目录

[一、**几个重要的类** 2](#_Toc466058128)

[**二、JAVA的基础知识** 2](#_Toc466058129)

[**三、大致了解** 2](#_Toc466058130)

[**四 从入门到放弃的讲解** 3](#_Toc466058131)

[**Code1：ZK** 3](#_Toc466058132)

[**Code2：创建 Zookeeper实例,实例化ClientCnxn,实例化ClientCnxnSocketNIO** 3](#_Toc466058133)

[**Code3：实例化ClientCnxnSocketNIO (which extends ClientCnxnSocket)** 4](#_Toc466058134)

[**Code4：ClientCnxn的具体实例化** 4](#_Toc466058135)

[**Code5：SendThread的具体实例化** 5](#_Toc466058136)

[**Code6：EventThread的具体实例化** 5](#_Toc466058137)

[**Code 7：SendThread核心run流程** 5](#_Toc466058138)

[**Code 8：startConnect()** 7](#_Toc466058139)

[**Code 9： clientCnxnSocket.connect** 8](#_Toc466058140)

[**Code10：registerAndConnect()** 8](#_Toc466058141)

[**Code11：primeConnection()** 8](#_Toc466058142)

[**Code 12：doTransport()** 9](#_Toc466058143)

[**Code13：findSendablePacket()** 10](#_Toc466058144)

[**Code14：IO write** 11](#_Toc466058145)

[**Code15：createBB()** 12](#_Toc466058146)

[**Code 16：IO read** 13](#_Toc466058147)

[**Code 17： readResponse** 14](#_Toc466058148)

[**Code 18：EventThread run:** 16](#_Toc466058149)

源码:  Zookeeper 3.4.6.jar(吐血总结)

一、**几个重要的类**

1） ZookeeperMain: main函数为入口，由zkCli.sh脚本调用启动

2) ZooKeeper：客户端入口

3) ZooKeeper.SendThread: IO线程

4） ZooKeeper.EventThread: 事件处理线程，处理各类消息callback

5） ClientCnxn: 客户端与服务器端交互的主要类

6） ClientCnxnSocketNIO：继承自ClientCnxnSocket，专门处理IO, 利用JAVANIO

7） Watcher: 用于监控Znode节点

9） WatcherManager：用来管理Watcher，管理了ZK Client绑定的所有Watcher。

**二、JAVA的基础知识**

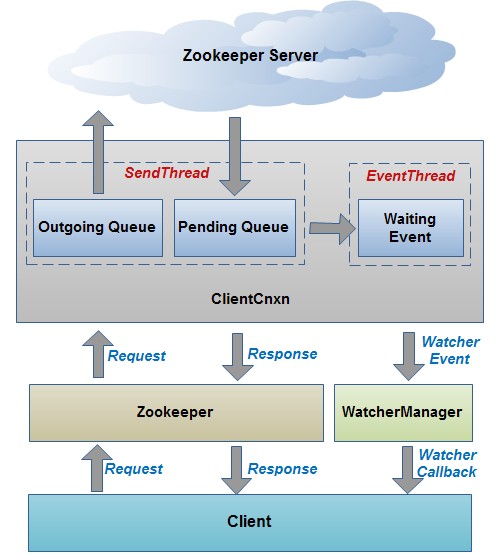
1）JAVA多线程

2）JAVANIO: 可参考：<http://blog.csdn.net/cnh294141800/article/details/52996819>

3）Socket编程（稍微了解即可）

4）JLine: 是一个用来处理控制台输入的Java类库

**三、大致了解**



上图就是对Zookeeper源码一个最好的解释，

1. Client端发送Request(封装成Packet)请求到Zookeeper
2. Zookeeper处理Request并将该请求放入Outgoing Queue(顾名思义，外出队列，就是让Zookeeper服务器处理的队列)，
3. Zookeeper端处理Outgoing Queue，并将该事件移到Pending Queue中
4. Zookeeper端消费Pending Queue，并调用finishPacket(),生成Event
5. EventThread线程消费Event事件,并且处理Watcher.

**四 从入门到放弃的讲解**

**（1）应用 提供watch实例(new MyWatcher(null))**

private class MyWatcher implements Watcher { // 默认Watcher

public void process(WatchedEvent event) {

if (getPrintWatches()) {

ZooKeeperMain.printMessage("WATCHER::");

ZooKeeperMain.printMessage(event.toString());

}

}

}

（2）**实例化zookeeper**

* 实例化socket,默认使用ClientCnxnSocketNIO
* 实例化ClientCnxn
* 实例化SendThread
* 实例化EventThread

**Code1：ZK**

zk = new ZooKeeper(host Integer.parseInt(cl.getOption("timeout")),

new MyWatcher(), readOnly); // 初始化ZK

**Code2：创建 Zookeeper实例,实例化ClientCnxn,实例化ClientCnxnSocketNIO**

public ZooKeeper(String connectString, int sessionTimeout, Watcher watcher,

boolean canBeReadOnly)

throws IOException

{

…

watchManager.defaultWatcher = watcher; // 设置defaultWatcher 为 MyWatcher

ConnectStringParser connectStringParser = new ConnectStringParser(

connectString); // 解析-server 获取 IP以及PORT

HostProvider hostProvider = new StaticHostProvider(

connectStringParser.getServerAddresses());

cnxn = new ClientCnxn(connectStringParser.getChrootPath(),

hostProvider, sessionTimeout, this, watchManager,

getClientCnxnSocket(), canBeReadOnly); // 创建 ClientCnxn实例

cnxn.start(); // 启动cnxn中的SendThread and EventThread进程

}

**Code3：实例化ClientCnxnSocketNIO (which extends ClientCnxnSocket)**

private static ClientCnxnSocket getClientCnxnSocket() throws IOException {

String clientCnxnSocketName = System

.getProperty(ZOOKEEPER\_CLIENT\_CNXN\_SOCKET);

if (clientCnxnSocketName == null) {

clientCnxnSocketName = ClientCnxnSocketNIO.class.getName();

}

try {

return (ClientCnxnSocket) Class.forName(clientCnxnSocketName)

.newInstance();

} catch (Exception e) {

IOException ioe = new IOException("Couldn't instantiate "

+ clientCnxnSocketName);

ioe.initCause(e);

throw ioe;

}

}

**Code4：ClientCnxn的具体实例化**

/\* 另一个ClientCnxn构造函数， 可见时sessionId=0

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

ClientWatchManager watcher, ClientCnxnSocket clientCnxnSocket, boolean canBeReadOnly)

throws IOException {

this(chrootPath, hostProvider, sessionTimeout, zooKeeper, watcher,

clientCnxnSocket, 0, new byte[16], canBeReadOnly);

}

\*/

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;

//主机列表

this.hostProvider = hostProvider;

this.chrootPath = chrootPath;

//连接超时

connectTimeout = sessionTimeout / hostProvider.size();

//读超时

readTimeout = sessionTimeout \* 2 / 3;

readOnly = canBeReadOnly;

//初始化client2个核心线程，SendThread是client的IO核心线程，EventThread从SendThread里拿到event

sendThread = new SendThread(clientCnxnSocket);

eventThread = new EventThread();

}

**Code5：SendThread的具体实例化**

SendThread(ClientCnxnSocket clientCnxnSocket) {

super(makeThreadName("-SendThread()"));

state = States.CONNECTING; // 将状态设置为连接状态（此时还未连接）

this.clientCnxnSocket = clientCnxnSocket;

setUncaughtExceptionHandler(uncaughtExceptionHandler);

setDaemon(true); //设为守护线程

}

**Code6：EventThread的具体实例化**

EventThread() {

super(makeThreadName("-EventThread"));

setUncaughtExceptionHandler(uncaughtExceptionHandler);

setDaemon(true);

}

**至此所有对象实例化完成，然后启动SendThread、EventThread进程**

**（3）启动zookeeper**

* + 启动SendThread
    - 连接服务器
      * 产生真正的socket，见ClientCnxnSocketNIO.createSock
* 向select注册一个OP\_CONNECT事件并连接服务器，由于是非阻塞连接，此时有可能并不会立即连上，如果连上就会调用SendThread.primeConnection初始化连接来注册读写事件，否则会在接下来的轮询select获取连接事件中处理
* 复位socket的incomingBuffer
* 连接成功后会产生一个connect型的请求发给服务，用于获取本次连接的sessionid
* 进入循环等待来自应用的请求，如果没有就根据时间来ping 服务器
  + 启动EventThread

开始进入无限循环，从队列waitingEvents中获取事件，如果没有就阻塞等待

**Code 7：SendThread核心run流程**

可以对run进行抽象看待，流程如下

loop:

- try:

- - !isConnected()

- - - connect()

- - doTransport()

- catch:

- - cleanup()

close()

先判断是否连接，没有连接则调用connect方法进行连接，有连接则直接使用；然后调用doTransport方法进行通信，若连接过程中出现异常，则调用cleanup()方法；最后关闭连接。

public void run() {

while (state.isAlive()) { // this != CLOSED && this != AUTH\_FAILED; 刚才设置了首次状态为连接状态

try {

//如果还没连上，则启动连接程序

if (!clientCnxnSocket.isConnected()) { //所有的clientCnxnSocket都是clientCnxnSocketDIO实例

//不是首次连接则休息1S

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;

}

startConnect();// 启动连接

clientCnxnSocket.updateLastSendAndHeard(); //更新Socket最后一次发送以及听到消息的时间

}

if (state.isConnected()) {

// determine whether we need to send an AuthFailed event.

if (zooKeeperSaslClient != null) {

......

}

// 下一次超时时间

to = readTimeout - clientCnxnSocket.getIdleRecv();

} else {

// 如果还没连接上 重置当前剩余可连接时间

to = connectTimeout - clientCnxnSocket.getIdleRecv();

}

// 连接超时

if (to <= 0) {

}

// 判断是否 需要发送Ping心跳包

if (state.isConnected()) {

sendPing();

}

// If we are in read-only mode, seek for read/write server

if (state == States.CONNECTEDREADONLY) {

}

// The most important step. Do real IO

clientCnxnSocket.doTransport(to, pendingQueue, outgoingQueue, ClientCnxn.this);

} catch (Throwable e) {

}

}

cleanup();

...

}

}

**Code 8：startConnect()**

// 具体实际连接部分

private void startConnect() throws IOException {

state = States.CONNECTING; //state 状态设置为连接

InetSocketAddress addr;

if (rwServerAddress != null) {

addr = rwServerAddress;

rwServerAddress = null;

} else {

addr = hostProvider.next(1000);

}

setName(getName().replaceAll("\\(.\*\\)",

"(" + addr.getHostName() + ":" + addr.getPort() + ")"));

if (ZooKeeperSaslClient.isEnabled()) {

}

logStartConnect(addr); //写连接日志

clientCnxnSocket.connect(addr); //连接Socket

}

**Code 9： clientCnxnSocket.connect**

void connect(InetSocketAddress addr) throws IOException {

SocketChannel sock = createSock(); // 创建一个非阻塞空SocketChannel

try {

registerAndConnect(sock, addr); //注册并且连接sock到辣个addr

} catch (IOException e) {

….

}

initialized = false;

/\* Reset incomingBuffer

\*/

lenBuffer.clear();

incomingBuffer = lenBuffer;

}

}

**Code10：registerAndConnect()**

void registerAndConnect(SocketChannel sock, InetSocketAddress addr)

throws IOException {

sockKey = sock.register(selector, SelectionKey.OP\_CONNECT); //将socket注册到selector中

boolean immediateConnect = sock.connect(addr); //socket连接服务器

if (immediateConnect) {

sendThread.primeConnection(); //初始化连接事件

}

}

**Code11：primeConnection()**

void primeConnection() throws IOException {

LOG.info("Socket connection established to "

+ clientCnxnSocket.getRemoteSocketAddress()

+ ", initiating session");

isFirstConnect = false; // 设置为非首次连接

long sessId = (seenRwServerBefore) ? sessionId : 0; // 客户端默认sessionid为0

// 创建连接request lastZxid 代表最新一次的节点ZXID

ConnectRequest conReq = new ConnectRequest(0, lastZxid,

sessionTimeout, sessId, sessionPasswd);

// 线程安全占用outgoing

synchronized (outgoingQueue) {

…

//组合成通讯层的Packet对象，添加到发送队列，对于ConnectRequest其requestHeader为null

outgoingQueue.addFirst(new Packet(null, null, conReq,

null, null, readOnly));

}

//确保读写事件都监听，也就是设置成可读可写

clientCnxnSocket.enableReadWriteOnly();

if (LOG.isDebugEnabled()) {

LOG.debug("Session establishment request sent on "

+ clientCnxnSocket.getRemoteSocketAddress());

}

}

至此Channelsocket已经成功连接，并且已将连接请求做为队列放到Outgoing中。此时，需要再回头看Code7, 也就是一直在循环的SendThread部分。可以看到连接部分成功完成，接下来需要做doTransport()。// CnxnClientSocketNio

**Code 12：doTransport()**

void doTransport(int waitTimeOut, List<Packet> pendingQueue, LinkedList<Packet> outgoingQueue,

ClientCnxn cnxn)

throws IOException, InterruptedException {

//select

selector.select(waitTimeOut);

Set<SelectionKey> selected;

synchronized (this) {

selected = selector.selectedKeys();

}

// Everything below and until we get back to the select is

// non blocking, so time is effectively a constant. That is

// Why we just have to do this once, here

updateNow();

for (SelectionKey k : selected) {

SocketChannel sc = ((SocketChannel) k.channel());

//如果之前连接没有立马连上，则在这里处理OP\_CONNECT事件

if ((k.readyOps() & SelectionKey.OP\_CONNECT) != 0) {

if (sc.finishConnect()) {

updateLastSendAndHeard();

sendThread.primeConnection();

}

}

//如果读写就位，则处理之

else if ((k.readyOps() & (SelectionKey.OP\_READ | SelectionKey.OP\_WRITE)) != 0) {

doIO(pendingQueue, outgoingQueue, cnxn);

}

}

if (sendThread.getZkState().isConnected()) {

synchronized(outgoingQueue) {

//找到连接Packet并且将他放到队列头

if (findSendablePacket(outgoingQueue,

cnxn.sendThread.clientTunneledAuthenticationInProgress()) != null) {

// 将要Channecl设置为可读

enableWrite();

}

}

}

selected.clear();

}

**Code13：findSendablePacket()**

private Packet findSendablePacket(LinkedList<Packet> outgoingQueue,

boolean clientTunneledAuthenticationInProgress) {

synchronized (outgoingQueue) {

..

// Since client's authentication with server is in progress,

// send only the null-header packet queued by primeConnection().

// This packet must be sent so that the SASL authentication process

// can proceed, but all other packets should wait until

// SASL authentication completes.

//因为Conn Packet需要发送到SASL authentication进行处理，其他Packet都需要等待直到该处理完成，

//Conn Packet必须第一个处理，所以找出它并且把它放到OutgoingQueue头,也就是requestheader=null的辣个

ListIterator<Packet> iter = outgoingQueue.listIterator();

while (iter.hasNext()) {

Packet p = iter.next();

if (p.requestHeader == null) {

// We've found the priming-packet. Move it to the beginning of the queue.

iter.remove();

outgoingQueue.add(0, p); // 将连接放到outgogingQueue第一个

return p;

} else {

// Non-priming packet: defer it until later, leaving it in the queue

// until authentication completes.

if (LOG.isDebugEnabled()) {

LOG.debug("deferring non-priming packet: " + p +

"until SASL authentication completes.");

}

}

}

// no sendable packet found.

return null;

}

**}**

**然后就是最重要的IO部分：**

* + **需要处理两类网络事件（读、写）**

**Code14：IO write**

if (sockKey.isWritable()) {

synchronized(outgoingQueue) {

// 获得packet

Packet p = findSendablePacket(outgoingQueue,

cnxn.sendThread.clientTunneledAuthenticationInProgress());

if (p != null) {

updateLastSend();

// If we already started writing p, p.bb will already exist

if (p.bb == null) {

if ((p.requestHeader != null) &&

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

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

//如果不是 连接事件，不是ping 事件，不是 认证时间

p.requestHeader.setXid(cnxn.getXid());

}

// 序列化

p.createBB();

}

//将数据写入Channel

sock.write(p.bb);

// p.bb中如果没有内容 则表示发送成功

if (!p.bb.hasRemaining()) {

//发送数+1

sentCount++;

//将该P从队列中移除

outgoingQueue.removeFirstOccurrence(p);

//如果该事件不是连接事件，不是ping事件，不是认证事件， 则将他加入pending队列中

if (p.requestHeader != null

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

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

synchronized (pendingQueue) {

pendingQueue.add(p);

}

}

}

}

if (outgoingQueue.isEmpty()) {

// No more packets to send: turn off write interest flag.

// Will be turned on later by a later call to enableWrite(),

// from within ZooKeeperSaslClient (if client is configured

// to attempt SASL authentication), or in either doIO() or

// in doTransport() if not.

disableWrite();

} else if (!initialized && p != null && !p.bb.hasRemaining()) {

// On initial connection, write the complete connect request

// packet, but then disable further writes until after

// receiving a successful connection response. If the

// session is expired, then the server sends the expiration

// response and immediately closes its end of the socket. If

// the client is simultaneously writing on its end, then the

// TCP stack may choose to abort with RST, in which case the

// client would never receive the session expired event. See

// http://docs.oracle.com/javase/6/docs/technotes/guides/net/articles/connection\_release.html

disableWrite();

} else {

// Just in case

enableWrite();

}

}

}

**Code15：createBB()**

public void createBB() {

try {

ByteArrayOutputStream baos = new ByteArrayOutputStream();

BinaryOutputArchive boa = BinaryOutputArchive.getArchive(baos);

boa.writeInt(-1, "len"); // We'll fill this in later

// 如果不是连接事件则设置协议头

if (requestHeader != null) {

requestHeader.serialize(boa, "header");

}

//设置协议体

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");

}

baos.close();

//生成ByteBuffer

this.bb = ByteBuffer.wrap(baos.toByteArray());

//将bytebuffer的前4个字节修改成真正的长度，总长度减去一个int的长度头

this.bb.putInt(this.bb.capacity() - 4);

//准备给后续读 让buffer position = 0

this.bb.rewind();

} catch (IOException e) {

LOG.warn("Ignoring unexpected exception", e);

}

}

**Code 16：IO read**

if (sockKey.isReadable()) {

//先从Channel读4个字节，代表头

int rc = sock.read(incomingBuffer);

if (rc < 0) {

throw new EndOfStreamException(

"Unable to read additional data from server sessionid 0x"

+ Long.toHexString(sessionId)

+ ", likely server has closed socket");

}

if (!incomingBuffer.hasRemaining()) {

incomingBuffer.flip();

if (incomingBuffer == lenBuffer) {

recvCount++;

readLength();

}

//初始化

else if (!initialized) {

readConnectResult(); // 读取连接结果

enableRead(); // Channel 可读

if (findSendablePacket(outgoingQueue,

cnxn.sendThread.clientTunneledAuthenticationInProgress()) != null) {

// Since SASL authentication has completed (if client is configured to do so),

// outgoing packets waiting in the outgoingQueue can now be sent.

enableWrite();

}

lenBuffer.clear();

incomingBuffer = lenBuffer;

updateLastHeard();

initialized = true;

} else {

// 处理其他请求

sendThread.readResponse(incomingBuffer);

lenBuffer.clear();

incomingBuffer = lenBuffer;

updateLastHeard();

}

}

}

还有一个比较关键的函数就是readResponse函数，用来消费PendingQueue,处理的消息分为三类

* + ping 消息 XID=-2
  + auth认证消息 XID=-4
  + 订阅的消息，即各种变化的通知，比如子节点变化、节点内容变化，由服务器推过来的消息 ，获取到这类消息或通过eventThread.queueEvent将消息推入事件队列

XID=-1

**Code 17： readResponse**

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");

// 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));

}

//将事件加入到 event队列中

eventThread.queueEvent( we );

return;

}

结束了IO之后就是对于事件的消费，也就是一开始图示的右半部分也是接近最后部分啦

**Code 18：EventThread run:**

public void run() {

try {

isRunning = true;

while (true) {

// 获取事件

Object event = waitingEvents.take();

if (event == eventOfDeath) {

wasKilled = true;

} else {

//处理事件

processEvent(event);

}

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");

}

}

}

**最后就是processEvent了，这个就不贴代码了（代码备注的累死了），写思路。**

ProcessEvent：

processEvent 是 EventThread 处理事件核心函数，核心逻辑如下：

1、如果 event instanceof WatcherSetEventPair ，取出 pair 中的 Watchers ，逐个调用 watcher.process(pair.event)

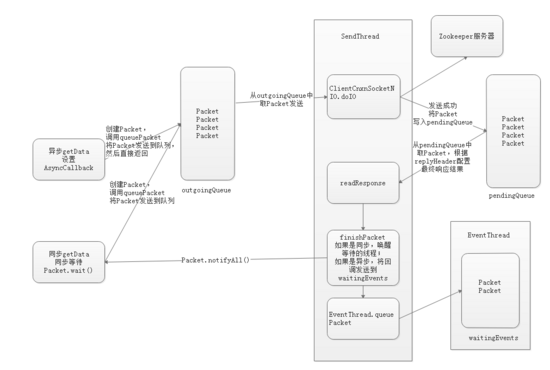
2、否则 event 为 AsyncCallback ，根据 p.response 判断为哪种响应类型，执行响应的回调 processResult 。

**Watcher 和 AsyncCallback 的区别**

Watcher： Watcher 是用于监听节点，session 状态的，比如 getData 对数据节点 a 设置了 watcher ，那么当 a 的数据内容发生改变时，客户端会收到 NodeDataChanged 通知，然后进行 watcher 的回调。

AsyncCallback : AsyncCallback 是在以异步方式使用 ZooKeeper API 时，用于处理返回结果的。例如：getData 同步调用的版本是： byte[] getData(String path, boolean watch,Stat stat) ，异步调用的版本是： void getData(String path,Watcher watcher,AsyncCallback.DataCallback cb,Object ctx) ，可以看到，前者是直接返回获取的结果，后者是通过 AsyncCallback 回调处理结果的。

**下图能完成描述流程，但是不太清晰，将就着看吧。**



**接下来就是客户端发送指令与负责端进行交互比如:**

**Ls、getChildren、getData等**

**参考文献：**

**[1]** [**http://www.cnblogs.com/davidwang456/p/5000927.html**](http://www.cnblogs.com/davidwang456/p/5000927.html)

**[2]** [**http://www.verydemo.com/demo\_c89\_i33659.html**](http://www.verydemo.com/demo_c89_i33659.html)

**[3]** [**http://blog.csdn.net/pwlazy/article/details/8000566**](http://blog.csdn.net/pwlazy/article/details/8000566)

**[4]** [**http://www.cnblogs.com/ggjucheng/p/3376548.html**](http://www.cnblogs.com/ggjucheng/p/3376548.html)

**[5]** [**http://zookeeper.apache.org/doc/r3.3.6/api/index.html**](http://zookeeper.apache.org/doc/r3.3.6/api/index.html)

**[6]** [**http://www.tuicool.com/articles/i6vMVze**](http://www.tuicool.com/articles/i6vMVze)

**[7]http://www.ibm.co**m/developerworks/cn/opensource/os-cn-apache-zookeeper-watcher/