咕泡学院 -Zookeeper-watcher 机制源码分析

听完课程以后，有很多同学对源码这块比较晕，所以我把源码再整理一个文档，大家再按照文档来理解下

Watcher 的基本流程

ZooKeeper 的 Watcher 机制，总的来说可以分为三个过程：客户端注册 Watcher、服务器处理 Watcher 和客户端回调 Watcher

客户端注册 watcher 有 3 种方式，getData、exists、getChildren；以如下代码为例来分析整个触发机制的原理

```
ZooKeeper zookeeper=new ZooKeeper("192.168.11.152:2181",4000,new Watcher(){
    public void processor(WatchedEvent event){
        System.out.println("event.type");
    }
});
```

```
zookeeper.create("/mic","0".getByte(),ZooDefs.Ids.
OPEN_ACL_UNSAFE,CreateModel. PERSISTENT);  //创建节点

zookeeper.exists("/mic",true); //注册监听

zookeeper.setData("/mic", "1".getByte(),-1) ; //修改节点的值触发监听
```

ZooKeeper API 的初始化过程

```
ZooKeeper                                zookeeper=new
ZooKeeper("192.168.11.152:2181",4000,new Watcher(){
    public void processor(WatchedEvent event){
        System.out.println("event.type");
    }
});
```

在创建一个 ZooKeeper 客户端对象实例时，我们通过 new Watcher()向构造方法中传入一个默认的 Watcher，这个 Watcher 将作为整个 ZooKeeper 会话期间的默认 Watcher，会一直被保存在客户端 ZKWatchManager 的

defaultWatcher 中;代码如下

```
    public ZooKeeper(String connectString, int sessionTimeout, Watcher watcher,
            boolean canBeReadOnly, HostProvider aHostProvider,
            ZKClientConfig clientConfig) throws IOException {
        LOG.info("Initiating client connection, connectString=" + connectString
                + " sessionTimeout=" + sessionTimeout + " watcher=" + watcher);

        if (clientConfig == null) {
            clientConfig = new ZKClientConfig();
        }
        this.clientConfig = clientConfig;
        watchManager = defaultWatchManager();
        watchManager.defaultWatcher = watcher;  --在这里将 watcher 设置到 ZKWatchManager
        ConnectStringParser connectStringParser = new ConnectStringParser(
                connectString);
```

```
            hostProvider = aHostProvider;
            --初始化了 ClientCnxn，并且调用 cnxn.start()
方法
            cnxn                    =                    new
ClientCnxn(connectStringParser.getChrootPath(),
                    hostProvider,  sessionTimeout,  this,
watchManager,
                    getClientCnxnSocket(),
canBeReadOnly);
            cnxn.start();
        }
```

ClientCnxn:是 Zookeeper 客户端和 Zookeeper 服务器端进行通信和事件通知处理的主要类，它内部包含两个类，

1. SendThread ：负责客户端和服务器端的数据通信，也包括事件信息的传输

2. EventThread：主要在客户端回调注册的 Watchers 进行通知处理

ClientCnxn 初始化

```
    public ClientCnxn(String chrootPath, HostProvider
hostProvider, int sessionTimeout, ZooKeeper
zooKeeper,
```

```
                ClientWatchManager watcher,
ClientCnxnSocket clientCnxnSocket,
                long sessionId, byte[] sessionPasswd,
boolean canBeReadOnly) {
        this.zooKeeper = zooKeeper;
        this.watcher = watcher;
        this.sessionId = sessionId;
        this.sessionPasswd = sessionPasswd;
        this.sessionTimeout = sessionTimeout;
        this.hostProvider = hostProvider;
        this.chrootPath = chrootPath;

        connectTimeout = sessionTimeout /
hostProvider.size();
        readTimeout = sessionTimeout * 2 / 3;
        readOnly = canBeReadOnly;

        sendThread = new
SendThread(clientCnxnSocket);  --初始化 sendThread
        eventThread = new EventThread();
--初始化 eventThread
```

```
this.clientConfig=zooKeeper.getClientConfig();
    }

    public void start() { --启动两个线程
        sendThread.start();
        eventThread.start();
    }
```

客户端通过 exists 注册监听

```
zookeeper.exists("/mic",true); //注册监听
```

通过 exists 方法来注册监听，代码如下

```
    public Stat exists(final String path, Watcher
watcher)
        throws KeeperException,
InterruptedException
    {
        final String clientPath = path;
        PathUtils.validatePath(clientPath);

        // the watch contains the un-chroot path
        WatchRegistration wcb = null;
```

```
            if (watcher != null) {
                wcb = new ExistsWatchRegistration(watcher, clientPath); // 构建 ExistWatchRegistration
            }

            final String serverPath = prependChroot(clientPath);

            RequestHeader h = new RequestHeader();
            h.setType(ZooDefs.OpCode.exists);    // 设置操作类型为 exists
            ExistsRequest request = new ExistsRequest(); // 构造 ExistsRequest
            request.setPath(serverPath);
            request.setWatch(watcher != null);  //是否注册监听
            SetDataResponse response = new SetDataResponse();  //设置服务端响应的接收类
            //将封装的 RequestHeader、ExistsRequest、SetDataResponse、WatchRegistration 添加到发送队列
            ReplyHeader r = cnxn.submitRequest(h,
```

```
request, response, wcb);
            if (r.getErr() != 0) {
                if                  (r.getErr()                  ==
KeeperException.Code.NONODE.intValue()) {
                    return null;
                }
                throw
KeeperException.create(KeeperException.Code.get(r.g
etErr()),
                                clientPath);
            }
            //返回 exists 得到的结果（Stat 信息）
            return  response.getStat().getCzxid()  ==  -1  ?
null : response.getStat();
        }
```

cnxn.submitRequest

```
public ReplyHeader submitRequest(RequestHeader h,
Record request,
                Record    response,    WatchRegistration
watchRegistration,
                WatchDeregistration
```

```
watchDeregistration)
                throws InterruptedException {
        ReplyHeader r = new ReplyHeader();
        //将消息添加到队列,并构造一个 Packet 传输对象
        Packet packet = queuePacket(h, r, request, response, null, null, null, null, watchRegistration, watchDeregistration);
        synchronized (packet) {
            while (!packet.finished) { //在数据包没有处理完成之前，一直阻塞
                packet.wait();
            }
        }
        return r;
    }
```

```
    public Packet queuePacket(RequestHeader h, ReplyHeader r, Record request,
                Record response, AsyncCallback cb, String clientPath,
                String serverPath, Object ctx,
```

```
WatchRegistration watchRegistration,
                    WatchDeregistration
watchDeregistration) {
        //将相关传输对象转化成 Packet
        Packet packet = null;
        packet = new Packet(h, r, request, response,
watchRegistration);
        packet.cb = cb;
        packet.ctx = ctx;
        packet.clientPath = clientPath;
        packet.serverPath = serverPath;
        packet.watchDeregistration                    =
watchDeregistration;

        synchronized (state) {
            if (!state.isAlive() || closing) {
                conLossPacket(packet);
            } else {
                if              (h.getType()          ==
OpCode.closeSession) {
                    closing = true;
                }
```

```
                    outgoingQueue.add(packet); //添加到 outgoingQueue
                }
            }

sendThread.getClientCnxnSocket().packetAdded();//此处是多路复用机制，唤醒 Selector，告诉他有数据包添加过来了
            return packet;
        }
```

在 ZooKeeper 中，Packet 是一个最小的通信协议单元，即数据包。Pakcet 用于进行客户端与服务端之间的网络传输，任何需要传输的对象都需要包装成一个 Packet 对象。在 ClientCnxn 中 WatchRegistration 也会被封装到 Pakcet 中，然后由 SendThread 线程调用 queuePacket 方法把 Packet 放入发送队列中等待客户端发送，这又是一个异步过程，分布式系统采用异步通信是一个非常常见的手段

SendThread 的发送过程

在初始化连接的时候，zookeeper 初始化了两个线程并且启动了。接下来我们来分析 SendThread 的发送过程，因

为是一个线程，所以启动的时候会调用 SendThread.run 方法

```
public void run() {
                clientCnxnSocket.introduce(this,
sessionId, outgoingQueue);
                clientCnxnSocket.updateNow();

clientCnxnSocket.updateLastSendAndHeard();
                int to;
                long          lastPingRwServer          =
Time.currentElapsedTime();
                final  int  MAX_SEND_PING_INTERVAL  =
10000; //10 seconds
                while (state.isAlive()) {
                    try {
                        if
(!clientCnxnSocket.isConnected()) {// 如果没有连接：发起连接
                            //      don't      re-establish
connection if we are closing
                            if (closing) {
                                break;
```

```
                    }
                    startConnect(); //发起连接

clientCnxnSocket.updateLastSendAndHeard();
                }

                if (state.isConnected()) { //如果是连接状态，则处理 sasl 的认证授权
                    // determine whether we need to send an AuthFailed event.
                    if (zooKeeperSaslClient != null) {
                        boolean sendAuthEvent = false;
                        if (zooKeeperSaslClient.getSaslState() == ZooKeeperSaslClient.SaslState.INITIAL) {
                            try {
                                zooKeeperSaslClient.initialize(ClientCnxn.this);
                            } catch (SaslException e) {
```

```
LOG.error("SASL authentication with Zookeeper Quorum member failed: " + e);
                                state = States.AUTH_FAILED;

sendAuthEvent = true;
                            }
                        }
                        KeeperState authState = zooKeeperSaslClient.getKeeperState();
                        if (authState != null) {
                            if (authState == KeeperState.AuthFailed) {
                                // An authentication error occurred during authentication with the Zookeeper Server.
                                state = States.AUTH_FAILED;

sendAuthEvent = true;
                            } else {
```

```
                                        if (authState
== KeeperState.SaslAuthenticated) {

sendAuthEvent = true;
                                        }
                                    }
                                }

                                if (sendAuthEvent ==
true) {

eventThread.queueEvent(new WatchedEvent(

Watcher.Event.EventType.None,

authState,null));
                                }
                            }
                            to = readTimeout -
clientCnxnSocket.getIdleRecv();
                        } else {
                            to = connectTimeout -
```

```
clientCnxnSocket.getIdleRecv();
                    }
                    //to,表示客户端距离 timeout 还剩多少时间，准备发起 ping 连接
                    if (to <= 0) {//表示已经超时了。
                        String warnInfo;
                        warnInfo = "Client session timed out, have not heard from server in "
                                +
clientCnxnSocket.getIdleRecv()
                                + "ms"
                                + " for sessionid 0x"
                                +
Long.toHexString(sessionId);
                        LOG.warn(warnInfo);
                        throw new SessionTimeoutException(warnInfo);
                    }
                    if (state.isConnected()) {
                        //计算下一次 ping 请求的时间
                        int timeToNextPing =
```

```
readTimeout / 2 - clientCnxnSocket.getIdleSend() -

    ((clientCnxnSocket.getIdleSend() > 1000) ? 1000 : 0);
                        //send a ping request either
time is due or no packet sent out within
MAX_SEND_PING_INTERVAL
                        if (timeToNextPing <= 0 ||
clientCnxnSocket.getIdleSend()                    >
MAX_SEND_PING_INTERVAL) {
                            sendPing(); //发送 ping
请求

clientCnxnSocket.updateLastSend();
                        } else {
                            if (timeToNextPing <
to) {
                                to                =
timeToNextPing;
                            }
                        }
                    }
                // If we are in read-only mode,
```

```
seek for read/write server
                    if (state ==
States.CONNECTEDREADONLY) {
                        long now =
Time.currentElapsedTime();
                        int idlePingRwServer = (int)
(now - lastPingRwServer);
                        if (idlePingRwServer >=
pingRwTimeout) {
                            lastPingRwServer =
now;
                            idlePingRwServer = 0;
                            pingRwTimeout =

Math.min(2*pingRwTimeout, maxPingRwTimeout);
                            pingRwServer();
                        }
                        to = Math.min(to,
pingRwTimeout - idlePingRwServer);
                    }
```

调用 clientCnxnSocket，发起传输

其中 pendingQueue 是一个用

来存放已经发送、等待回应的 Packet 队列，

clientCnxnSocket 默认使用 ClientCnxnSocketNIO（ps：还记得在哪里初始化吗？在实例化 zookeeper 的时候）

```
clientCnxnSocket.doTransport(to, pendingQueue,
ClientCnxn.this);
                } catch (Throwable e) {
                    if (closing) {
                        if (LOG.isDebugEnabled()) {
                            // closing so this is
expected
                            LOG.debug("An
exception was thrown while closing send thread for
session 0x"
                                    +
Long.toHexString(getSessionId())
                                    + " : " +
e.getMessage());
                        }
                        break;
                    } else {
```

```
                    // this is ugly, you have a
better way speak up
                    if (e instanceof
SessionExpiredException) {

LOG.info(e.getMessage() + ", closing socket
connection");
                    } else if (e instanceof
SessionTimeoutException) {

LOG.info(e.getMessage() + RETRY_CONN_MSG);
                    } else if (e instanceof
EndOfStreamException) {

LOG.info(e.getMessage() + RETRY_CONN_MSG);
                    } else if (e instanceof
RWServerFoundException) {

LOG.info(e.getMessage());
                    } else {
                        LOG.warn(
                            "Session 0x"
```

```
                                                    +
Long.toHexString(getSessionId())
                                                    +    "
for server "
                                                    +
clientCnxnSocket.getRemoteSocketAddress()
                                                    +    ",
unexpected error"
                                                    +
RETRY_CONN_MSG, e);
                                }
                                // At this point, there might
still be new packets appended to outgoingQueue.
                                // they will be handled in
next connection or cleared up if closed.
                                cleanup();
                                if (state.isAlive()) {
eventThread.queueEvent(new WatchedEvent(
Event.EventType.None,
```

```
Event.KeeperState.Disconnected,
                                                    null));
                                }

clientCnxnSocket.updateNow();

clientCnxnSocket.updateLastSendAndHeard();
                            }
                        }
                    }
                    synchronized (state) {
                        // When it comes to this point, it
guarantees that later queued
                        // packet to outgoingQueue will be
notified of death.
                        cleanup();
                    }
                    clientCnxnSocket.close();
                    if (state.isAlive()) {
                        eventThread.queueEvent(new
WatchedEvent(Event.EventType.None,
```

```
Event.KeeperState.Disconnected, null));
                }
                ZooTrace.logTraceMessage(LOG,
ZooTrace.getTextTraceLevel(),
                        "SendThread exited loop for
session: 0x"
                                +
Long.toHexString(getSessionId()));
            }
```

client 和 server 的网络交互

```
@Override
    void doTransport(int waitTimeOut, List<Packet>
pendingQueue, ClientCnxn cnxn) throws IOException,
InterruptedException {
            try {
                if (!firstConnect.await(waitTimeOut,
TimeUnit.MILLISECONDS)) {
                    return;
                }
                Packet head = null;
                if (needSasl.get()) {
```

```
                    if (!waitSasl.tryAcquire(waitTimeOut,
TimeUnit.MILLISECONDS)) {
                        return;
                    }
                } else {
                    //判断 outgoingQueue 是否存在待发
送的数据包，不存在则直接返回
                    if ((head =
outgoingQueue.poll(waitTimeOut,
TimeUnit.MILLISECONDS)) == null) {
                        return;
                    }
                }
                // check if being waken up on closing.
                if (!sendThread.getZkState().isAlive()) {
                    // adding back the patck to notify of
failure in conLossPacket().
                    addBack(head);
                    return;
                }
                // channel disconnection happened
                if (disconnected.get()) { //异常流程，
```

```
channel 关闭了，讲当前的 packet 添加到 addBack 中
                    addBack(head);
                    throw                    new
EndOfStreamException("channel for sessionid 0x"
                            +
Long.toHexString(sessionId)
                            + " is lost");
                }
                if (head != null) { //如果当前存在需要发送
的数据包，则调用 doWrite 方法，pendingQueue 表示
处于已经发送过等待响应的 packet 队列
                    doWrite(pendingQueue,       head,
cnxn);
                }
            } finally {
                updateNow();
            }
        }
```

DoWrite 方法

```
    private        void        doWrite(List<Packet>
pendingQueue, Packet p, ClientCnxn cnxn) {
```

```
            updateNow();
            while (true) {
                if (p != WakeupPacket.getInstance()) {
                    if ((p.requestHeader != null) && //判
断请求头以及判断当前请求类型不是 ping 或者 auth 操
作

(p.requestHeader.getType() != ZooDefs.OpCode.ping)
&&

(p.requestHeader.getType() != ZooDefs.OpCode.auth))
{

p.requestHeader.setXid(cnxn.getXid());  //设置 xid，这
个 xid 用来区分请求类型
                        synchronized (pendingQueue) {
                            pendingQueue.add(p); //将
当前的 packet 添加到 pendingQueue 队列中
                        }
                    }
                    sendPkt(p); //将数据包发送出去
                }
```

```
                if (outgoingQueue.isEmpty()) {
                    break;
                }
                p = outgoingQueue.remove();
            }
        }
```

sendPkt

```
    private void sendPkt(Packet p) {
            // Assuming the packet will be sent out successfully. Because if it fails,
            // the channel will close and clean up queues.
            p.createBB(); //序列化请求数据
            updateLastSend();  // 更新最后一次发送 updateLastSend
            sentCount++; //更新发送次数

channel.write(ChannelBuffers.wrappedBuffer(p.bb)); // 通过 nio channel 发送字节缓存到服务端
        }
```

createBB

```
public void createBB() {
                try {
                        ByteArrayOutputStream baos = new
ByteArrayOutputStream();
                        BinaryOutputArchive boa =
BinaryOutputArchive.getArchive(baos);
                        boa.writeInt(-1, "len"); // We'll fill this
in later
                        if (requestHeader != null) {
                                requestHeader.serialize(boa,
"header"); //序列化 header 头(requestHeader)
                        }
                        if (request instanceof
ConnectRequest) {
                                request.serialize(boa,
"connect");
                                // append "am-I-allowed-to-
be-readonly" flag
                                boa.writeBool(readOnly,
"readOnly");
```

```
                    } else if (request != null) {
                        request.serialize(boa, "request");
//序列化 request(request)
                    }
                    baos.close();
                    this.bb                                    =
ByteBuffer.wrap(baos.toByteArray());
                    this.bb.putInt(this.bb.capacity() - 4);
                    this.bb.rewind();
                } catch (IOException e) {
                    LOG.warn("Ignoring      unexpected
exception", e);
                }
            }
```

从 createBB 方法中，我们看到在底层实际的网络传输序列化中，zookeeper 只会讲 requestHeader 和 request 两个属性进行序列化，即只有这两个会被序列化到底层字节数组中去进行网络传输，不会将 watchRegistration 相关的信息进行网络传输。

总结

用户调用 exists 注册监听以后，会做几个事情

1. 讲请求数据封装为 packet，添加到 outgoingQueue
2. SendThread 这个线程会执行数据发送操作，主要是将 outgoingQueue 队列中的数据发送到服务端
3. 通过 clientCnxnSocket.doTransport(to, pendingQueue, ClientCnxn.this); 其中 ClientCnxnSocket 只 zookeeper 客户端和服务端的连接通信的封装，有两个具体的实现类 ClientCnxnSocketNetty 和 ClientCnxnSocketNIO;具体使用哪一个类来实现发送，是在初始化过程是在实例化 Zookeeper 的时候设置的，代码如下

```
cnxn = new ClientCnxn(connectStringParser.getChrootPath(),
                      hostProvider, sessionTimeout, this, watchManag
                      getClientCnxnSocket(), canBeReadOnly);
```

```
private ClientCnxnSocket getClientCnxnSocket() throws IOException {
        String clientCnxnSocketName = getClientConfig().getProperty(
                ZKClientConfig.ZOOKEEPER_CLIENT_CNXN_SOCKET);
        if (clientCnxnSocketName == null) {
            clientCnxnSocketName = ClientCnxnSocketNIO.class.getName();
        }
        try {
            Constructor<?> clientCxnConstructor =
            Class.forName(clientCnxnSocketName).getDeclaredConstructor(ZKClient
```

```
            ClientCnxnSocket clientCxnSocket = (ClientCnxnSocket) clientCxnConstru
            return clientCxnSocket;
        } catch (Exception e) {
            IOException ioe = new IOException("Couldn't instantiate "
                    + clientCnxnSocketName);
            ioe.initCause(e);
            throw ioe;
        }
    }
```

4. 基于第 3 步，最终会在 ClientCnxnSocketNetty 方法中执行 sendPkt 将请求的数据包发送到服务端

服务端接收请求处理流程

服务端有一个 NettyServerCnxn 类，用来处理客户端发送过来的请求

NettyServerCnxn

```
public void receiveMessage(ChannelBuffer message) {
        try {
            while(message.readable() && !throttled) {
```

```
if (bb != null) { //ByteBuffer 不为空
    if (LOG.isTraceEnabled()) {
        LOG.trace("message readable " + messag
                + " bb len " + bb.remaining() +
        ByteBuffer dat = bb.duplicate();
        dat.flip();
        LOG.trace(Long.toHexString(sessionId)
                + " bb 0x"
                + ChannelBuffers.hexDump(
                        ChannelBuffers.copied
    }
    //bb 剩余空间大于 message 中可读字节大小
    if (bb.remaining() > message.readableBytes()
        int newLimit = bb.position() + message.r
        bb.limit(newLimit);
    }
    //  将 message 写入 bb 中
    message.readBytes(bb);
    bb.limit(bb.capacity());

    if (LOG.isTraceEnabled()) {
        LOG.trace("after readBytes message reac
```

```
                    + message.readableBytes()
                    + " bb len " + bb.remaining() +
            ByteBuffer dat = bb.duplicate();
            dat.flip();
            LOG.trace("after readbytes "
                    + Long.toHexString(sessionId)
                    + " bb 0x"
                    + ChannelBuffers.hexDump(
                            ChannelBuffers.copiedB
    }
    if (bb.remaining() == 0) { // 已经读完 messag
            packetReceived(); // 统计接收信息
            bb.flip();

            ZooKeeperServer zks = this.zkServer;
            if (zks == null || !zks.isRunning()) {//Zook
                throw new IOException("ZK down");
            }
            if (initialized) {
                //处理客户端传过来的数据包
                zks.processPacket(this, bb);
```

```
                    if (zks.shouldThrottle(outstandingCo
                        disableRecvNoWait();
                    }
                } else {
                    LOG.debug("got conn req request fr
                            + getRemoteSocketAddres
                    zks.processConnectRequest(this, bb)
                    initialized = true;
                }
                bb = null;
            }
        } else { //bb 为 null 的情况，大家自己去看，我就不
            if (LOG.isTraceEnabled()) {
                LOG.trace("message readable "
                        + message.readableBytes()
                        + " bblenrem " + bbLen.remaini
                ByteBuffer dat = bbLen.duplicate();
                dat.flip();
                LOG.trace(Long.toHexString(sessionId)
                        + " bbLen 0x"
                        + ChannelBuffers.hexDump(
                                ChannelBuffers.copied
```

```
        }

        if (message.readableBytes() < bbLen.remainin
            bbLen.limit(bbLen.position() + message.r
        }
        message.readBytes(bbLen);
        bbLen.limit(bbLen.capacity());
        if (bbLen.remaining() == 0) {
            bbLen.flip();

            if (LOG.isTraceEnabled()) {
                LOG.trace(Long.toHexString(sessionI
                        + " bbLen 0x"
                        + ChannelBuffers.hexDump
                                ChannelBuffers.co
            }
            int len = bbLen.getInt();
            if (LOG.isTraceEnabled()) {
                LOG.trace(Long.toHexString(sessionI
                        + " bbLen len is " + len);
            }
```

```
                    bbLen.clear();
                    if (!initialized) {
                        if (checkFourLetterWord(channel, me
                            return;
                        }
                    }
                    if (len < 0 || len > BinaryInputArchive.ma
                        throw new IOException("Len error "
                    }
                    bb = ByteBuffer.allocate(len);
                }
            }
        }
    } catch(IOException e) {
        LOG.warn("Closing connection to " + getRemoteSocke
        close();
    }
}
```

ZookeeperServer-zks.processPacket(this, bb);

处理客户端传送过来的数据包

```
public void processPacket(ServerCnxn cnxn, ByteBuffer incomingBu
```

```
// We have the request, now process and setup for next
InputStream bais = new ByteBufferInputStream(incomingB
BinaryInputArchive bia = BinaryInputArchive.getArchive(ba
RequestHeader h = new RequestHeader();
h.deserialize(bia, "header"); //反序列化客户端 header 头信息
// Through the magic of byte buffers, txn will not be
// pointing
// to the start of the txn
incomingBuffer = incomingBuffer.slice();
if (h.getType() == OpCode.auth) { //判断当前操作类型，如
    LOG.info("got auth packet " + cnxn.getRemoteSocketA
    AuthPacket authPacket = new AuthPacket();
    ByteBufferInputStream.byteBuffer2Record(incomingBu
    String scheme = authPacket.getScheme();
    ServerAuthenticationProvider ap = ProviderRegistry.ge
    Code authReturn = KeeperException.Code.AUTHFAILE
    if(ap != null) {
        try {
            authReturn = ap.handleAuthentication(new S
        } catch(RuntimeException e) {
            LOG.warn("Caught runtime exception from A
            authReturn = KeeperException.Code.AUTHFA
```

```
            }
        }
        if (authReturn == KeeperException.Code.OK) {
            if (LOG.isDebugEnabled()) {
                LOG.debug("Authentication succeeded for sc
            }
            LOG.info("auth success " + cnxn.getRemoteSocke
            ReplyHeader rh = new ReplyHeader(h.getXid(), 0,
                    KeeperException.Code.OK.intValue());
            cnxn.sendResponse(rh, null, null);
        } else {
            if (ap == null) {
                LOG.warn("No authentication provider for sch
                        + scheme + " has "
                        + ProviderRegistry.listProviders());
            } else {
                LOG.warn("Authentication failed for scheme:
            }
            // send a response...
            ReplyHeader rh = new ReplyHeader(h.getXid(), 0,
                    KeeperException.Code.AUTHFAILED.intVa
            cnxn.sendResponse(rh, null, null);
```

```
                // ... and close connection
                cnxn.sendBuffer(ServerCnxnFactory.closeConn);
                cnxn.disableRecv();
            }
            return;
        } else { //如果不是授权操作，再判断是否为 sasl 操作
            if (h.getType() == OpCode.sasl) {
                Record rsp = processSasl(incomingBuffer,cnxn);
                ReplyHeader rh = new ReplyHeader(h.getXid(), 0,
                cnxn.sendResponse(rh,rsp, "response"); // not sure
                return;
            }
            else {//最终进入这个代码块进行处理
                //封装请求对象
                Request si = new Request(cnxn, cnxn.getSessionId
                  h.getType(), incomingBuffer, cnxn.getAuthInfo());
                si.setOwner(ServerCnxn.me);
                // Always treat packet from the client as a possible
                // local request.
                setLocalSessionFlag(si);
                submitRequest(si); //提交请求
            }
```

```
            }
            cnxn.incrOutstandingRequests(h);
        }
```

submitRequest

负责在服务端提交当前请求

```
public void submitRequest(Request si) {
        if (firstProcessor == null) { //processor 处理器，request 过来
            synchronized (this) {
                try {
                    // Since all requests are passed to the reques
                    // processor it should wait for setting up the r
                    // processor chain. The state will be updated
                    // after the setup.
                    while (state == State.INITIAL) {
                        wait(1000);
                    }
                } catch (InterruptedException e) {
                    LOG.warn("Unexpected interruption", e);
                }
                if (firstProcessor == null || state != State.RUNNING
```

```
                    throw new RuntimeException("Not started");
                }
            }
        }
        try {
            touch(si.cnxn);
            boolean validpacket = Request.isValid(si.type); //判断是
            if (validpacket) {
                firstProcessor.processRequest(si);    调用 firstProces
往下看吧
                if (si.cnxn != null) {
                    incInProcess();
                }
            } else {
                LOG.warn("Received packet at server of unknown
                new UnimplementedRequestProcessor().processR
            }
        } catch (MissingSessionException e) {
            if (LOG.isDebugEnabled()) {
                LOG.debug("Dropping request: " + e.getMessage
            }
        } catch (RequestProcessorException e) {
```

```
                LOG.error("Unable to process request:" + e.getMessag
            }
        }
```

firstProcessor 的请求链组成

1. firstProcessor 的初始化是在 ZookeeperServer 的 setupRequestProcessor 中完成的，代码如下

```
        protected void setupRequestProcessors() {
            RequestProcessor finalProcessor = new FinalReques
            RequestProcessor syncProcessor = new SyncReques
            ((SyncRequestProcessor)syncProcessor).start();
            firstProcessor = new PrepRequestProcessor(this, syn
            ((PrepRequestProcessor)firstProcessor).start();
    }
```

从上面我们可以看到 firstProcessor 的实例是一个 PrepRequestProcessor，而这个构造方法中又传递了一个 Processor 构成了一个调用链。

RequestProcessor syncProcessor = new SyncRequestProcessor(this, finalProcessor);

而 syncProcessor 的构造方法传递的又是一个 Processor，对应的是 FinalRequestProcessor

2. 所以整个调用链是 PrepRequestProcessor -> SyncRequestProcessor ->FinalRequestProcessor

PredRequestProcessor.processRequest(si);

通过上面了解到调用链关系以后，我们继续再看 firstProcessor.processRequest(si) ； 会调用到 PrepRequestProcessor

```
public void processRequest(Request request) {
        submittedRequests.add(request);
}
```

唉，很奇怪，processRequest 只是把 request 添加到 submittedRequests 中，根据前面的经验，很自然的想到这里又是一个异步操作。而 subittedRequests 又是一个阻塞队列

LinkedBlockingQueue<Request> submittedRequests = new LinkedBlockingQueue<Request>();

而 PrepRequestProcessor 这个类又继承了线程类，因此我们直接找到当前类中的 run 方法如下

```
public void run() {
        try {
            while (true) {
                Request request =
```

```
submittedRequests.take(); //ok，从队列中拿到请求进行处理
                    long traceMask = ZooTrace.CLIENT_REQUEST_TRACE_MASK;
                    if (request.type == OpCode.ping) {
                        traceMask = ZooTrace.CLIENT_PING_TRACE_MASK;
                    }
                    if (LOG.isTraceEnabled()) {
                        ZooTrace.logRequest(LOG, traceMask, 'P', request, "");
                    }
                    if (Request.requestOfDeath == request) {
                        break;
                    }
                    pRequest(request); //调用 pRequest 进行预处理
                }
            } catch (RequestProcessorException e) {
                if (e.getCause() instanceof XidRolloverException) {
```

```
LOG.info(e.getCause().getMessage());
                }
                handleException(this.getName(), e);
            } catch (Exception e) {
                handleException(this.getName(), e);
            }
            LOG.info("PrepRequestProcessor exited loop!");
        }
```

pRequest

预处理这块的代码太长，就不好贴了。前面的 N 行代码都是根据当前的 OP 类型进行判断和做相应的处理，在这个方法中的最后一行中，我们会看到如下代码

```
nextProcessor.processRequest(request);
```

很显然，nextProcessor 对应的应该是 SyncRequestProcessor

SyncRequestProcessor. processRequest

```
    public void processRequest(Request request) {
        // request.addRQRec(">sync");
```

```
            queuedRequests.add(request);
        }
```

这个方法的代码也是一样，基于异步化的操作，把请求添加到 queuedRequets 中，那么我们继续在当前类找到 run 方法

```
public void run() {
            try {
                int logCount = 0;

                // we do this in an attempt to ensure that
not all of the servers
                // in the ensemble take a snapshot at the
same time
                int randRoll = r.nextInt(snapCount/2);
                while (true) {
                    Request si = null;
                    //从阻塞队列中获取请求
                    if (toFlush.isEmpty()) {
                        si = queuedRequests.take();
                    } else {
                        si = queuedRequests.poll();
```

```
                    if (si == null) {
                        flush(toFlush);
                        continue;
                    }
                }
                if (si == requestOfDeath) {
                    break;
                }
                if (si != null) {
                    // track the number of records
written to the log
                    //下面这块代码，粗略看来是触发
快照操作，启动一个处理快照的线程
                    if
(zks.getZKDatabase().append(si)) {
                        logCount++;
                        if (logCount > (snapCount /
2 + randRoll)) {
                            randRoll =
r.nextInt(snapCount/2);
                            // roll the log
```

```
zks.getZKDatabase().rollLog();
                                    // take a snapshot
                                    if (snapInProcess != null && snapInProcess.isAlive()) {
                                        LOG.warn("Too busy to snap, skipping");
                                    } else {
                                        snapInProcess = new ZooKeeperThread("Snapshot Thread") {
                                            public void run() {
                                                try {
zks.takeSnapshot();
                                                }
catch(Exception e) {
LOG.warn("Unexpected exception", e);
                                                }
                                            }
                                        };
```

```
snapInProcess.start();
                    }
                    logCount = 0;
                }
            } else if (toFlush.isEmpty()) {
                // optimization for read
heavy workloads
                // iff this is a read, and there
are no pending
                // flushes (writes), then just
pass this to the next
                // processor
                if (nextProcessor != null) {

nextProcessor.processRequest(si); //继续调用下一个处
理器来处理请求
                    if (nextProcessor
instanceof Flushable) {

((Flushable)nextProcessor).flush();
                    }
                }
```

```
                    continue;
                }
                toFlush.add(si);
                if (toFlush.size() > 1000) {
                    flush(toFlush);
                }
            }
        }
    } catch (Throwable t) {
        handleException(this.getName(), t);
    } finally{
        running = false;
    }
    LOG.info("SyncRequestProcessor exited!");
}
```

FinalRequestProcessor. processRequest

这个方法就是我们在课堂上分析到的方法了，FinalRequestProcessor.processRequest 方法并根据 Request 对象中的操作更新内存中 Session 信息或者 znode 数据。

这块代码有小 300 多行，就不全部贴出来了，我们直接定

位到关键代码，根据客户端的 OP 类型找到如下的代码

```
case OpCode.exists: {
    lastOp = "EXIS";
    // TODO we need to figure out the security requirement for this!
    ExistsRequest existsRequest = new ExistsRequest();
    //反序列化  (将 ByteBuffer 反序列化成为 ExitsRequest.这个就是我们在客户端发起请求的时候传递过来的 Request 对象

    ByteBufferInputStream.byteBuffer2Record(request.request,
                                            existsRequest);
    String path = existsRequest.getPath(); //得到请求的路径
    if (path.indexOf('\0') != -1) {
        throw new KeeperException.BadArgumentsException();
    }
    //终于找到一个很关键的代码，判断请求的 getWatch 是否存在，如果存在，则传递 cnxn
```

```
（servercnxn）
                    //对于 exists 请求，需要监听 data 变化事件，添加 watcher
                    Stat stat = zks.getZKDatabase().statNode(path, existsRequest.getWatch() ? cnxn : null);
                    rsp = new ExistsResponse(stat); //在服务端内存数据库中根据路径得到结果进行组装，设置为 ExistsResponse
                    break;
                }
```

statNode 这个方法做了什么？

```
    public Stat statNode(String path, ServerCnxn serverCnxn) throws KeeperException.NoNodeException {
        return dataTree.statNode(path, serverCnxn);
    }
```

一路向下，在下面这个方法中，讲 ServerCnxn 向上转型为 Watcher 了。 因为 ServerCnxn 实现了 Watcher 接口

```
public Stat statNode(String path, Watcher watcher)
```

```
                throws
KeeperException.NoNodeException {
        Stat stat = new Stat();
        DataNode n = nodes.get(path); //获得节点数
据
        if (watcher != null) { //如果 watcher 不为空，
则讲当前的 watcher 和 path 进行绑定
            dataWatches.addWatch(path, watcher);
        }
        if (n == null) {
            throw                                   new
KeeperException.NoNodeException();
        }
        synchronized (n) {
            n.copyStat(stat);
            return stat;
        }
    }
```

WatchManager.addWatch(path, watcher);

```
synchronized void addWatch(String path, Watcher
watcher) {
```

```
HashSet<Watcher> list = watchTable.get(path);  //判断 watcherTable 中是否存在当前路径对应的 watcher
if (list == null) { //不存在则主动添加
    // don't waste memory if there are few watches on a node
    // rehash when the 4th entry is added, doubling size thereafter
    // seems like a good compromise
    list = new HashSet<Watcher>(4); // 新生成 watcher 集合
    watchTable.put(path, list);
}
list.add(watcher); //添加到 watcher 表

HashSet<String> paths = watch2Paths.get(watcher);
if (paths == null) {
    // cnxns typically have many watches, so use default cap here
    paths = new HashSet<String>();
    watch2Paths.put(watcher, paths); // 设置
```

```
watcher 到节点路径的映射
            }
            paths.add(path);    // 将路径添加至 paths 集合
        }
```

其大致流程如下

① 通过传入的 path（节点路径）从 watchTable 获取相应的 watcher 集合，进入②

② 判断①中的 watcher 是否为空，若为空，则进入③，否则，进入④

③ 新生成 watcher 集合，并将路径 path 和此集合添加至 watchTable 中，进入④

④ 将传入的 watcher 添加至 watcher 集合，即完成了 path 和 watcher 添加至 watchTable 的步骤，进入⑤

⑤ 通过传入的 watcher 从 watch2Paths 中获取相应的 path 集合，进入⑥

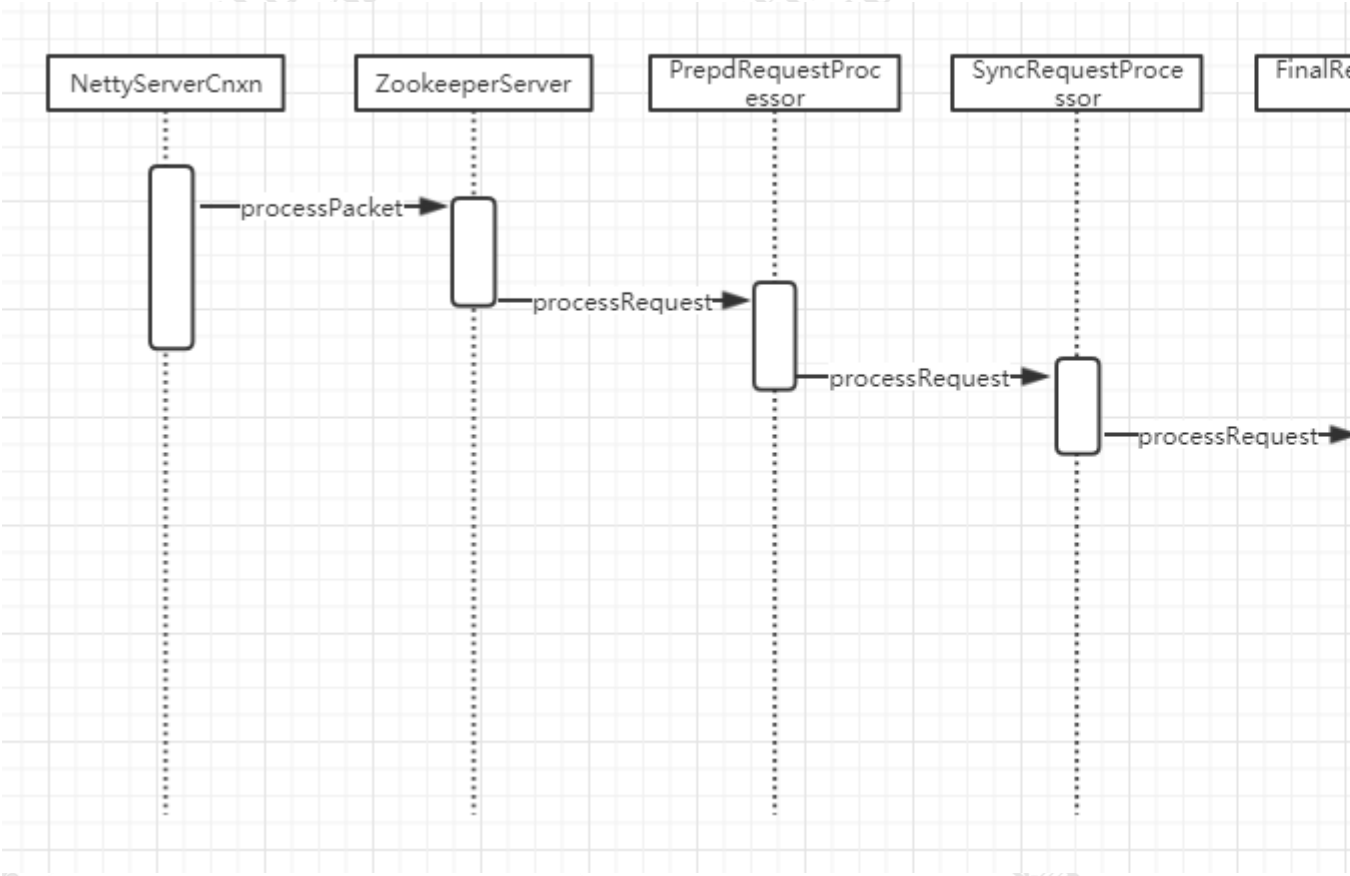
⑥ 判断 path 集合是否为空，若为空，则进入⑦，否则，进入⑧

⑦ 新生成 path 集合，并将 watcher 和 paths 添加至 watch2Paths 中，进入⑧

⑧ 将传入的 path（节点路径）添加至 path 集合，即完成了 path 和 watcher 添加至 watch2Paths 的步骤

总结

调用关系链如下



客户端接收服务端处理完成的响应

ClientCnxnSocketNetty.messageReceived

服务端处理完成以后，会通过 NettyServerCnxn.sendResponse 发送返回的响应信息，客户端会在 ClientCnxnSocketNetty.messageReceived 接收服务端的返回

```
public void messageReceived(ChannelHandlerContext
```

```
ctx,

MessageEvent e) throws Exception {
                updateNow();
                ChannelBuffer buf = (ChannelBuffer)
e.getMessage();
                while (buf.readable()) {
                    if (incomingBuffer.remaining() >
buf.readableBytes()) {
                        int newLimit =
incomingBuffer.position()
                                + buf.readableBytes();
                        incomingBuffer.limit(newLimit);
                    }
                    buf.readBytes(incomingBuffer);

incomingBuffer.limit(incomingBuffer.capacity());

                    if (!incomingBuffer.hasRemaining()) {
                        incomingBuffer.flip();
                        if (incomingBuffer == lenBuffer)
{
```

```
                    recvCount++;
                    readLength();
                } else if (!initialized) {
                    readConnectResult();
                    lenBuffer.clear();
                    incomingBuffer = lenBuffer;
                    initialized = true;
                    updateLastHeard();
                } else {

sendThread.readResponse(incomingBuffer)；收到消息以后触发 SendThread.readResponse 方法
                    lenBuffer.clear();
                    incomingBuffer = lenBuffer;
                    updateLastHeard();
                }
            }
        }
        wakeupCnxn();
    }
```

SendThread. readResponse

这个方法里面主要的流程如下

首先读取 header，如果其 xid == -2，表明是一个 ping 的 response，return

如果 xid 是 -4 ，表明是一个 AuthPacket 的 response return

如果 xid 是 -1, 表明是一个 notification,此时要继续读取并构造一个 enent，通过 EventThread.queueEvent 发送，return

其它情况下：

从 pendingQueue 拿出一个 Packet，校验后更新 packet 信息

```
void readResponse(ByteBuffer incomingBuffer) throws
IOException {
                ByteBufferInputStream bbis = new
ByteBufferInputStream(
                        incomingBuffer);
                BinaryInputArchive bbia =
BinaryInputArchive.getArchive(bbis);
                ReplyHeader replyHdr = new
ReplyHeader();
```

```
                replyHdr.deserialize(bbia, "header"); //反
序列化 header
                if (replyHdr.getXid() == -2) { //?
                    // -2 is the xid for pings
                    if (LOG.isDebugEnabled()) {
                        LOG.debug("Got ping response
for sessionid: 0x"
                                +
Long.toHexString(sessionId)
                                + " after "
                                + ((System.nanoTime()
- lastPingSentNs) / 1000000)
                                + "ms");
                    }
                    return;
                }
                if (replyHdr.getXid() == -4) {
                    // -4 is the xid for AuthPacket
                    if(replyHdr.getErr()                ==
KeeperException.Code.AUTHFAILED.intValue()) {
                        state = States.AUTH_FAILED;
```

```
                    eventThread.queueEvent(    new
WatchedEvent(Watcher.Event.EventType.None,

Watcher.Event.KeeperState.AuthFailed,        null)        );

                }
                if (LOG.isDebugEnabled()) {
                    LOG.debug("Got              auth
sessionid:0x"
                            +
Long.toHexString(sessionId));
                }
                return;
            }
            if (replyHdr.getXid() == -1) { //表示当前的
消息类型为一个 notification(意味着是服务端的一个响
应事件)
                // -1 means notification
                if (LOG.isDebugEnabled()) {
                    LOG.debug("Got      notification
sessionid:0x"
                            +
```

```
Long.toHexString(sessionId));
                    }
                    WatcherEvent event = new
WatcherEvent();//?
                    event.deserialize(bbia, "response");
//反序列化响应信息

                    // convert from a server path to a
client path
                    if (chrootPath != null) {
                        String serverPath =
event.getPath();

if(serverPath.compareTo(chrootPath)==0)
                                event.setPath("/");
                        else if (serverPath.length() >
chrootPath.length())

event.setPath(serverPath.substring(chrootPath.length()
));
                            else {
                            LOG.warn("Got server path " +
```

```
event.getPath()
                                + " which is too short for
chroot path "
                                + chrootPath);
                        }
                    }

                    WatchedEvent we = new
WatchedEvent(event);
                    if (LOG.isDebugEnabled()) {
                        LOG.debug("Got " + we + " for
sessionid 0x"
                                +
Long.toHexString(sessionId));
                    }

                    eventThread.queueEvent( we );
                    return;
                }

                // If SASL authentication is currently in
progress, construct and
```

```
                // send a response packet immediately,
rather than queuing a
                // response as with other packets.
                if (tunnelAuthInProgress()) {
                    GetSASLRequest request = new
GetSASLRequest();
                    request.deserialize(bbia,"token");

zooKeeperSaslClient.respondToServer(request.getToke
n(),
                      ClientCnxn.this);
                    return;
                }

                Packet packet;
                synchronized (pendingQueue) {
                    if (pendingQueue.size() == 0) {
                        throw                    new
IOException("Nothing in the queue, but got "
                                + replyHdr.getXid());
                    }
                    packet = pendingQueue.remove();
```

```
//因为当前这个数据包已经收到了响应，所以讲它从pendingQueued 中移除
                }
                /*
                 * Since requests are processed in order, we better get a response
                 * to the first request!
                 */
                try {//校验数据包信息，校验成功后讲数据包信息进行更新（替换为服务端的信息）
                    if (packet.requestHeader.getXid() != replyHdr.getXid()) {
                        packet.replyHeader.setErr(
                                KeeperException.Code.CONNECTIONLOSS.intValue());
                        throw new IOException("Xid out of order. Got Xid "
                                + replyHdr.getXid() + " with err " +
                                + replyHdr.getErr() +
                                " expected Xid "
                                +
```

```
packet.requestHeader.getXid()
                                    + " for a packet with
details: "
                                    + packet );
                    }


packet.replyHeader.setXid(replyHdr.getXid());

packet.replyHeader.setErr(replyHdr.getErr());

packet.replyHeader.setZxid(replyHdr.getZxid());
                    if (replyHdr.getZxid() > 0) {
                        lastZxid = replyHdr.getZxid();
                    }
                    if (packet.response != null &&
replyHdr.getErr() == 0) {

packet.response.deserialize(bbia, "response"); //获得服
务端的响应，反序列化以后设置到 packet.response 属性
中。所以我们可以在 exists 方法的最后一行通过
packet.response 拿到改请求的返回结果
```

```
                    }

                    if (LOG.isDebugEnabled()) {
                        LOG.debug("Reading reply
sessionid:0x"
                                +
Long.toHexString(sessionId) + ", packet:: " + packet);
                    }
                } finally {
                    finishPacket(packet); // 最后调用
finishPacket 方法完成处理
                }
            }
```

finishPacket 方法

主要功能是把从 Packet 中取出对应的 Watcher 并注册到 ZKWatchManager 中去

```
private void finishPacket(Packet p) {
        int err = p.replyHeader.getErr();
        if (p.watchRegistration != null) {
            p.watchRegistration.register(err);     //将
事件注册到 zkwatchemanager 中
```

```
                watchRegistration，熟悉吗？在组装请求的时候，我们初始化了这个对象
                把 watchRegistration 子类里面的 Watcher 实例放到 ZKWatchManager 的 existsWatches 中存储起来。
            }
            //将所有移除的监视事件添加到事件队列，这样客户端能收到 "data/child 事件被移除"的事件类型
            if (p.watchDeregistration != null) {
                Map<EventType, Set<Watcher>> materializedWatchers = null;
                try {
                    materializedWatchers = p.watchDeregistration.unregister(err);
                    for (Entry<EventType, Set<Watcher>> entry : materializedWatchers.entrySet()) {
                        Set<Watcher> watchers = entry.getValue();
                        if (watchers.size() > 0) {

queueEvent(p.watchDeregistration.getClientPath(), err,
```

```
                                        watchers,
entry.getKey());
                                //    ignore    connectionloss
when removing from local
                                // session

p.replyHeader.setErr(Code.OK.intValue());
                            }
                        }
                    }                                          catch
(KeeperException.NoWatcherException nwe) {

p.replyHeader.setErr(nwe.code().intValue());
                } catch (KeeperException ke) {

p.replyHeader.setErr(ke.code().intValue());
                }
            }
            //cb 就是 AsnycCallback，如果为 null，表明是
同步调用的接口，不需要异步回掉，因此，直接 notifyAll
即可。
            if (p.cb == null) {
```

```
                synchronized (p) {
                    p.finished = true;
                    p.notifyAll();
                }
            } else {
                p.finished = true;
                eventThread.queuePacket(p);
            }
        }
```

watchRegistration

```
        public void register(int rc) {
            if (shouldAddWatch(rc)) {
                Map<String, Set<Watcher>>
watches = getWatches(rc); // //通过子类的实现取得
ZKWatchManager 中的 existsWatches
                synchronized(watches) {
                    Set<Watcher> watchers =
watches.get(clientPath);
                    if (watchers == null) {
                        watchers = new
HashSet<Watcher>();
```

```
                    watches.put(clientPath,
watchers);
                }
                watchers.add(watcher);  // 将
Watcher 对象放到 ZKWatchManager 中的
existsWatches 里面
            }
        }
    }
```

下面这段代码是客户端存储 watcher 的几个 map 集合，分别对应三种注册监听事件

```
static class ZKWatchManager implements
ClientWatchManager {
    private final Map<String, Set<Watcher>>
dataWatches =
        new HashMap<String, Set<Watcher>>();
    private final Map<String, Set<Watcher>>
existWatches =
        new HashMap<String, Set<Watcher>>();
    private final Map<String, Set<Watcher>>
childWatches =
        new HashMap<String, Set<Watcher>>();
```

总的来说，当使用 ZooKeeper 构造方法或者使用 getData、exists 和 getChildren 三个接口来向 ZooKeeper 服务器注册 Watcher 的时候，首先将此消息传递给服务端，传递成功后，服务端会通知客户端，然后客户端将该路径和 Watcher 对应关系存储起来备用。

EventThread.queuePacket()

finishPacket 方法最终会调用 eventThread.queuePacket，讲当前的数据包添加到等待事件通知的队列中

```
public void queuePacket(Packet packet) {
        if (wasKilled) {
            synchronized (waitingEvents) {
                if (isRunning)
waitingEvents.add(packet);
                else processEvent(packet);
            }
        } else {
            waitingEvents.add(packet);
        }
    }
```

事件触发

前面这么长的说明，只是为了清洗的说明事件的注册流程，最终的触发，还得需要通过事务型操作来完成

在我们最开始的案例中，通过如下代码去完成了事件的触发

```
zookeeper.setData("/mic", "1".getByte(),-1) ; //修改节点的值触发监听
```

前面的客户端和服务端对接的流程就不再重复讲解了，交互流程是一样的，唯一的差别在于事件触发了

服务端的事件响应 DataTree.setData()

```
public Stat setData(String path, byte data[], int version, long zxid,
                    long time) throws KeeperException.NoNodeException {
        Stat s = new Stat();
        DataNode n = nodes.get(path);
```

```
            if (n == null) {
                throw new
KeeperException.NoNodeException();
            }
            byte lastdata[] = null;
            synchronized (n) {
                lastdata = n.data;
                n.data = data;
                n.stat.setMtime(time);
                n.stat.setMzxid(zxid);
                n.stat.setVersion(version);
                n.copyStat(s);
            }
            // now update if the path is in a quota subtree.
            String lastPrefix =
getMaxPrefixWithQuota(path);
            if(lastPrefix != null) {
                this.updateBytes(lastPrefix, (data == null ?
0 : data.length)
                        - (lastdata == null ? 0 :
lastdata.length));
            }
```

```
            dataWatches.triggerWatch(path,
EventType.NodeDataChanged);  // 触发对应节点的
NodeDataChanged 事件
        return s;
    }
```

WatcherManager. triggerWatch

```
Set<Watcher> triggerWatch(String path, EventType
type, Set<Watcher> supress) {
        WatchedEvent e = new WatchedEvent(type,
KeeperState.SyncConnected, path); //  根据事件类型、
连接状态、节点路径创建 WatchedEvent
        HashSet<Watcher> watchers;
        synchronized (this) {
            watchers = watchTable.remove(path); //
从 watcher 表中移除 path，并返回其对应的 watcher 集
合
            if (watchers == null || watchers.isEmpty())
{
                if (LOG.isTraceEnabled()) {
```

```
ZooTrace.logTraceMessage(LOG,

ZooTrace.EVENT_DELIVERY_TRACE_MASK,
                                    "No watchers for " +
path);
                        }
                        return null;
                    }
                    for (Watcher w : watchers) { // 遍历
watcher 集合
                        HashSet<String> paths =
watch2Paths.get(w);// 根据 watcher 从 watcher 表中取
出路径集合
                        if (paths != null) {
                            paths.remove(path); //移除路径
                        }
                    }
                }
                for (Watcher w : watchers) { // 遍历 watcher
集合
                    if (supress != null && supress.contains(w))
{
```

```
                    continue;
                }
                w.process(e);  //OK，重点又来了，w.process 是做什么呢?
            }
            return watchers;
        }
```

w.process(e);

还记得我们在服务端绑定事件的时候，watcher 绑定是是什么？是 ServerCnxn， 所以 w.process(e)，其实调用的应该是 ServerCnxn 的 process 方法。而 servercnxn 又是一个抽象方法，有两个实现类，分别是：NIOServerCnxn 和 NettyServerCnxn。那接下来我们扒开 NettyServerCnxn 这个类的 process 方法看看究竟

```
public void process(WatchedEvent event) {
        ReplyHeader h = new ReplyHeader(-1, -1L, 0);
        if (LOG.isTraceEnabled()) {
            ZooTrace.logTraceMessage(LOG,
ZooTrace.EVENT_DELIVERY_TRACE_MASK,
```

```
                                                    "Deliver
event " + event + " to 0x"
                                                        +
Long.toHexString(this.sessionId)
                                                    + " through "
+ this);
            }

            // Convert WatchedEvent to a type that can
be sent over the wire
            WatcherEvent e = event.getWrapper();

            try {
                sendResponse(h,     e,     "notification");
//look，  这个地方发送了一个事件，事件对象为
WatcherEvent。完美
            } catch (IOException e1) {
                if (LOG.isDebugEnabled()) {
                    LOG.debug("Problem sending to " +
getRemoteSocketAddress(), e1);
                }
                close();
```

```
        }
    }
```

那接下里，客户端会收到这个 response，触发 SendThread.readResponse 方法

客户端处理事件响应

SendThread.readResponse

这块代码上面已经贴过了，所以我们只挑选当前流程的代码进行讲解，按照前面我们将到过的，notifacation 通知消息的 xid 为-1，意味着~直接找到-1 的判断进行分析

在下面代码标红处.

```
void readResponse(ByteBuffer incomingBuffer) throws IOException {
                ByteBufferInputStream bbis = new ByteBufferInputStream(
                        incomingBuffer);
                BinaryInputArchive bbia = BinaryInputArchive.getArchive(bbis);
                ReplyHeader replyHdr = new ReplyHeader();
```

```
                replyHdr.deserialize(bbia, "header");
                if (replyHdr.getXid() == -2) { //?
                    // -2 is the xid for pings
                    if (LOG.isDebugEnabled()) {
                        LOG.debug("Got ping response
for sessionid: 0x"
                                +
Long.toHexString(sessionId)
                                + " after "
                                + ((System.nanoTime()
- lastPingSentNs) / 1000000)
                                + "ms");
                    }
                    return;
                }
                if (replyHdr.getXid() == -4) {
                    // -4 is the xid for AuthPacket
                    if(replyHdr.getErr()                    ==
KeeperException.Code.AUTHFAILED.intValue()) {
                        state = States.AUTH_FAILED;
                        eventThread.queueEvent(   new
WatchedEvent(Watcher.Event.EventType.None,
```

```
Watcher.Event.KeeperState.AuthFailed, null) );

                }
                if (LOG.isDebugEnabled()) {
                    LOG.debug("Got auth
sessionid:0x"
                            +
Long.toHexString(sessionId));
                }
                return;
            }
            if (replyHdr.getXid() == -1) {
                // -1 means notification
                if (LOG.isDebugEnabled()) {
                    LOG.debug("Got notification
sessionid:0x"
                            +
Long.toHexString(sessionId));
                }
                WatcherEvent event = new
WatcherEvent();
```

```
                    event.deserialize(bbia,    "response");
//这个地方，是反序列化服务端的 WatcherEvent 事件。

                    // convert from a server path to a
client path
                    if (chrootPath != null) {
                        String          serverPath     =
event.getPath();

if(serverPath.compareTo(chrootPath)==0)
                                event.setPath("/");
                            else   if   (serverPath.length()   >
chrootPath.length())

event.setPath(serverPath.substring(chrootPath.length()
));
                            else {
                             LOG.warn("Got server path " +
event.getPath()
                                    + " which is too short for
chroot path "
                                        + chrootPath);
```

```
                    }
                }

                WatchedEvent we = new
WatchedEvent(event); //组装 watchedEvent 对象。
                if (LOG.isDebugEnabled()) {
                    LOG.debug("Got " + we + " for
sessionid 0x"
                            +
Long.toHexString(sessionId));
                }

                eventThread.queueEvent( we ); //通
过 eventTherad 进行事件处理
                return;
            }

            // If SASL authentication is currently in
progress, construct and
            // send a response packet immediately,
rather than queuing a
            // response as with other packets.
```

```
                if (tunnelAuthInProgress()) {
                    GetSASLRequest request = new
GetSASLRequest();
                    request.deserialize(bbia,"token");

zooKeeperSaslClient.respondToServer(request.getToke
n(),
                          ClientCnxn.this);
                    return;
                }

                Packet packet;
                synchronized (pendingQueue) {
                    if (pendingQueue.size() == 0) {
                        throw                    new
IOException("Nothing in the queue, but got "
                                + replyHdr.getXid());
                    }
                    packet = pendingQueue.remove();
                }
                /*
                 * Since requests are processed in order,
```

```
we better get a response
                    * to the first request!
                    */
                   try {
                        if (packet.requestHeader.getXid() !=
replyHdr.getXid()) {
                             packet.replyHeader.setErr(

KeeperException.Code.CONNECTIONLOSS.intValue());
                             throw new IOException("Xid out
of order. Got Xid "
                                  + replyHdr.getXid() + "
with err " +
                                  + replyHdr.getErr() +
                                  " expected Xid "
                                  +
packet.requestHeader.getXid()
                                  + " for a packet with
details: "
                                  + packet );
                        }
```

```
packet.replyHeader.setXid(replyHdr.getXid());

packet.replyHeader.setErr(replyHdr.getErr());

packet.replyHeader.setZxid(replyHdr.getZxid());
                    if (replyHdr.getZxid() > 0) {
                        lastZxid = replyHdr.getZxid();
                    }
                    if (packet.response != null &&
replyHdr.getErr() == 0) {

packet.response.deserialize(bbia, "response");
                    }

                    if (LOG.isDebugEnabled()) {
                        LOG.debug("Reading reply
sessionid:0x"
                                +
Long.toHexString(sessionId) + ", packet:: " + packet);
                    }
                } finally {
```

```
                finishPacket(packet);
            }
        }
```

eventThread.queueEvent

SendThread 接收到服务端的通知事件后，会通过调用 EventThread 类的 queueEvent 方法将事件传给 EventThread 线程，queueEvent 方法根据该通知事件，从 ZKWatchManager 中取出所有相关的 Watcher，如果获取到相应的 Watcher，就会让 Watcher 移除失效。

```
private void queueEvent(WatchedEvent event,
Set<Watcher> materializedWatchers) {
        if (event.getType() == EventType.None
&& sessionState == event.getState()) { //判断类型
            return;
        }
        sessionState = event.getState();
        final Set<Watcher> watchers;
        if (materializedWatchers == null) {
            // materialize the watchers based on
the event
            watchers =
```

```
watcher.materialize(event.getState(),
                                    event.getType(),
event.getPath());
                } else {
                        watchers = new
HashSet<Watcher>();

watchers.addAll(materializedWatchers);
                }
                //封装 WatcherSetEventPair 对象，添加到
waitngEvents 队列中
                WatcherSetEventPair pair = new
WatcherSetEventPair(watchers, event);
                // queue the pair (watch set & event) for
later processing
                waitingEvents.add(pair);
        }
```

Meterialize 方法

通过 dataWatches 或者 existWatches 或者 childWatches 的 remove 取出对应的 watch，表明客户端 watch 也是注

册一次就移除

同时需要根据 keeperState、eventType 和 path 返回应该被通知的 Watcher 集合

```
public Set<Watcher>
materialize(Watcher.Event.KeeperState state,

Watcher.Event.EventType type,
                                 String
clientPath)
        {
            Set<Watcher> result = new
HashSet<Watcher>();

            switch (type) {
            case None:
                result.add(defaultWatcher);
                boolean clear =
disableAutoWatchReset && state !=
Watcher.Event.KeeperState.SyncConnected;
                synchronized(dataWatches) {
                    for(Set<Watcher> ws:
dataWatches.values()) {
```

```
                    result.addAll(ws);
                }
                if (clear) {
                    dataWatches.clear();
                }
            }

            synchronized(existWatches) {
                for(Set<Watcher> ws:
existWatches.values()) {
                    result.addAll(ws);
                }
                if (clear) {
                    existWatches.clear();
                }
            }

            synchronized(childWatches) {
                for(Set<Watcher> ws:
childWatches.values()) {
                    result.addAll(ws);
                }
```

```
                    if (clear) {
                        childWatches.clear();
                    }
                }

                return result;
            case NodeDataChanged:
            case NodeCreated:
                synchronized (dataWatches) {
addTo(dataWatches.remove(clientPath), result);
                }
                synchronized (existWatches) {
addTo(existWatches.remove(clientPath), result);
                }
                break;
            case NodeChildrenChanged:
                synchronized (childWatches) {
addTo(childWatches.remove(clientPath), result);
                }
```

```
                    break;
                case NodeDeleted:
                    synchronized (dataWatches) {

addTo(dataWatches.remove(clientPath), result);
                    }
                    // XXX This shouldn't be needed, but
just in case
                    synchronized (existWatches) {
                        Set<Watcher>      list      =
existWatches.remove(clientPath);
                        if (list != null) {

addTo(existWatches.remove(clientPath), result);
                            LOG.warn("We      are
triggering an exists watch for delete! Shouldn't
happen!");
                        }
                    }
                    synchronized (childWatches) {

addTo(childWatches.remove(clientPath), result);
```

```
                    }
                    break;
                default:
                    String msg = "Unhandled watch
event type " + type
                            + " with state " + state + " on
path " + clientPath;
                    LOG.error(msg);
                    throw new RuntimeException(msg);
                }

            return result;
        }
    }
```

waitingEvents.add

最后一步，接近真相了

waitingEvents 是 EventThread 这个线程中的阻塞队列，很明显，又是在我们第一步操作的时候实例化的一个线程。从名字可以指导，waitingEvents 是一个待处理 Watcher 的队列，EventThread 的 run() 方法会不断从队列中取数据，交由 processEvent 方法处理：

```
        public void run() {
            try {
                isRunning = true;
                while (true) { //死循环
                    Object event =
waitingEvents.take(); //从待处理的事件队列中取出事件
                    if (event == eventOfDeath) {
                        wasKilled = true;
                    } else {
                        processEvent(event); //执行事件
处理

                    }
                    if (wasKilled)
                        synchronized (waitingEvents) {
                            if (waitingEvents.isEmpty()) {
                                isRunning = false;
                                break;
                            }
                        }
                }
            } catch (InterruptedException e) {
                LOG.error("Event thread exiting due to
```

```
interruption", e);
                }

                LOG.info("EventThread shut down for
session: 0x{}",

Long.toHexString(getSessionId()));
            }
```

ProcessEvent

由于这块的代码太长，我只把核心的代码贴出来，这里就是处理事件触发的核心代码

```
private void processEvent(Object event) {
            try {
                if (event instanceof
WatcherSetEventPair) { //判断事件类型
                    // each watcher will process the
event
                    WatcherSetEventPair pair =
(WatcherSetEventPair) event; // 得到
watcherseteventPair
                    for (Watcher watcher :
```

```
pair.watchers) { //拿到符合触发机制的所有 watcher 列表，循环进行调用
                        try {

watcher.process(pair.event);  // 调 用 客 户 端 的 回 调 process
                        } catch (Throwable t) {
                            LOG.error("Error    while calling watcher ", t);
                        }
                    }
                }
```

总结

因为时间太晚了，有些图还没话，有时间大家自己根据理解把流程图或者时序图话出来；