                 url.getParameter(Constants.SESSION_TIMEOUT_KEY, Constants.DEFAULT_
                 url.getParameter(Constants.TIMEOUT_KEY, Constants.DEFAULT_REGISTR
     }
```

getBackupAddress方法获取主用地址后叠加了备用地址：

```
public String getBackupAddress() {
        return getBackupAddress(0);
    }

    public String getBackupAddress(int defaultPort) {
        StringBuilder address = new StringBuilder(appendDefaultPort(getAddress(),
        String[] backups = getParameter(Constants.BACKUP_KEY, new String[0]);
        if (backups != null && backups.length > 0) {
            for (String backup : backups) {
                address.append(",");
                address.append(appendDefaultPort(backup, defaultPort));
            }
        }
        return address.toString();
    }
```

当主用服务不能正常工作时(宕机)，dubbo会把注册信息写入到可用的备用地址：
首先ZookeeperRegistry注册器会创建一个ZookeeperClient对象：

```
//创建dubbo包装后的zk客户端
        zkClient = zookeeperTransporter.connect(url);
//新增一个状态监听器
        zkClient.addStateListener(new StateListener() {
//连接状态发生变化时触发
            public void stateChanged(int state) {
//判断状态是断线重填
                if (state == RECONNECTED) {
                    try {
//重新恢复注册信息
                        recover();
                    } catch (Exception e) {
                        logger.error(e.getMessage(), e);
                    }
                }
            }
        });
```

ZookeeperClient对象初始化代码如下：

```
// 实例化zk自身客户端
    client = new ZkClient(
                url.getBackupAddress(),
                url.getParameter(Constants.SESSION_TIMEOUT_KEY, Constants.DEFAULT_
                url.getParameter(Constants.TIMEOUT_KEY, Constants.DEFAULT_REGISTR
// 监听连接状态变化
client.subscribeStateChanges(new IZkStateListener() {
            public void handleStateChanged(KeeperState state) throws Exception {
                ZkclientZookeeperClient.this.state = state;
                if (state == KeeperState.Disconnected) {
                    stateChanged(StateListener.DISCONNECTED);
                } else if (state == KeeperState.SyncConnected) {
                    stateChanged(StateListener.CONNECTED);
                }
            }
            // 新打开一个连接触发（前一个连接不可用）
            public void handleNewSession() throws Exception {
                stateChanged(StateListener.RECONNECTED);
            }
        });
```

recover方法继承于FailbackRegistry类，会把已注册和已订阅的地址信息分别放入两个Set列表中：

```
    @Override
      protected void recover() throws Exception {
          // 把已注册的地址信息放入成员变量failedRegistered中
          Set<URL> recoverRegistered = new HashSet<URL>(getRegistered());
          if (! recoverRegistered.isEmpty()) {
              if (logger.isInfoEnabled()) {
                  logger.info("Recover register url " + recoverRegistered);
              }
              for (URL url : recoverRegistered) {
                  failedRegistered.add(url);
              }
          }
           // 把已订阅的地址信息放入成员变量failedSubscribed中
          Map<URL, Set<NotifyListener>> recoverSubscribed = new HashMap<URL, Set<No
          if (! recoverSubscribed.isEmpty()) {
              if (logger.isInfoEnabled()) {
                  logger.info("Recover subscribe url " + recoverSubscribed.keySet()
              }
              for (Map.Entry<URL, Set<NotifyListener>> entry : recoverSubscribed.en
                  URL url = entry.getKey();
                  for (NotifyListener listener : entry.getValue()) {
                      addFailedSubscribed(url, listener);
                  }
              }
          }
      }
```

最后会有一个定时器去扫描上面说的失败列表并尝试重新注册：

```
public FailbackRegistry(URL url) {
        super(url);
        int retryPeriod = url.getParameter(Constants.REGISTRY_RETRY_PERIOD_KEY, C
        this.retryFuture = retryExecutor.scheduleWithFixedDelay(new Runnable() {
            public void run() {
                // 检测并连接注册中心
                try {
                    retry();
                } catch (Throwable t) { // 防御性容错
                    logger.error("Unexpected error occur at failed retry, cause:
                }
            }
        }, retryPeriod, retryPeriod, TimeUnit.MILLISECONDS);
    }
// 重试失败的动作(源码有删减)
    protected void retry() {
        if (! failedRegistered.isEmpty()) {
             for (URL url : failed) {
                    doRegister(url);
                    failedRegistered.remove(url);
                }
            }
        }
        if(! failedUnregistered.isEmpty()) {
           for (URL url : failed) {
                doUnregister(url);
                failedUnregistered.remove(url);
            }
        }
        if (! failedSubscribed.isEmpty()) {
              for (Map.Entry<URL, Set<NotifyListener>> entry : failed.entrySet())
                    URL url = entry.getKey();
                    Set<NotifyListener> listeners = entry.getValue();
                    for (NotifyListener listener : listeners) {
                        try {
                            doSubscribe(url, listener);
                            listeners.remove(listener);
                        } catch (Throwable t) { // 忽略所有异常，等待下次重试
                            logger.warn("Failed to retry subscribe " + failed + "
                         }
                    }
                }
        }
        // .....
    }
}
```

结论

- 客户端连接zookeeper服务器允许容错的数量是n-1
- 一个集群只要还有超过半数的机器正常那么整个服务对外是正常的(官方说法)
- dubbo的注册中心至少要保证主用和备用其中一个服务正常

相关参考

zookeeper 跨机房 (http://m635674608.iteye.com/blog/2278725)
zookeeper 饿了么多活实践 (http://s.itho.me/modernweb/2017/day2/201-2-%E8%B6%99%E5%AD%90%E6%98%8E.pdf)