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

基于 zkclient 客户端发起一个数据操作

```
<dependency>
    <groupId>com.101tec</groupId>
    <artifactId>zkclient</artifactId>
    <version>0.10</version>
</dependency>
```

```
public static void main( String[] args ) throws KeeperException, Interr
uptedException, IOException {
    ZooKeeper zookeeper=new ZooKeeper("192.168.13.102:2181",4000,new Wa
tcher(){
        @Override
        public void process(WatchedEvent event) {
            System.out.println("event.type"+event.getType());
        }
    });
    zookeeper.create("/watch","0".getBytes(), ZooDefs.Ids. OPEN_ACL_UNS
AFE, CreateMode.PERSISTENT); //创建节点
    zookeeper.exists("/watch",true); //注册监听
    Thread.sleep(1000);
    zookeeper.setData("/watch", "1".getBytes(),-1) ; //修改节点的值触发监
听

    System.in.read();
}
```

ZooKeeper API 的初始化过程

```
ZooKeeper zookeeper=new ZooKeeper(“192.168.11.152:2181”,4000,new Watche
r(){
      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 wat
cher,
            boolean canBeReadOnly, HostProvider aHostProvider,
            ZKClientConfig clientConfig) throws IOException {
        LOG.info("Initiating client connection, connectString=" + conne
ctString
 + " 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 ConnectStringPars
er(
                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 se
ssionTimeout, ZooKeeper zooKeeper,
            ClientWatchManager watcher, ClientCnxnSocket clientCnxnSock
et,
            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 sendThr
ead
        eventThread = new EventThread();                 --初始化 eventTh
read
        this.clientConfig=zooKeeper.getClientConfig();
    }

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

服务端接收请求处理流程

服务端有一个 NIOServerCnxn 类，用来处理客户端发送过来的请求

NIOServerCnxn

ZookeeperServer-zks.processPacket(this, bb);

处理客户端传送过来的数据包

```
public void processPacket(ServerCnxn cnxn, ByteBuffer incomingBuffer) t
hrows IOException {
        // We have the request, now process and setup for next
        InputStream bais = new ByteBufferInputStream(incomingBuffer);
        BinaryInputArchive bia = BinaryInputArchive.getArchive(bais);
        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，如果是 auth
```

```
操作，则执行下面的代码
            LOG.info("got auth packet " + cnxn.getRemoteSocketAddress
());
            AuthPacket authPacket = new AuthPacket();
            ByteBufferInputStream.byteBuffer2Record(incomingBuffer, aut
hPacket);
            String scheme = authPacket.getScheme();
            ServerAuthenticationProvider ap = ProviderRegistry.getServe
rProvider(scheme);
            Code authReturn = KeeperException.Code.AUTHFAILED;
            if(ap != null) {
                try {
                    authReturn = ap.handleAuthentication(new ServerAuth
enticationProvider.ServerObjs(this, cnxn), authPacket.getAuth());
                } catch(RuntimeException e) {
                    LOG.warn("Caught runtime exception from Authenticat
ionProvider: " + scheme + " due to " + e);
                    authReturn = KeeperException.Code.AUTHFAILED;
                }
            }
            if (authReturn == KeeperException.Code.OK) {
                if (LOG.isDebugEnabled()) {
                    LOG.debug("Authentication succeeded for scheme: " +
 scheme);
                }
                LOG.info("auth success " + cnxn.getRemoteSocketAddress
());
                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 scheme: "
                            + scheme + " has "
                            + ProviderRegistry.listProviders());
                } else {
                    LOG.warn("Authentication failed for scheme: " + sch
eme);
                }
                // send a response...
                ReplyHeader rh = new ReplyHeader(h.getXid(), 0,
                        KeeperException.Code.AUTHFAILED.intValue());
                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, KeeperE
xception.Code.OK.intValue());
                cnxn.sendResponse(rh,rsp, "response"); // not sure abou
t 3rd arg..what is it?
                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.g
etXid(),
                  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 过来以
后会经历一系列处理器的处理过程
            synchronized (this) {
                try {
                    // Since all requests are passed to the request
                    // processor it should wait for setting up the requ
est
                    // processor chain. The state will be updated to RU
NNING
                    // 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); //判断是否合
法
                if (validpacket) {
                    firstProcessor.processRequest(si);  调用 firstProcessor
发起请求，而这个 firstProcess 是一个接口，有多个实现类，具体的调用链是怎么样的？
往下看吧
                    if (si.cnxn != null) {
                        incInProcess();
                    }
                } else {
                    LOG.warn("Received packet at server of unknown type " +
 si.type);
                    new UnimplementedRequestProcessor().processRequest(si);
                }
            } catch (MissingSessionException e) {
                if (LOG.isDebugEnabled()) {
                    LOG.debug("Dropping request: " + e.getMessage());
                }
            } catch (RequestProcessorException e) {
                LOG.error("Unable to process request:" + e.getMessage(), e);
            }
        }
```

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 FinalRequestProcessor(thi
s);
        RequestProcessor syncProcessor = new SyncRequestProcessor(this,
 finalProcessor);
        ((SyncRequestProcessor)syncProcessor).start();
        firstProcessor = new PrepRequestProcessor(this, syncProcessor);
//需要注意的是，PrepRequestProcessor 中传递的是一个 syncProcessor
        ((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

\1. 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 submittedRequests = new LinkedBlockingQueue();

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);
```

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 s
ervers
            // 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("Sn
apshot Thread") {
                                    public void run() {
                                        try {
                                            zks.takeSnapshot();
                                        } catch(Exception e) {
                                            LOG.warn("Unexpected ex
ception", 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;
            }
```

总结

调用关系链如下

客户端接收服务端处理完成的响应

ClientCnxnSocketNIO.doIO

服务端处理完成以后，会通过 NIOServerCnxn.sendResponse 发送返回的响应信息，

客户端会在 ClientCnxnSocketNIO.doIO 接收服务端的返回，

注意一下 SendThread.readResponse,接收服务端的信息进行读取

```
void doIO(List<Packet> pendingQueue, LinkedList<Packet> outgoingQueue,
ClientCnxn cnxn)
      throws InterruptedException, IOException {
        SocketChannel sock = (SocketChannel) sockKey.channel();
        if (sock == null) {
            throw new IOException("Socket is null!");
        }
        if (sockKey.isReadable()) {
            int rc = sock.read(incomingBuffer);
            if (rc < 0) {
                throw new EndOfStreamException(
                        "Unable to read additional data from server ses
sionid 0x"
                                + Long.toHexString(sessionId)
                                + ", likely server has closed socket");

            }
            if (!incomingBuffer.hasRemaining()) {
                incomingBuffer.flip();
                if (incomingBuffer == lenBuffer) {
                    recvCount++;
                    readLength();
                } else if (!initialized) {
                    readConnectResult();
                    enableRead();
                    if (findSendablePacket(outgoingQueue,
                            cnxn.sendThread.clientTunneledAuthenticatio
nInProgress()) != null) {
                        // Since SASL authentication has completed (if
client is configured to do so),
                        // outgoing packets waiting in the outgoingQueu
e can now be sent.
                        enableWrite();
                    }
                    lenBuffer.clear();
                    incomingBuffer = lenBuffer;
                    updateLastHeard();
                    initialized = true;
                } else {
                    sendThread.readResponse(incomingBuffer);
                    lenBuffer.clear();
                    incomingBuffer = lenBuffer;
                    updateLastHeard();
                }
            }
        }
```

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(bbi
s);
            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) / 1
000000)
                            + "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.Ev
ent.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.getToken(),
```

```
                  ClientCnxn.this);
              return;
          }

          Packet packet;
          synchronized (pendingQueue) {
              if (pendingQueue.size() == 0) {
                  throw new IOException("Nothing in the queue, but go
t "
                          + 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 r
esponse
           * 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.intValu
e());
                  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 exis
ts 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zkwatchem
anager 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 exists
Watches 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 : materialize
dWatchers.entrySet()) {
                    Set<Watcher> watchers = entry.getValue();
                    if (watchers.size() > 0) {
                        queueEvent(p.watchDeregistration.getClientPath
(), err,
                                watchers, entry.getKey());
                        // ignore connectionloss when removing from loc
al
                        // 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 ZKWatch
Manager 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

NIOServerCnxn。那接下来我们扒开 NIOServerCnxn 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 wir
e
        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 " + getRemoteSocketAddres
s(), 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

```
void readResponse(ByteBuffer incomingBuffer) throws IOException {
            ByteBufferInputStream bbis = new ByteBufferInputStream(
                    incomingBuffer);
            BinaryInputArchive bbia = BinaryInputArchive.getArchive(bbi
s);
            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) / 1
000000)
                            + "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.Ev
ent.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.l
ength()));
                    else {
                        LOG.warn("Got server path " + event.getPath()
                                + " which is too short for chroot p
ath "
                                + chrootPath);
```

```
                }
            }

            WatchedEvent we = new WatchedEvent(event); //组装watche
dEvent 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, constru
ct 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.getToken(),
              ClientCnxn.this);
            return;
        }

        Packet packet;
        synchronized (pendingQueue) {
            if (pendingQueue.size() == 0) {
                throw new IOException("Nothing in the queue, but go
t "
                        + replyHdr.getXid());
            }
            packet = pendingQueue.remove();
        }
        /*
         * Since requests are processed in order, we better get a r
esponse
         * to the first request!
         */
        try {
            if (packet.requestHeader.getXid() != replyHdr.getXid())
 {
                packet.replyHeader.setErr(
                        KeeperException.Code.CONNECTIONLOSS.intValu
e());
                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> materializedWa
tchers) {
            if (event.getType() == EventType.None && sessionState == ev
ent.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 != Watch
er.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) even
t;  //得到watchersteventPair
                  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);
                    }
                }
            }
```

服务端接收数据请求

服务端收到的数据包应该在哪里呢？在上节课分析过了，zookeeper 启动的时候，通过下面的代码构建了一个

ServerCnxnFactory cnxnFactory = ServerCnxnFactory.*createFactory*();

NIOServerCnxnFactory，它实现了 Thread，所以在启动的时候，会在 run 方法中不断循环接收客户端的请求进行分发

NIOServerCnxnFactory.run

```
public void run() {
    while (!ss.socket().isClosed()) {
        try {
            for (SelectionKey k : selectedList) {
// 获取client 的连接请求
                if ((k.readyOps() & SelectionKey.OP_ACCEPT) != 0) {

                } else if ((k.readyOps() & (SelectionKey.OP_READ | Sele
ctionKey.OP_WRITE)) != 0) {
                    //处理客户端的读/写请求
NIOServerCnxn c = (NIOServerCnxn) k.attachment();
                    c.doIO(k);//处理IO 操作
                } else {
                    if (LOG.isDebugEnabled()) {
                        LOG.debug("Unexpected ops in select "
                                + k.readyOps());
                    }
                }
            }
            selected.clear();
        } catch (RuntimeException e) {
            LOG.warn("Ignoring unexpected runtime exception", e);
        } catch (Exception e) {
            LOG.warn("Ignoring exception", e);
        }
    }
```

```
    closeAll();
    LOG.info("NIOServerCnxn factory exited run method");
}
```

NIOServerCnxn.doIO

```
void doIO(SelectionKey k)
try {
    //省略部分代码..
    if (k.isReadable()) {//处理读请求，表示接收
        //中间这部分逻辑用来处理报文以及粘包问题
            if (isPayload) { // not the case for 4letterword
                readPayload();//处理报文
            }
            else {
                // four letter words take care
                // need not do anything else
                return;
            }
        }
    }
```

NIOServerCnxn.readRequest

读取客户端的请求，进行具体的处理

```
private void readRequest() throws IOException {
    zkServer.processPacket(this, incomingBuffer);
}
```

ZookeeperServer.processPacket

这个方法根据数据包的类型来处理不同的数据包，对于读写请求，我们主要关注下面这块代码即可

```
Request si = new Request(cnxn, cnxn.getSessionId(), h.getXid(),
  h.getType(), incomingBuffer, cnxn.getAuthInfo());
si.setOwner(ServerCnxn.me);
submitRequest(si);
```

后续的流程，在前面的源码分析中有些，就不做重复黏贴了。

集群模式下的处理流程

集群模式下，涉及到 zab 协议，所以处理流程比较复杂，大家可以基于这个图来定位代码的流程