messaging 通信机制

公共接口

IContext

定义了创建和关闭Socket连接的方法：

- `bind()`用于绑定到一个端口号并返回一个Socket对象，该Socket对象主要负责接收消息；
- `connect()`用于连接到一个端口号并返回该端口号， 主要负责发送消息。
- `term()`用来关闭连接

IConnection

用于接收和发送消息

- `registerRecv()`注册一个回调函数，当数据准备好处理时通知它。
- `sendLoadMetrics()`将负载指标发送给所有下游连接。
- `send()`发送一个或者一批带有taskId和负载信息的消息
- `getLoad()`通过给定的taskId获取负载信息
- `getPort()`获取当前连接对象的端口好
- `close()`关闭此连接

IConnectionCallback

回调类，当TaskMassage到达时，调用

- `recv()处理到达的一批新的消息。

进程内通信-Context

进程内部通信主要使用Context类，该类实现了IContext接口

除了实现IContext的方法之外，还具有两个内部类LocalServer和LocalClient，这两个内部类实现了IConnection。

也就是说本地环境的通信采用了Server->Client的方式。

1. LocalClient在构造函数初始化时就创建了定时任务，不断使用flushPending刷新队列，向LocalServer发送空的**TaskMassage**{taskId,message}；
2. 如果有具体数据时，首先使用flushPending刷新队列，然后LocalClient调用send方法向LocalServer发送具体的TaskMassage。
3. LocalServer本身并不处理数据，他有个回调成员类IConnectionCallback，起具体实现是DeserializingConnectionCallback。LocalClient调用其recv方法来对数据进行反序列化，然后构成广播元组**AddressedTuple**{taskId,message.length}；
4. 广播元组AddressedTuple将仍然给回调函数来发送，这里的回调函数是ILocalTransferCallback.transfer，在daemon.worker.WorkerState类中，回调主要是使用WorkerState的transferLocal方法。（原因：DeserializingConnectionCallback是被WorkerState初始化的，transfer则是被transferLocal覆盖）。

先看WorkerState的注册回调函数：

```
public void registerCallbacks() {
    LOG.info("Registering IConnectionCallbacks for {}:{}", assignmentId, port);
    receiver.registerRecv(new DeserializingConnectionCallback(topologyConf,
                getWorkerTopologyContext(),
                this::transferLocal));
    //这里的this就是WorkerState
}
```

再来看DeserializingConnectionCallback构造函数

```
public DeserializingConnectionCallback(final Map conf, final GeneralTopologyContext context,
WorkerState.ILocalTransferCallback callback) {
    this.conf = conf;
    this.context = context;
    cb = callback;
    sizeMetricsEnabled =
ObjectReader.getBoolean(conf.get(Config.TOPOLOGY_SERIALIZED_MESSAGE_SIZE_METRICS), false);
}
```

可以看到有个非常巧妙的回调设计，不仅用this代替了WorkerState来实现WorkerState.ILocalTransferCallback，而且使用this::transferLocal这种java8风格的表达式在接口的方法中调用了transferLocal方法。

实际实现是这样的：

```
new DeserializingConnectionCallback(topologyConf,getWorkerTopologyContext(), new
ILocalTransferCallback() {
            @Override
            public void transfer(List<AddressedTuple> tupleBatch) {
                transferLocal(tupleBatch);
            }
        })
```

在WorkerState.transferLocal方法中有个taskToShortExecutor变量，结构为`Map<taskId,start_task_id>`，表示当有一批的tuple要处理时，找到该批次数据的起始元组的taskId，放入到shortExecutorReceiveQueueMap变量中，结构为`Map<start_task_id,DisruptorQueue>`，DisruptorQueue存的是该批次的所有元组。存储的方法是publish()。

最后梳理一下这个流程：

1. 在Worker进程内部，存在不同的线程Executor，一个线程作为LocalClient，另外一个线程作为LocalServer；
2. LocalClient在向LocalServer发送数据的过程其实就是往WorkerState（Worker进程）中的shortExecutorReceiveQueueMap变量存入批数据，这个存入就是调用DisruptorQueue.publish方法；
3. 有存入就有读取，读取的方法是DisruptorQueue.consumeBatch，这主要出现SpoutExecutor和BoltExecutor中；
4. 因此，从宏观层面来看，进程内部的各线程通信是使用DisruptorQueue完成的。

进程间通信

在daemon.Worker类的start方法内部有判断当前是否为分布式模式：`ConfigUtils.isLocalMode(conf)`

```
for (List<long> e : workerState.getExecutors()) {
    if (ConfigUtils.isLocalMode(topologyConf)) {
        newExecutors.add(LocalExecutor.mkExecutor(workerState, e, initCreds).execute());
    } else {
        newExecutors.add(Executor.mkExecutor(workerState, e, initCreds).execute());
    }
}
```

可以看到，对于是否为分布式做了分情况考虑。

本地模式的进程间通信-LocalExecutor

在storm.server.~.daemon.supervisor.ContainerLauncher.make方法：

```
    /**
     * Factory to create the right container launcher
     * for the config and the environment.
     * @param conf the config
     * @param supervisorId the ID of the supervisor
     * @param sharedContext Used in local mode to let workers talk together without netty
     * @return the proper container launcher
     * @throws IOException on any error
     */
    public static ContainerLauncher make(Map<String, Object> conf, String supervisorId, IContext
sharedContext) throws IOException {
        if (ConfigUtils.isLocalMode(conf)) {
            return new LocalContainerLauncher(conf, supervisorId, sharedContext);
        }
```

其中提到了sharedContext参数用于本地模式让进程之间不用netty通信。

思路

在daemon.worker.Worker.start()中存在以下代码：

```
//这里是指定一个变量，是EventHandler接口，(packets, seqId, batchEnd)也就是其方法onEvent的三个参数，然
后用sendTuplesToRemoteWorker实现onEvent；
EventHandler<Object> tupleHandler = (packets, seqId, batchEnd) -> workerState
                    .sendTuplesToRemoteWorker((HashMap<Integer, ArrayList<TaskMessage>>)
packets, seqId, batchEnd);
//使用Iconnection发送数据
// This thread will publish the messages destined for remote tasks to remote connections
//此线程将发布远程任务指定的消息到远程连接。
transferThread = Utils.asyncLoop(() -> {
    //重复调用的线程，不断消费给定数据
    workerState.transferQueue.consumeBatchWhenAvailable(tupleHandler);
    //传入了(packets, seqId, batchEnd)参数值，其实是DisruptorQueue的缓存的位置之类的参数
    return 0L;
}
);
```

这里涉及到一个sendTuplesToRemoteWorker，主要是完成将数据进行发送，这种发送方式是需要通过实现IConnection接口的类方法send来实现，而consumeBatchWhenAvailable内部其实使用了consumeBatchToCursor，如下所示：

```
private void consumeBatchToCursor(long cursor, EventHandler<Object> handler) {
    for (long curr = _consumer.get() + 1; curr <= cursor; curr++) {
        try {
                ...
                handler.onEvent(o, curr, curr == cursor);
                ...
}
```

这里使用的onEvent方法就是由workerState.sendTuplesToRemoteWorker实现的。只是发送的是高速缓存队列中的批数据，因为看方法的参数知道，有一些跟游标有关的参数。

再回过头来看transferThread变量其实就是一个线程，不停的调用IConnection.send向远程发送数据。

问题？这个远程怎么定义：如果不是分布式呢

追溯sendTuplesToRemoteWorker方法

```
public void sendTuplesToRemoteWorker(HashMap<Integer, ArrayList<TaskMessage>> packets, long
seqId, Boolean batchEnd) {
    drainer.add(packets);
    if (batchEnd) {
        ReentrantReadWriteLock.ReadLock readLock = endpointSocketLock.readLock();
        try {
            readLock.lock();
            drainer.send(cachedTaskToNodePort.get(), cachedNodeToPortSocket.get());
        }
        finally {
            readLock.unlock();
        }
        drainer.clear();
    }
}
```

看到其中有具体的drainer.send方法，这个方法需要需要两个参数，关心的参数是`connections<hostport,IConnection>`。

```
public void send(Map<Integer, NodeInfo> taskToNode, Map<NodeInfo, IConnection> connections) {
    HashMap<NodeInfo, ArrayList<ArrayList<TaskMessage>>> bundleMapByDestination =
groupBundleByDestination(taskToNode);
    for (Map.Entry<NodeInfo, ArrayList<ArrayList<TaskMessage>>> entry :
bundleMapByDestination.entrySet()) {
        NodeInfo hostPort = entry.getKey();
        IConnection connection = connections.get(hostPort);
        if (null != connection) {
            ArrayList<ArrayList<TaskMessage>> bundle = entry.getValue();
            Iterator<TaskMessage> iter = getBundleIterator(bundle);

            if (null != iter && iter.hasNext()) {
```

```
                connection.send(iter);//发送数据
            }
        } else {
            LOG.warn("Connection is not available for hostPort {}", hostPort);
        }
    }
}
```

可以看到其中有个connection.send方法，是在向远程发送数据的，其中还提到了端口号。

那么问题就是：谁实现了IConnection。

再看WorkerState.sendTuplesToRemoteWorker方法，需要cachedNodeToPortSocket参数，这个参数的赋值是在WorkerState.refreshConnections方法中：

```
// Add new connections atomically
cachedNodeToPortSocket.getAndUpdate(prev -> {
    Map<NodeInfo, IConnection> next = new HashMap<>(prev);
    for (NodeInfo nodeInfo : newConnections) {

 next.put(nodeInfo,mqContext.connect(topologyId,assignment.get_node_host().get(nodeInfo.get_node
()),// Host
        nodeInfo.get_port().iterator().next().intValue()));// Port
    }
    return next;
}
);
```

关键变量为mqContext，是它的connect方法放回IConnection类型数据。就如同IConnection所说的：

```
//This method establish a client side connection to a remote server
public IConnection connect(String storm_id, String host, int port);
```

connect方法创建了一个客户端侧的连接到远程服务端。但是这个方法又是在IContext接口中，使用idea的ctrl+H查看发现只有两种类实现了IContext：

1. messaging.local.Context;
2. messaging.netty.Context;

这里我们先不考虑具体的发送是怎么实现的，先考虑一个问题，在什么情况下使用local和netty？

在WorkerState的构造函数中有这样一个语句：

```
this.mqContext = (null != mqContext) ? mqContext : TransportFactory.makeContext(topologyConf);
```

定位到TransportFactory.makeContext:

```
public static IContext makeContext(Map<String, Object> topoConf) {
    //get factory class name

    String transport_plugin_klassName = (String)topoConf.get(Config.STORM_MESSAGING_TRANSPORT);
```

```
    LOG.info("Storm peer transport plugin:"+transport_plugin_klassName);
    IContext transport;
    try {
        //create a factory class
        Class klass = Class.forName(transport_plugin_klassName);
        //obtain a context object
        Object obj = klass.newInstance();
        if (obj instanceof IContext) {
            //case 1: plugin is a IContext class
            transport = (IContext)obj;
            //initialize with storm configuration
            transport.prepare(topoConf);
...
}
```

利用了反射的方式newInstance了一个IContext，名字来源是Config.STORM_MESSAGING_TRANSPORT：

```
//The transporter for communication among Storm tasks
public static final String STORM_MESSAGING_TRANSPORT = "storm.messaging.transport";
```

在storm.conf包的Default.yaml中有：

```
storm.messaging.transport: "org.apache.storm.messaging.netty.Context"
```

也就是说这里指定了使用netty.Context实现了IContext接口。

是否STORM_MESSAGING_TRANSPORT的配置和topoConf有关？

这个topoConf是由Worker的start方法传入的，那么Worker的conf又是谁传入的？跟踪发现其构造函数在LocalContainer.launch方法中被调用：

```
public void launch() throws IOException {
    Worker worker = new Worker(_conf, _sharedContext, _topologyId, _supervisorId, _port,
_workerId);
    try {
        worker.start();
    }
    catch (Exception e) {
        throw new IOException(e);
    }
    saveWorkerUser(System.getProperty("user.name"));
    ProcessSimulator.registerProcess(_workerId, worker);
    _isAlive = true;
}
```

同时，不仅构造函数，而且start()方法也是在其中被调用。LocalContainer类实现了Container，它是一个可以让worker在其中运行的容器。有local配置，是否有分布式配置？在ContainerLauncher类中发现make方法为：

```
public static ContainerLauncher make(Map<String, Object> conf, String supervisorId, IContext
```

```
sharedContext) throws IOException {
    if (ConfigUtils.isLocalMode(conf)) {
        return new LocalContainerLauncher(conf, supervisorId, sharedContext);
    }
    ResourceIsolationInterface resourceIsolationManager = null;
    if (ObjectReader.getBoolean(conf.get(DaemonConfig.STORM_RESOURCE_ISOLATION_PLUGIN_ENABLE),
false)) {
        resourceIsolationManager = ReflectionUtils.newInstance((String)
conf.get(DaemonConfig.STORM_RESOURCE_ISOLATION_PLUGIN));
        resourceIsolationManager.prepare(conf);
        LOG.info("Using resource isolation plugin {} {}",
conf.get(DaemonConfig.STORM_RESOURCE_ISOLATION_PLUGIN), resourceIsolationManager);
    }
    if (ObjectReader.getBoolean(conf.get(Config.SUPERVISOR_RUN_WORKER_AS_USER), false)) {
        return new RunAsUserContainerLauncher(conf, supervisorId, resourceIsolationManager);
    }
    return new BasicContainerLauncher(conf, supervisorId, resourceIsolationManager);
}
```

可以发现，对于是否为本地模式，进行了区分但是只有在本地模式中，也就是LocalContainerLauncher方法中有初始化Worker类，但是分布式模式中没有找到初始化Worker类。原因是这样的：BasicContainerLauncher调用了BasicContainer.launch方法：

```
@Override
    public void launch() throws IOException {
    ...
    //构造启动workers的shell命令
    List<String> commandList = mkLaunchCommand(memOnHeap, stormRoot, jlp);
    LOG.info("Launching worker with command: {}. ", ServerUtils.shellCmd(commandList));
    String workerDir = ConfigUtils.workerRoot(_conf, _workerId);
    //用构造好的shell命令启动Worker进程
    launchWorkerProcess(commandList, topEnvironment, logPrefix, processExitCallback, new
File(workerDir));
}
```

再来看看是怎么构造shell命令的：

```
private List<String> mkLaunchCommand(final int memOnheap, final String stormRoot,
            final String jlp) throws IOException {
    final String javaCmd = javaCmd("java");
    ...
    //Worker Command...
    commandList.add(javaCmd);
    commandList.add("-server");
    commandList.addAll(commonParams);
    commandList.addAll(substituteChildopts(_conf.get(Config.WORKER_CHILDOPTS), memOnheap));
    commandList.addAll(substituteChildopts(_topoConf.get(Config.TOPOLOGY_WORKER_CHILDOPTS),
memOnheap));
    commandList.addAll(substituteChildopts(Utils.OR(
                    _topoConf.get(Config.TOPOLOGY_WORKER_GC_CHILDOPTS),
                    _conf.get(Config.WORKER_GC_CHILDOPTS)), memOnheap));

    commandList.addAll(getWorkerProfilerChildOpts(memOnheap));
```

```
    commandList.add("-Djava.library.path=" + jlp);
    commandList.add("-Dstorm.conf.file=" + topoConfFile);
    commandList.add("-Dstorm.options=" + stormOptions);
    commandList.add("-Djava.io.tmpdir=" + workerTmpDir);
    commandList.addAll(classPathParams);
    commandList.add(getWorkerMain(topoVersion));
    commandList.add(_topologyId);
    commandList.add(_supervisorId);
    commandList.add(String.valueOf(_port));
    commandList.add(_workerId);
    return commandList;
}
```

可以看到它使用了java -server命令，还有getWorkerMain方法获取Worker类的main方法来运行Worker。那么在Worker的main方法中就有worker.start()。

问题解决

综上所述，并没有发现Worker与Worker之间的通信有区分本地模式和分布式模式，他们都是使用的Netty作为Worker之间的通信。而对于更加细致的Executor来说则是使用高速缓存队列来通信。Worker主要体现在通过Netty发送数据，使用Netty的Client到Server，Server接受到了数据，就根据注册的回调函数（这个回调函数的注册靠Worker类完成），sever.received->enqueue:

```
//enqueue a received message 将收到的信息入队
protected void enqueue(List<TaskMessage> msgs, String from) throws InterruptedException {
    if (null == msgs || msgs.size() == 0 || closing) {
        return;
    }
    addReceiveCount(from, msgs.size());
    if (_cb != null) {
        _cb.recv(msgs);
    }
}
```

这里调用二楼回调函数的recv方法，而这个回调，正是DeserializingConnectionCallback类，与第一部分的进程内通信的回调是一个类。

把数据写入到高速缓存队列DisruptorQueue中，那么task怎么使用数据，就是对DisruptorQueue的处理的，属于进程内部的通信，仍然是LocalClient和LocalServer这种机制。