# Storm源码走读笔记

徽沪一郎
hsxupeng@gmail.com

May 28, 2014

# 目录

## 1 楔子

为什么要对storm源码进行分析？ real-time bigdata analytic是一个非常热门的话题，起步不久，所以想在比较短的时间内深入的了解这个领域的问题及其解决方案。

# Part I
# 启动场景分析

## 2 nimbus启动场景分析

storm cluster中有以下各种进程

1. nimbus
2. zookeeper
3. supervisor
4. worker

进程之间的逻辑关系如下图所示，其中executor是运行于进程worker中 的线程用于执行spout或是bolt.

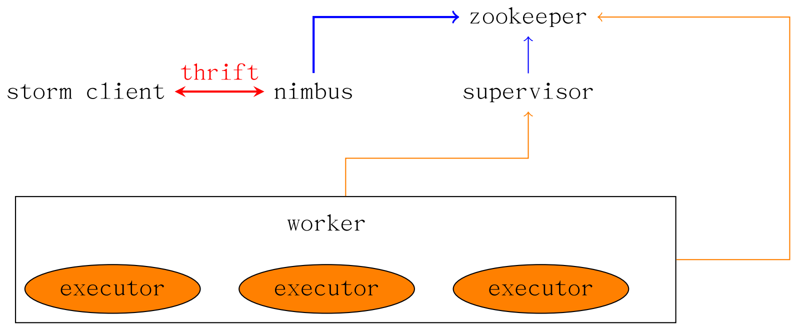


Figure 1: Storm Cluster Architecture

由Figure 1可以看出， Nimbus主要的工作就是用来处理thrift接口和zookeeper 接口上的消息与事件。本文试图阐述在nimbus启动之际在这两个接口上都需要做 哪些初始化的工作。

### 2.1 程序入口

入口函数main定义入下

```
(defn -main []
  (-launch (standalone-nimbus)))
```

注意在clojure中的函数命名规范，-functionname表示该函数是public的，如上面的-main,调用该函数的时候，不需要加-,使用main即可。

而与此相对的是defn-，这个表示该函数是私有函数，不可在外部调用。

standalone-nimbus用以实现INimbus接口。

```
(defn standalone-nimbus []
  (reify INimbus
  (prepare [this conf local-dir]
    )
  (allSlotsAvailableForScheduling
        [this supervisors topologies topologies-missing-assignments]
    (->> supervisors
         (mapcat (fn [^SupervisorDetails s]
                   (for [p (.getMeta s)]
                     (WorkerSlot. (.getId s) p))))
         set ))
  (assignSlots [this topology slots]
    )
  (getForcedScheduler [this]
    nil )
  (getHostName [this supervisors node-id]
    (if-let [^SupervisorDetails supervisor (get supervisors node-id)]
      (.getHost supervisor)))
  ))

(defn -launch [nimbus]
    (launch-server! (read-storm-config) nimbus))
```

## 2.2 配置文件读取

阅读源码其实都会遵循一个范式，那就是程序的入口在哪，配置文件是在什么时候读入的。那么好，现在就来讲配置参数的读入，在上面的-launch函数中，已经可以见到用以读取配置文件的函数了，那就是read-storm-config。

非常狗血的是，在nimbus.clj中有一个名称非常类似的函数称为read-storm-conf，这个可不是来读取storm cluster的配置信息，它其实是用来读取Topology的配置内容的。

read-storm-config定义于config.clj中,此时你会说等等，没见到有地方import或是use backtype.storm.config啊。这一切都被包装了，它们统统被放到bootstrap.clj中了。

注意到这行没

```
(bootstrap)
```

好了，上述有关文件引用的疑问解决之后，还是回到正题，看看read-storm-config的定义吧。storm默认的配置文件使用的是yaml格式，一定要找到使用yaml parser的地方。

```
(defn read-storm-config []
  (let [
        conf (clojurify-structure (Utils/readStormConfig))]
    (validate-configs-with-schemas conf)
    conf))
```

真正实现对配置文件storm.yaml进行读取的是由java代码来实现的，read-StormConfig定义于Utils.java中。

```
    public static Map readStormConfig() {
        Map ret = readDefaultConfig();
        String confFile = System.getProperty("storm.conf.file");
        Map storm;
        if (confFile==null || confFile.equals("")) {
            storm = findAndReadConfigFile("storm.yaml", false);
        } else {
            storm = findAndReadConfigFile(confFile, true);
        }
        ret.putAll(storm);
        ret.putAll(readCommandLineOpts());
        return ret;
    }

     public static Map findAndReadConfigFile(String name, boolean mustExist) {
        try {
            HashSet<URL> resources = new HashSet<URL>(findResources(name));
            if(resources.isEmpty()) {
              if(mustExist) throw
              new RuntimeException("Could not find config file on classpath " + name);
              else return new HashMap();
            }
            if(resources.size() > 1) {
              throw
              new RuntimeException("Found multiple "
              + name
              + " resources. You're probably bundling the Storm jars with your topology jar. "
              + resources);
            }
            URL resource = resources.iterator().next();
            Yaml yaml = new Yaml();
            Map ret = (Map) yaml.load(new InputStreamReader(resource.openStream()));
            if(ret==null) ret = new HashMap();


            return new HashMap(ret);

        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }


public static Map findAndReadConfigFile(String name) {
  return findAndReadConfigFile(name, true);
}
```

终于看到神秘的Yaml了,那么Yaml这个类又是由谁提供的呢，看看Utils.java

的 开头部分有这么一句话。

```
import org.yaml.snakeyaml.Yaml;
```

再看看在storm-core/project.clj中定义的dependencies,

```
[org.yaml/snakeyaml "1.11"]
```

至此，yaml文件的解析及其依赖关系的解决探索完毕。

### 2.2.1 配置文件的内容

配置文件到底长的是个啥样子呢，让我们来一探究竟。下面就是一份随storm源码发布的storm.yaml

```
########### These MUST be filled in for a storm configuration
 storm.zookeeper.servers:
     - "localhost"
#     - "server2"
#
 nimbus.host: "localhost"
#
#
# ##### These may optionally be filled in:
#
## List of custom serializations
# topology.kryo.register:
#     - org.mycompany.MyType
#     - org.mycompany.MyType2: org.mycompany.MyType2Serializer
#
## List of custom kryo decorators
# topology.kryo.decorators:
#     - org.mycompany.MyDecorator
#
## Locations of the drpc servers
# drpc.servers:
#     - "server1"
#     - "server2"

## Metrics Consumers
# topology.metrics.consumer.register:
#   - class: "backtype.storm.metrics.LoggingMetricsConsumer"
#     parallelism.hint: 1
#   - class: "org.mycompany.MyMetricsConsumer"
#     parallelism.hint: 1
#     argument:
#       - endpoint: "metrics-collector.mycompany.org"
 java.library.path: "/usr/local/lib:/usr/local/share/java"
```

```
supervisor.slots.ports:
  - 6700
  - 6701
```

在配置文件中需要至少回答以下三个问题

1. zookeeper server在哪台机器上运行，具体就来说就是ip地址啦
2. nimbus在哪运行，可以填写ip地址或域名
3. 在每台supervisor运行的机器上可以启几个slot，指定这些slot监听的端 口号

## 2.3 thrift

网络结点之间的消息交互一般会牵涉到两个基本的问题，

- 消息通道的建立
- 消息的编解码

如果每变化一个需求就手工来重写一次，一是繁琐，二是易错。为了一劳永逸的 解决此类问题，神一样的工具就出现了，如google proto1buffer,如thrift.

thrift的使用步骤如下

1. 编写后缀名为thrift的文件，使用工具生成对应语言的源码，thrift支持 的语言很多的，什么c,c++,java,python等，统统不是问题。
2. 实现thrift client
3. 实现thrift server

thrift server需要实现thrift文件中定义的service接口。更为具体的信息可以 通过阅读thrift.apache.org中的文档来获得。

有了thrift这个背景，我们再重新拾起上述的代码执行路径。上头讲到程序执行 至。

```
(defn -launch [nimbus]
  (launch-server! (read-storm-config) nimbus))
```

我们来仔细看看1aunch-sever!这个看起来名字怪怪的函数都要弄些啥东东。

```
(defn launch-server! [conf nimbus]
  (validate-distributed-mode! conf)
  (let [service-handler (service-handler conf nimbus)
        options (-> (TNonblockingServerSocket. (int (conf NIMBUS-THRIFT-PORT)))
                    (THsHaServer$Args.)
                    (.workerThreads 64)
                    (.protocolFactory (TBinaryProtocol$Factory.))
                    (.processor (Nimbus$Processor. service-handler))
                    )
       server (THsHaServer. options)]
    (.addShutdownHook (Runtime/getRuntime) (Thread. (fn [] (.shutdown service-handler) (.stop server))))
    (log-message "Starting Nimbus server...")
    (.serve server)))
```

launch-server!说白了，就是让nimbus作为一个thrift server运行起来，那么 storm.thrift中service指定的各个接口函数实现在service-handler中完成。

这里是存在包的依赖问题的，ThsHaServer是libthrift提供的，所以打开storm-core/project.clj文件可以看到这么一行来说明库的依赖。

```
[storm/libthrift7 "0.7.0-2" :exclusions [org.slf4j/slf4j-api]
```

service-handler可是一个大家伙，具体代码就不罗列了。对比一下 service-handler可以发现，在storm.thrift中的定义的Nimbus服务，其接口在 service-handler中一一得以实现。 以下是storm.thrift中关于service Nimbus的声明。

```
service Nimbus {
  void submitTopology(1: string name, 2: string uploadedJarLocation, 3:
       string jsonConf, 4: StormTopology topology) throws (1:
      AlreadyAliveException e, 2: InvalidTopologyException ite);
  void submitTopologyWithOpts(1: string name, 2: string
      uploadedJarLocation, 3: string jsonConf, 4: StormTopology
      topology, 5: SubmitOptions options) throws (1:
      AlreadyAliveException e, 2: InvalidTopologyException ite);
  void killTopology(1: string name) throws (1: NotAliveException e);
  void killTopologyWithOpts(1: string name, 2: KillOptions options)
      throws (1: NotAliveException e);
  void activate(1: string name) throws (1: NotAliveException e);
  void deactivate(1: string name) throws (1: NotAliveException e);
  void rebalance(1: string name, 2: RebalanceOptions options) throws
      (1: NotAliveException e, 2: InvalidTopologyException ite);

  // need to add functions for asking about status of storms, what
      nodes they're running on, looking at task logs

  string beginFileUpload();
  void uploadChunk(1: string location, 2: binary chunk);
  void finishFileUpload(1: string location);

  string beginFileDownload(1: string file);
  //can stop downloading chunks when receive 0-length byte array back
  binary downloadChunk(1: string id);

  // returns json
  string getNimbusConf();
  // stats functions
  ClusterSummary getClusterInfo();
  TopologyInfo getTopologyInfo(1: string id) throws (1:
      NotAliveException e);
  //returns json
  string getTopologyConf(1: string id) throws (1: NotAliveException e);
  StormTopology getTopology(1: string id) throws (1: NotAliveException
      e);
  StormTopology getUserTopology(1: string id) throws (1:
      NotAliveException e);
}
```

到这里nimbus就启动起来，问题是没见到nimbus是如何与zookeeper cluster建立连接啊，这个放在下回分析。

## 2.4 zookeeper

在上节中讲到了service-handler,此函数在开头处就是要建立与zookeeperserver之间的通讯，不过代码写的很隐蔽。

zookeeper提供的原生java api用于实际场景中对于开发者要求很高，为了减少开发周期，storm中使用了curator来与zookeeper server进行交互。

zookeeper.clj中的mk-client函数就是初始化Curator即建立与zookeeper server的连接。

```
(defnk mk-client [conf servers port :root "" :watcher default-watcher :auth-conf nil]
  (let [fk (Utils/newCurator conf servers port root (when auth-conf (ZookeeperAuthInfo. auth-conf)))]
    (.. fk
        (getCuratorListenable)
        (addListener
         (reify CuratorListener
           (^void eventReceived [this ^CuratorFramework _fk ^CuratorEvent e]
             (when (= (.getType e) CuratorEventType/WATCHED)
               (let [^WatchedEvent event (.getWatchedEvent e)]
                 (watcher (zk-keeper-states (.getState event))
                          (zk-event-types (.getType event))
                          (.getPath event))))))))
(.start fk)
    fk))
```

nimbus中的service-handler是如何与zk/mk-client建立联系的呢，其调用关系如下图所示。

```
service-handler
          └── nimbus-data
                      └── mk-storm-cluster-state
                                        └── mk-distributed-cluster-state
                                                              └── zk/mk-client
```

Figure 2: call tree for mk-client

需要注意的是，nimbus中并没有注册任何回调函数来处理zookeeper送来的notification。nimbus是通过定时机制来定期轮询zookeeper中的目录进而感知到supervisor的加入和退出。

```
(defprotocol StormClusterState
  (assignments [this callback])
  (assignment-info [this storm-id callback])
  (active-storms [this])
  (storm-base [this storm-id callback])

  (get-worker-heartbeat [this storm-id node port])
  (executor-beats [this storm-id executor->node+port])
  (supervisors [this callback])
  (supervisor-info [this supervisor-id])  ;; returns nil if doesn't exist
  (setup-heartbeats! [this storm-id])
  (teardown-heartbeats! [this storm-id])
  (teardown-topology-errors! [this storm-id])
  (heartbeat-storms [this])
  (error-topologies [this])

  (worker-heartbeat! [this storm-id node port info])
  (remove-worker-heartbeat! [this storm-id node port])
  (supervisor-heartbeat! [this supervisor-id info])
  (activate-storm! [this storm-id storm-base])
  (update-storm! [this storm-id new-elems])
  (remove-storm-base! [this storm-id])
  (set-assignment! [this storm-id info])
  (remove-storm! [this storm-id])
  (report-error [this storm-id task-id error])
  (errors [this storm-id task-id])

  (disconnect [this])
  )
```

## 3 supervisor启动场景分析

supervisor与外部的主要连接只有一个，即与zookeeper的通讯。 通过zookeeper，supervisor可以感知到有哪些新的任务需要认领或是哪些任务已经被重新安排。

### 3.1 程序入口

程序入口很容易找到

```
(defn -main []
  (-launch (standalone-supervisor)))
```

### 3.2 zookeeper

在nimbus启动场景分析中讲到过，与zookeeper建立连接都是通过mk-storm-cluster-state函数来实现。在supervisor中需要注意的是，哪些回调函数被注册。

mk-synchronize-supervisor中封装了这个过程

```
assignments-snapshot (assignments-snapshot storm-cluster-state sync-callback)

(defn- assignments-snapshot [storm-cluster-state callback]
  (let [storm-ids (.assignments storm-cluster-state callback)]
     (->> (dofor [sid storm-ids] {sid (.assignment-info storm-cluster-state sid callback)})
          (apply merge)
          (filter-val not-nil?)
          )))
```

```
mk-synchronize-supervisor
          └── assignments-snapshot
                        ├── assignments
                        └── assignment-info
```

有了上述两个回调函数，supervisor就能知晓以下几件事情

- 有没有新的assignments或assignment被删除
- 某一已经存在的assignment是否需要重新分配

### 3.2.1 event-manager

event-manager事件处理的管理器，在一单独的线程中运行。
event-manager采用多生产者，单消费者模式。

## 3.3 sync-processes

supervisor会管理多个worker，对这些worker的启动与停止都是通过sync-processes来完成。

通过配置文件，supervisor知道自己应该管理多少个worker;通过zookeeper，知道有哪些task要在本机执行。

worker是通过执行函数launch-worker被启动起来的，那么supervisor是如何知道worker正常启动与否的呢。通过wait-for-workers-launch来判断。

wait-for-workers-launch的逻辑比较简单，就是利用zookeeper来检查worker是否已经发送心跳消息给zookeeper

# Part II
# Topology

# 4 Topology提交过程

storm cluster可以想像成为一个工厂，nimbus主要负责从外部接收订单和任务分配。除了从外部接单，nimbus还要将这些外部订单转换成为内部工作分配，这个时候nimbus充当了调度室的角色。

supervisor作为中层干部，职责就是生产车间的主任，他的日常工作就是时刻等待着调度到给他下达新的工作。作为车间主任，supervisor领到的活是不

用自己亲力亲为去作的，他手下有着一班的普通工人。supervisor对这些工人只会喊两句话，开工，收工。注意，讲收工的时候并不意味着worker手上的活已经干完了，只是进入休息状态而已。

topology的提交过程涉及到以下角色。

**storm client** 负责将用户创建的topology提交到nimbus

**nimbus** 通过thrift接口接收用户提交的topology

**supervisor** 根据zk接口上提示的消息下载最新的任务安排，并负责启动worker

**worker** worker内可以运行task,这些task要么属于bolt类型，要么属于spout类型

**executor** executor是一个个运行的线程，同一个executor内可以运行同一种类型的task,即一个线程中的task要么全部是bolt类型，要么全部是spout类型

## 4.1 StormBuilder

Topology的提交和执行过程很复杂，也是理解storm源码框架必须要解释的经典用例。原本理解的是supervisor认领一个assignment，启动单个worker，在worker内执行多个executor简单场景。

如果同一个topology，其中多个task落在不同的assignment中，这些assignment是如何被一一分配直到executor被执行。

### 4.1.1 setSpout

### 4.1.2 setBolt

### 4.1.3 createTopology

## 4.2 storm client

storm client需要执行下面这句指令将要提交的topology提交给storm cluster 假设jar文件名为storm-starter-0.0.1-snapshot-standalone.jar,启动程序为 storm.starter.ExclamationTopology,给这个topology起的名称为exclamationTopology

```
#./storm jar
  $HOME/working/storm-starter/target/storm-starter-0.0.1-SNAPSHOT-standalone.jar
  storm.starter.ExclamationTopology exclamationTopology
```

这么短短的一句话对于storm client来说，究竟意味着什么呢？ 源码面前是没有任何秘密可言的，那好打开storm client的源码文件

```
def jar(jarfile, klass, *args):
    """Syntax: [storm jar topology-jar-path class ...]

    Runs the main method of class with the specified arguments.
    The storm jars and configs in ~/.storm are put on the classpath.
    The process is configured so that StormSubmitter
    (http://nathanmarz.github.com/storm/doc/backtype/storm/StormSubmitter.html)
    will upload the jar at topology-jar-path when the topology is submitted.
    """
    exec_storm_class(
        klass,
        jvmtype="-client",
        extrajars=[jarfile, USER_CONF_DIR, STORM_DIR + "/bin"],
        args=args,
        jvmopts=["-Dstorm.jar=" + jarfile])
```

```
def exec_storm_class(klass, jvmtype="-server", jvmopts=[],
              extrajars=[], args=[], fork=False):
    global CONFFILE
    all_args = [
        "java", jvmtype, get_config_opts(),
        "-Dstorm.home=" + STORM_DIR,
        "-Djava.library.path=" + confvalue("java.library.path", extrajars),
        "-Dstorm.conf.file=" + CONFFILE,
        "-cp", get_classpath(extrajars),
    ] + jvmopts + [klass] + list(args)
    print "Running: " + " ".join(all_args)
    if fork:
        os.spawnvp(os.P_WAIT, "java", all_args)
    else:
        os.execvp("java", all_args) # replaces the current process and
        never returns
```

exec_storm_class说白了就是要运行传进来了的WordCountTopology类中main函数，再看看main函数的实现

```
public static void main(String[] args) throws Exception {
    TopologyBuilder builder = new TopologyBuilder();

    builder.setSpout("spout", new RandomSentenceSpout(), 5);
    builder.setBolt("split", new SplitSentence(), 8).shuffleGrouping("spout");
    builder.setBolt("count", new WordCount(), 12).fieldsGrouping("split", new Fields("word"));

    Config conf = new Config();
    conf.setDebug(true);

    if (args != null && args.length > 0) {
      conf.setNumWorkers(3);

      StormSubmitter.submitTopology(args[0], conf, builder.createTopology());
    }
}
```

对于storm client侧来说，最主要的函数StormSubmitter露出了真面目，submitTopology才是我们真正要研究的重点。

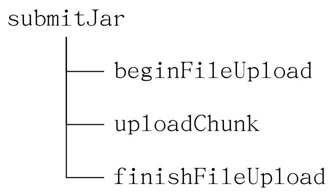
```
 public static void submitTopology(String name, Map stormConf,
 StormTopology topology, SubmitOptions opts)
throws AlreadyAliveException, InvalidTopologyException
```

```
{
        if(!Utils.isValidConf(stormConf)) {
            throw new IllegalArgumentException("Storm conf is not valid. Must be json-serializable");
        }
        stormConf = new HashMap(stormConf);
        stormConf.putAll(Utils.readCommandLineOpts());
        Map conf = Utils.readStormConfig();
        conf.putAll(stormConf);
        try {
            String serConf = JSONValue.toJSONString(stormConf);
            if(localNimbus!=null) {
                LOG.info("Submitting topology " + name + " in local mode");
                localNimbus.submitTopology(name, null, serConf, topology);
            } else {
                NimbusClient client = NimbusClient.getConfiguredClient(conf);
                if(topologyNameExists(conf, name)) {
                    throw new RuntimeException("Topology with name `"
                    + name
                    + "` already exists on cluster");
                }
                submitJar(conf);
                try {
                    LOG.info("Submitting topology " +  name
                    + " in distributed mode with conf " + serConf);
                    if(opts!=null) {
                        client.getClient().submitTopologyWithOpts(name, submittedJar, serConf, topology, opts);
                    } else {
                        // this is for backwards compatibility
                        client.getClient().submitTopology(name, submittedJar, serConf, topology);
                    }
                } catch(InvalidTopologyException e) {
                    LOG.warn("Topology submission exception", e);
                    throw e;
                } catch(AlreadyAliveException e) {
                    LOG.warn("Topology already alive exception", e);
                    throw e;
                } finally {
                    client.close();
                }
            }
            LOG.info("Finished submitting topology: " +  name);
        } catch(TException e) {
            throw new RuntimeException(e);
        }
    }
```

submitTopology函数其实主要就干两件事，一上传jar文件到storm cluster，另一件事通知storm cluster文件已经上传完毕，你可以执行某某某topology了。

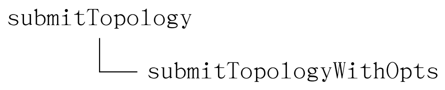
先看上传jar文件对应的函数submitJar,其调用关系如下图所示

Figure 3: call tree for submitJar



再看第二步中的调用关系

Figure 4: call tree for submitTopologyWithOpts



在上述两幅调用关系图中，处于子树位置的函数都曾在storm.thrift中声明，如果此刻已经忘记了的点话，可以翻看一下前面1.3节中有关storm.thrift的描述。 client侧的这些函数都是由thrift自动生成的。

## 4.3 nimbus

storm client侧通过thrift接口向nimbus发送了了jar并且通过预先定义好的submitTopologyWithOpts来处理上传的topology，那么nimbus是如何一步步的进行文件接收并将其任务细化最终下达给supervisor的呢。

### 4.3.1 submitTopologyWithOpts

一切还是要从thrift说起，supervisor.clj中的service-handler具体实现了thrift定义的Nimbus接口，代码这里就不罗列了，太占篇幅。主要看其是如何实现submitTopologyWithOpts

```
(^void submitTopologyWithOpts
        [this ^String storm-name ^String uploadedJarLocation ^String serializedConf ^StormTopology topology
         ^SubmitOptions submitOptions]
        (try
          (assert (not-nil? submitOptions))
          (validate-topology-name! storm-name)
          (check-storm-active! nimbus storm-name false)
          (.validate ^backtype.storm.nimbus.ITopologyValidator (:validator nimbus)
                     storm-name
                     (from-json serializedConf)
                     topology)
          (swap! (:submitted-count nimbus) inc)
          (let [storm-id (str storm-name "-" @(:submitted-count nimbus) "-" (current-time-secs))
                storm-conf (normalize-conf
                             conf
                             (-> serializedConf
                                 from-json
                                 (assoc STORM-ID storm-id)
```

```
                              (assoc TOPOLOGY-NAME storm-name))
                             topology)
                total-storm-conf (merge conf storm-conf)
                topology (normalize-topology total-storm-conf topology)
                topology (if (total-storm-conf TOPOLOGY-OPTIMIZE)
                           (optimize-topology topology)
                           topology)
                storm-cluster-state (:storm-cluster-state nimbus)]
            (system-topology! total-storm-conf topology) ;; this validates the structure of the topology
            (log-message "Received topology submission for " storm-name " with conf " storm-conf)
            ;; lock protects against multiple topologies being submitted at once and
            ;; cleanup thread killing topology in b/w assignment and starting the topology
            (locking (:submit-lock nimbus)
              (setup-storm-code conf storm-id uploadedJarLocation storm-conf topology)
              (.setup-heartbeats! storm-cluster-state storm-id)
              (let [thrift-status->kw-status {TopologyInitialStatus/INACTIVE :inactive
                                              TopologyInitialStatus/ACTIVE :active}]
                (start-storm nimbus storm-name storm-id (thrift-status->kw-status (.get_initial_status submitOpti
              (mk-assignments nimbus)))
          (catch Throwable e
            (log-warn-error e "Topology submission exception. (topology name='" storm-name "')")
            (throw e))))
```

storm cluster在zookeeper server上创建的目录结构。目录结构相关的源文件是config.clj.

白话一下上面这个函数的执行逻辑，对上传的topology作必要的检测，包括名字，文件内容及格式，好比你进一家公司上班之前做的体检。这些工作都完成之后进入关键区域，是进入关键区域所以上锁，呵呵。

normalize-topology会调用deepCopy,copy的源和目的分别是什么？

### 4.3.2 normalize-topology

normalize-topology的功能描述 all-components

```
(defn all-components [^StormTopology topology]
  (apply merge {}
         (for [f thrift/STORM-TOPOLOGY-FIELDS]
           (.getFieldValue topology f)
           )))
```

一旦列出所有的components,就可以读出这些component的配置信息。

### 4.3.3 mk-assignments

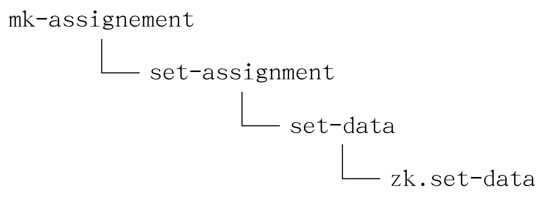
在这关键区域内执行的重点就是函数mk-assignments，mk-assignment有两个主要任务，第一是计算出有多少task,即有多少个spout,多少个bolt，第二就是在刚才的计算基础上通过调用zookeeper应用接口，写入assignment，以便supervisor感知到有新的任务需要认领。

先说第二点，因为逻辑简单。在mk-assignment中执行如下代码在zookeeper中设定相应的数据以便supervisor能够感知到有新的任务产生

```
  (doseq [[topology-id assignment] new-assignments
          :let [existing-assignment (get existing-assignments topology-id)
                topology-details (.getById topologies topology-id)]]
    (if (= existing-assignment assignment)
      (log-debug "Assignment for " topology-id " hasn't changed")
      (do
        (log-message "Setting new assignment for topology id " topology-id ": "
                (pr-str assignment))
        (.set-assignment! storm-cluster-state topology-id assignment)
        )))
```

调用关系如下图所示

Figure 5: call tree for mk-assignments



而第一点涉及到的计算相对繁杂，需要一一仔细道来。其实第一点中非常重要的课题就是如何进行任务的分发，即scheduling. 也许你已经注意到目录src/clj/backtype/storm/scheduler，或者注意到storm.yaml中与scheduler相关的配置项。那么这个scheduler到底是在什么时候起作用的呢。 mk-assignments会间接调用到这么一个名字看起来奇怪异常的函数。

compute-new-topology->executor->node+por

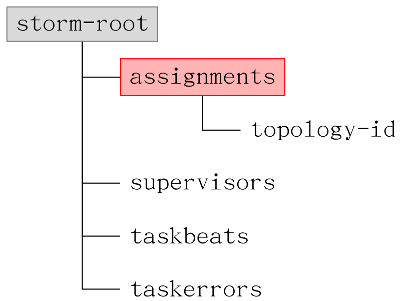
也就是在这么很奇怪的函数内，scheduler被调用

```
 _ (.schedule (:scheduler nimbus) topologies cluster)
new-scheduler-assignments (.getAssignments cluster)
;; add more information to convert SchedulerAssignment to Assignment
new-topology->executor->node+port (compute-topology->executor->node+port new-scheduler-assignments)]
```

schedule计算出来的assignments保存于Cluster.java中，这也是为什么new-scheduler-assignment要从其中读取数据的缘由所在。有了assignment，就可以计算出相应的node和port，其实就是这个任务应该交由哪个supervisor上的worker来执行。

storm在zookeeper server上创建的目录结构如下图所示

Figure 6: storm directory layout in zookeeper



有了这个目录结构，现在要解答的问题是在topology在提交的时候要写哪几个目录? assignments目录下会新创建一个新提交的topology的目录，在这个topology中需要写的数据，其数据结构是什么样子?

0.80版之前，有一个task-id的目录，在0.8.0版之后，这个子目录已经被删除，具体原因请看storm 0.8 release note

#### 4.3.4 Topology Monitoring

检测topology的运行状态，定时查看taskbeats和supervisors所在的目录。

### 4.4 supervisor

一旦有新的assignment被写入到zookeeper中，supervisor中的回调函数mk-synchronize-supervisor立马被唤醒执行

主要执行逻辑就是读入zookeeper server中新的assignments全集与已经运行与本机上的assignments作比较，区别出哪些是新增的。在sync-processes函数中将运行具体task的worker拉起。

重点分析mk-synchronize-supervisor函数

### 4.5 worker

worker是被supervisor通过函数launch-worker带起来的。并没有外部的指令显示的启动或停止worker,当然kill除外，:).

worker的主要任务有

- 发送心跳消息
- 接收外部tuple的消息
- 向外发送tuple消息

这些工作集中在mk-worker指定处理句柄。

## 4.6 executor

executor是通过worker执行mk-executor完成初始化过程。

```clojure
(defn mk-executor [worker executor-id]
 (let [executor-data (mk-executor-data worker executor-id)
   _ (log-message "Loading executor " (:component-id executor-data) ":" (pr-str executor-id))
   task-datas (->> executor-data
                   :task-ids
                   (map (fn [t] [t (task/mk-task executor-data t)]))
                   (into {})
                   (HashMap.))
   _ (log-message "Loaded executor tasks " (:component-id executor-data) ":" (pr-str executor-id))
   report-error-and-die (:report-error-and-die executor-data)
   component-id (:component-id executor-data)

   ;; starting the batch-transfer->worker ensures that anything publishing to that queue
   ;; doesn't block (because it's a single threaded queue and the caching/consumer started
   ;; trick isn't thread-safe)
   system-threads [(start-batch-transfer->worker-handler! worker executor-data)]
   handlers (with-error-reaction report-error-and-die
              (mk-threads executor-data task-datas))
   threads (concat handlers system-threads)]
    (setup-ticks! worker executor-data)

    (log-message "Finished loading executor " component-id ":" (pr-str executor-id))
    ;; TODO: add method here to get rendered stats... have worker call that when heartbeating
    (reify
      RunningExecutor
      (render-stats [this]
        (stats/render-stats! (:stats executor-data)))
      (get-executor-id [this]
        executor-id )
      Shutdownable
      (shutdown
        [this]
        (log-message "Shutting down executor " component-id ":" (pr-str executor-id))
        (disruptor/halt-with-interrupt! (:receive-queue executor-data))
        (disruptor/halt-with-interrupt! (:batch-transfer-queue executor-data))
        (doseq [t threads]
          (.interrupt t)
          (.join t))

        (doseq [user-context (map :user-context (vals task-datas))]
          (doseq [hook (.getHooks user-context)]
            (.cleanup hook)))
        (.disconnect (:storm-cluster-state executor-data))
        (when @(:open-or-prepare-was-called? executor-data)
          (doseq [obj (map :object (vals task-datas))]
            (close-component executor-data obj)))
        (log-message "Shut down executor " component-id ":" (pr-str executor-id)))
        )))
```

上述代码中mk-threads用来为spout或者bolt创建thread.

mk-threads使用到了clojure的函数重载机制，借用一下java或c++的术语吧。在clojure中使用defmulti来声明一个重名函数。

mk-threads函数有点长而且逻辑变得更为复杂，还是先从大体上有个概念为好，再去慢慢查看细节。

**async-loop** 线程运行的主函数，类似于pthread_create中的参数start_routine

**tuple-action-fn** spout和bolt都会收到tuple,处理tuple的逻辑不同但有一个同名的处理函数即是tuple-action-fn

**event-handler** 在这个创建的线程中又使用了disruptor模式，disruptor模式一个重要的概念就是要定义相应的event-handler。上面所讲的tuple-action-fn就是在event-handler中被处理。

调用逻辑如下图所示

Figure 7: call tree for tuple-action-fn

```
async-loop
      └── disruptor/consume-batch
                          └── tuple-action-fn
```

spout最主要的是去调用nextTuple，nextTuple是在async-loop中被调用;bolt最主要的是调用execute函数，而bolt中的execute是在event-handler的时候被唤起。nextTuple和execute会在消息发送一节中有更多的阐述。

有了这三个概念在心中，在看代码的时候，我们紧盯住这三点，其中的差异就会变得更清楚。

### 4.6.1 spout

先来看看如果是spout,mk-threads的处理步骤是啥样的,先说这个async-loops

```
[(async-loop
      (fn []
        ;; If topology was started in inactive state, don't call (.open spout) until it's activated first.
        (while (not @(:storm-active-atom executor-data))
          (Thread/sleep 100))

        (log-message "Opening spout " component-id ":" (keys task-datas))
        (doseq [[task-id task-data] task-datas
                :let [^ISpout spout-obj (:object task-data)
                      tasks-fn (:tasks-fn task-data)
                      send-spout-msg (fn [out-stream-id values message-id out-task-id]
                                       (.increment emitted-count)
   (let [out-tasks (if out-task-id
                     (tasks-fn out-task-id out-stream-id values)
                     (tasks-fn out-stream-id values))
         rooted? (and message-id has-ackers?)
         root-id (if rooted? (MessageId/generateId rand))
         out-ids (fast-list-for [t out-tasks] (if rooted? (MessageId/generateId rand)))]
     (fast-list-iter [out-task out-tasks id out-ids]
                     (let [tuple-id (if rooted?
                                      (MessageId/makeRootId root-id id)
                                      (MessageId/makeUnanchored))
                           out-tuple (TupleImpl. worker-context
                                                 values
                                                 task-id
                                                 out-stream-id
                                                 tuple-id)]
                       (transfer-fn out-task
                                    out-tuple
                                    overflow-buffer)
                       ))
     (if rooted?
       (do
         (.put pending root-id [task-id
                                message-id
                                {:stream out-stream-id :values values}
                                (if (sampler) (System/currentTimeMillis))])
         (task/send-unanchored task-data
                               ACKER-INIT-STREAM-ID
                               [root-id (bit-xor-vals out-ids) task-id]
                               overflow-buffer))
       (when message-id
         (ack-spout-msg executor-data task-data message-id
                        {:stream out-stream-id :values values}
                        (if (sampler) 0))))
     (or out-tasks [])
     ))]]
          (builtin-metrics/register-all (:builtin-metrics task-data) storm-conf (:user-context task-data))
          (builtin-metrics/register-queue-metrics {:sendqueue (:batch-transfer-queue executor-data)
                                                   :receive receive-queue}
                                                  storm-conf (:user-context task-data))

          (.open spout-obj
                 storm-conf
                 (:user-context task-data)
                 (SpoutOutputCollector.
                  (reify ISpoutOutputCollector
                    (^List emit [this ^String stream-id ^List tuple ^Object message-id]
                      (send-spout-msg stream-id tuple message-id nil)
                      )
                    (^void emitDirect [this ^int out-task-id ^String stream-id
                                       ^List tuple ^Object message-id]
                      (send-spout-msg stream-id tuple message-id out-task-id)
                      )
                    (reportError [this error]
                      (report-error error)
                      )))))
        (reset! open-or-prepare-was-called? true)
        (log-message "Opened spout " component-id ":" (keys task-datas))
        (setup-metrics! executor-data)

        (disruptor/consumer-started! (:receive-queue executor-data))
        (fn []
          ;; This design requires that spouts be non-blocking
          (disruptor/consume-batch receive-queue event-handler)

          ;; try to clear the overflow-buffer
          (try-cause
            (while (not (.isEmpty overflow-buffer))
              (let [[out-task out-tuple] (.peek overflow-buffer)]
                (transfer-fn out-task out-tuple false nil)
                (.removeFirst overflow-buffer)))
          (catch InsufficientCapacityException e
            ))

          (let [active? @(:storm-active-atom executor-data)
                curr-count (.get emitted-count)]
            (if (and (.isEmpty overflow-buffer)
                     (or (not max-spout-pending)
                         (< (.size pending) max-spout-pending)))
              (if active?
                (do
                  (when-not @last-active
                    (reset! last-active true)
                    (log-message "Activating spout " component-id ":" (keys task-datas))
                    (fast-list-iter [^ISpout spout spouts] (.activate spout)))

                  (fast-list-iter [^ISpout spout spouts] (.nextTuple spout)))
                (do
                  (when @last-active
                    (reset! last-active false)
                    (log-message "Deactivating spout " component-id ":" (keys task-datas))
                    (fast-list-iter [^ISpout spout spouts] (.deactivate spout)))
                  ;; TODO: log that it's getting throttled
                  (Time/sleep 100))))
            (if (and (= curr-count (.get emitted-count)) active?)
              (do (.increment empty-emit-streak)
                  (.emptyEmit spout-wait-strategy (.get empty-emit-streak)))
              (.set empty-emit-streak 0)
              ))
          0))
      :kill-fn (:report-error-and-die executor-data)
      :factory? true
      :thread-name component-id)]))
```

对于spout来说,如何处理收到的数据呢，这一切都要与disruptor/consume-batch关联起来。

注意上述代码中的这句

```
(disruptor/consume-batch receive-queue event-handler)
```

再看event-handler的定义

```
event-handler (mk-task-receiver executor-data tuple-action-fn)
```

Figure 8: tuple-action-fn for spout

```clojure
(fn [task-id ^TupleImpl tuple]
  [stream-id (.getSourceStreamId tuple)]
 ondp = stream-id
 Constants/SYSTEM_TICK_STREAM_ID (.rotate pending)
 Constants/METRICS_TICK_STREAM_ID (metrics-tick executor-data task-datas tuple)
 (let [id (.getValue tuple 0)
       [stored-task-id spout-id tuple-finished-info start-time-ms] (.remove pending id)]
   (when spout-id
     (when-not (= stored-task-id task-id)
       (throw-runtime "Fatal error, mismatched task ids: " task-id " " stored-task-id))
     (let [time-delta (if start-time-ms (time-delta-ms start-time-ms))]
       (condp = stream-id
         ACKER-ACK-STREAM-ID (ack-spout-msg executor-data (get task-datas task-id)
                                            spout-id tuple-finished-info time-delta)
         ACKER-FAIL-STREAM-ID (fail-spout-msg executor-data (get task-datas task-id)
                                              spout-id tuple-finished-info time-delta)
         )))
   ;; TODO: on failure, emit tuple to failure stream
   ))))
```

### 4.6.2 bolt

再来看bolt thread中async-loop的定义

```clojure
(async-loop
 (fn []
   ;; If topology was started in inactive state, don't call prepare bolt until it's activated first.
   (while (not @(:storm-active-atom executor-data))
     (Thread/sleep 100))

   (log-message "Preparing bolt " component-id ":" (keys task-datas))
   (doseq [[task-id task-data] task-datas
     :let [^IBolt bolt-obj (:object task-data)
           tasks-fn (:tasks-fn task-data)
           user-context (:user-context task-data)
           bolt-emit (fn [stream anchors values task]
    (let [out-tasks (if task
                      (tasks-fn task stream values)
                      (tasks-fn stream values))]
      (fast-list-iter [t out-tasks]
                      (let [anchors-to-ids (HashMap.)]
                        (fast-list-iter [^TupleImpl a anchors]
                                        (let [root-ids (-> a .getMessageId .getAnchorsToIds .keySet)]
                                          (when (pos? (count root-ids))
                                            (let [edge-id (MessageId/generateId rand)]
                                              (.updateAckVal a edge-id)
                                              (fast-list-iter [root-id root-ids]
                                                              (put-xor! anchors-to-ids root-id edge-id))
                                              ))))
                        (transfer-fn t
                                     (TupleImpl. worker-context
                                                 values
                                                 task-id
                                                 stream
                                                 (MessageId/makeId anchors-to-ids)))))
      (or out-tasks []))))]]
     (builtin-metrics/register-all (:builtin-metrics task-data) storm-conf user-context)
     (if (= component-id Constants/SYSTEM_COMPONENT_ID)
       (builtin-metrics/register-queue-metrics {:sendqueue (:batch-transfer-queue executor-data)
                                                :receive (:receive-queue executor-data)
                                                :transfer (:transfer-queue (:worker executor-data))}
                                               storm-conf user-context)
       (builtin-metrics/register-queue-metrics {:sendqueue (:batch-transfer-queue executor-data)
                                                :receive (:receive-queue executor-data)}
                                               storm-conf user-context)
       )

     (.prepare bolt-obj
 storm-conf
 user-context
 (OutputCollector.
  (reify IOutputCollector
    (emit [this stream anchors values]
      (bolt-emit stream anchors values nil))
    (emitDirect [this task stream anchors values]
      (bolt-emit stream anchors values task))
    (^void ack [this ^Tuple tuple]
      (let [^TupleImpl tuple tuple
            ack-val (.getAckVal tuple)]
        (fast-map-iter [[root id] (.. tuple getMessageId getAnchorsToIds)]
                       (task/send-unanchored task-data
                                             ACKER-ACK-STREAM-ID
                                             [root (bit-xor id ack-val)])
                       ))
      (let [delta (tuple-time-delta! tuple)]
        (task/apply-hooks user-context .boltAck (BoltAckInfo. tuple task-id delta))
        (when delta
          (builtin-metrics/bolt-acked-tuple! (:builtin-metrics task-data)
                                             executor-stats
                                             (.getSourceComponent tuple)
                                             (.getSourceStreamId tuple)
                                             delta)
          (stats/bolt-acked-tuple! executor-stats
                                   (.getSourceComponent tuple)
                                   (.getSourceStreamId tuple)
                                   delta))))
    (^void fail [this ^Tuple tuple]
      (fast-list-iter [root (.. tuple getMessageId getAnchors)]
                      (task/send-unanchored task-data
                                            ACKER-FAIL-STREAM-ID
                                            [root]))
      (let [delta (tuple-time-delta! tuple)]
        (task/apply-hooks user-context .boltFail (BoltFailInfo. tuple task-id delta))
        (when delta
          (builtin-metrics/bolt-failed-tuple! (:builtin-metrics task-data)
                                              executor-stats
                                              (.getSourceComponent tuple)
                                              (.getSourceStreamId tuple))
```

```
          (stats/bolt-failed-tuple! executor-stats
                                    (.getSourceComponent tuple)
                                    (.getSourceStreamId tuple)
                                    delta))))
    (reportError [this error]
      (report-error error)
      )))))
   (reset! open-or-prepare-was-called? true)
   (log-message "Prepared bolt " component-id ":" (keys task-datas))
   (setup-metrics! executor-data)

   (let [receive-queue (:receive-queue executor-data)
         event-handler (mk-task-receiver executor-data tuple-action-fn)]
     (disruptor/consumer-started! receive-queue)
     (fn []
       (disruptor/consume-batch-when-available receive-queue event-handler)
       0)))
 :kill-fn (:report-error-and-die executor-data)
 :factory? true
 :thread-name component-id)
```

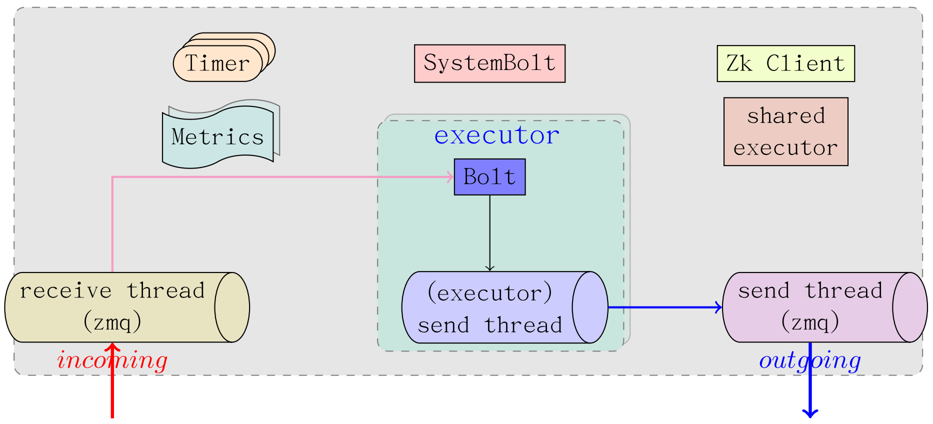
tuple-action-fn

```
(fn [task-id ^TupleImpl tuple]
  (let [stream-id (.getSourceStreamId tuple)]
    (condp = stream-id
      Constants/METRICS_TICK_STREAM_ID (metrics-tick executor-data task-datas tuple)
      (let [task-data (get task-datas task-id)
            ^IBolt bolt-obj (:object task-data)
            user-context (:user-context task-data)
            sampler? (sampler)
            execute-sampler? (execute-sampler)
            now (if (or sampler? execute-sampler?) (System/currentTimeMillis))]
        (when sampler?
          (.setProcessSampleStartTime tuple now))
        (when execute-sampler?
          (.setExecuteSampleStartTime tuple now))
        (.execute bolt-obj tuple)
        (let [delta (tuple-execute-time-delta! tuple)]
          (task/apply-hooks user-context .boltExecute (BoltExecuteInfo. tuple task-id delta))
          (when delta
            (builtin-metrics/bolt-execute-tuple! (:builtin-metrics task-data)
                                                 executor-stats
                                                 (.getSourceComponent tuple)
                                                 (.getSourceStreamId tuple)
                                                 delta)
            (stats/bolt-execute-tuple! executor-stats
                                       (.getSourceComponent tuple)
                                       (.getSourceStreamId tuple)
                                       delta)))))))
```

## 5 worker进程中线程的种类及用途

worker进程启动过程中最重要的两个函数是mk-worker和worker-data，代码就不一一列出了。worker顺利启动之后会拥有如下图所示的各类线程。 worker进程中所有的线程如下图所示，

Figure 9: tuple processing flow



### 5.1 接收和发送线程

worker在启动的时候会生成进程级别的消息接收和消息发送线程，它们视具体配置而定，可以是基于zmq，也可以基于netty,这个没有太多好说的。socket connection的建立过程可以在tuple消息传递一文中找到说明。

### 5.2 zk client

worker需要定期的向zk server发送心跳消息，与zk server之间的连接处理就落到zk client这个线程身上了。具体代码见函数do-heartbeat及do-executor-heartbeats。

### 5.3 定时器线程

worker进程需要定期的做些事情，比如发送心跳消息，刷新socket连接，这些定时器归为如下几类，每类定时器运行在各自的线程。

- :heartbeat-timer worker
- :refresh-connections-timer worker
- :refresh-active-timer worker

- :executor-heartbeat-timer worker
- :user-timer worker

上述定时器分类见于worker的shutdown函数,有时候在分析代码的时候，如果从入口看不清楚的话，不妨试试从退出的处理逻辑哪里找找答案。

## 5.4 SystemBolt

在topology提交的时候曾经见过函数system-topology!，这个函数会创建SystemBolt，每个worker内有且只有一个SystemBolt,可以见SystemBolt.java中注释的说明或参考github上storm对该改变的说明，https://github.com/nathanmarz/storm/pull/517。

SystemBolt主要进行进程相关的统计功能，比如内存使用情况，网络包的吞吐量，具体可见SystemBolt.java。SystemBolt是不接收tuple，只有出度，没有入度。

## 5.5 Metrics Bolt

MetricsBolt主要也是处理统计工作，与systembolt不同的是，metricsbolt主要处理executor级别的，如果用户在配置文件中定义了相关的MetricsConsumer类，那么这些类会在此被执行。

与之相关的配置内容

```
## Metrics Consumers
# topology.metrics.consumer.register:
#   - class: "backtype.storm.metrics.LoggingMetricsConsumer"
#     parallelism.hint: 1
#   - class: "org.mycompany.MyMetricsConsumer"
#     parallelism.hint: 1
#     argument:
#       - endpoint: "metrics-collector.mycompany.org"
```

## 5.6 Shared Executor

这个是在storm 0.8中引入的，其用途可在0.8的release notes中找到，创建共享线程池.

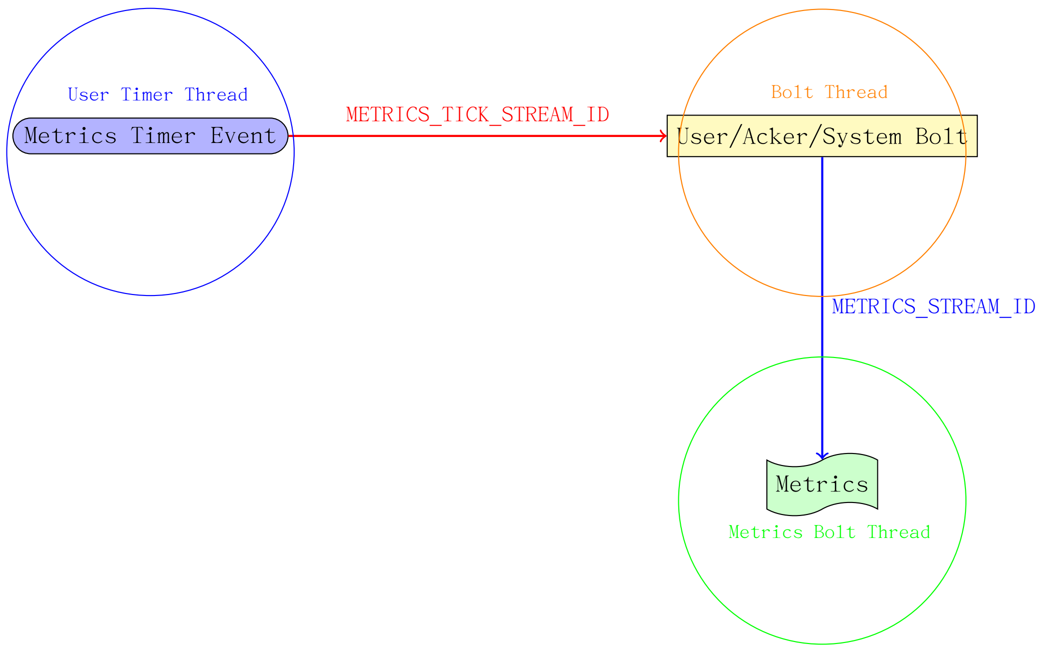
## 5.7 Metrics执行流程

metrics所做的计量工作是在什么时候被唤醒的呢，也就是说如何一步步的触发直到MetricsConsumeBolt的execute函数被调用。

下图勾勒出与metrics相关的线程间的消息传递过程。

简要说明如下

1. worker在启动的时候，会往:user-timer中注册metrics timer（见setup-metrics!函数）.



2. 一旦metrics timer超时，会发送一个stream-id为metrics-tick-stream-id的tuple到非metrics类型的bolt,如user/acker/system bolt.

3. 接收到tuple之后，会调用metrics-tick函数发送task-data给MetricsConsumerBolt，stream-id为metrics-stream-id

4. MetricsConsumerBolt接收到stream-id为metrics-stream-id的tuple后，会执行execute

注：在worker内部还有另一套计量api,定义于builtin-metrics.clj中，与MetricsConsumerBolt的区别在于，builtin-metrics是在处理外部进程发送过来的tuple时进行计量统计，而MetricsConsumerBolt是定时触发。

# Part III
# Message Passing

## 6 basic concepts

本章主要围绕Topology在执行过程中，消息的整个流转过程。首先要知道一些基本的概念。

1. Topology

2. Spout

3. Bolt

4. Tuple

5. Stream

从代码执行逻辑中来说组成topology的两个主要成员为spout和bolt.

从数据交互概念出发，流转于spout和bolt之间的数据称为tuple。从最上游的tuple到最下游的tuple,它们串联起来组成一个stream的虚拟概念。

数据交互的时会牵涉到分发策略，这就牵涉到另一个概念即grouping。

无论是Spout或Bolt的处理逻辑都需要在进程或线程内执行，那么它们与进程及线程间的映射关系又是如何呢。有关这个问题，Understanding the Parallelism of a Storm Topology 一文作了很好的总结，现重复一下其要点。

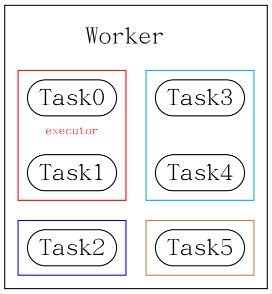
1. worker是进程，executor对应于线程，spout或bolt是一个个的task

2. 同一个worker只会执行同一个topology相关的task

3. 在同一个executor中可以执行多个同类型的task，即在同一个executor中，要么全部是bolt类的task，要么全部是spout类的task

4. 运行的时候，spout和bolt需要被包装成一个又一个task

worker，executor，task三者之间的关系可以用下图表示

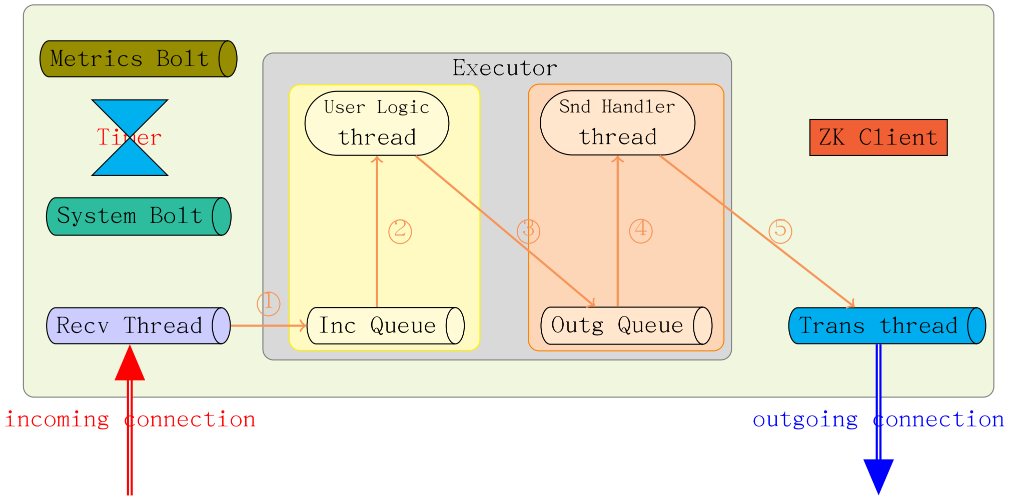


## 7 Message Passing

前面的章节讲述了各个运行着的实体是如何参与到topology的计算中的，本节主要试图阐述清楚tuple是如何产生并在组成topology的spout及bolt之间传送的。

假设同属于一个Topology的Spout与Bolt分别处于不同的JVM，即不同的worker中，不同的JVM可能处于同一台物理机器，也可能处于不同的物理机器中。为了让情景简单，认为JVM处于不同的物理机器中。

下图是外部消息接收处理的示意图。 注：圈起来的数字表示消息转换和处



理的序列。

### 步骤1

监听端口准备就绪，接收线程在收到外部的消息后，面临的问题就是如何确定由哪个task来处理该消息。接收到的tuple中含有task-id，根据task-id可以知

道运行该task的executor, executor中有receive-message-queue即(incoming queue)来存放外部的tuple. 定义的数据结构需要反映这个转换过程task-id->executor->receive-queue-map.

那么在worker-data中哪些数据项与这个过程相关呢

- :port
- :executor-receive-queue-map
- :short-executor-receive-queue-map
- :task->short-executor
- :transfer-local-fn

transfer-local-fn将数据从接收线程发送到spout或bolt所在的executor线程。

## 步骤2

接下来数据会被传递到executor, 于是又牵涉到executor的数据结构问题。executor-data由函数mk-executor-data创建, 其内容与worker-data比较起来相对较少。

executor收到tuple之后, 第一步需要进行反序列化, storm中使用kyro来进行序列化和反序列化, 这也是为什么在executor中有该数据项的原因。

executor中与步骤2相关的数据项

- :type executor-type
- :receive-queue
- :deserializer (executor-data中的数据项)

## 步骤3

步骤2处理结束, 会产生相应的tuple发送到外部。这个过程需要多解释一下, 首先tuple不是直接发送给worker的transfer-thread(负责向其它进程发送消息), 而是发送给send-handler线程, 每一个executor在创建的时候最起码会有两个线程被创建, 一个用于运行bolt或spout的处理逻辑, 另一个用以负责缓存bolt或spout产生的对外发送的tuple。

一旦snd-hander中的tuple数量达到阀值, 这些被缓存的tuple会一次性发送给worker级别的transfer-thread.

executor中与步骤3相关的数据项

- :transfer-fn (mk-executor-transfer-fn batch-transfer->worker)
- :batch-transfer-queue

在步骤3中生成outgoing的tuple, tuple生成的时候需要回答两个基本问题

- tuple中含有哪些字段 -- 该问题的解答由spout或bolt中的declareOut-Fields来解决
- 由哪个node+port来接收该tuple -- 由grouping来解决，这个时候就可以看出为什么需要task这一层的逻辑抽象了

## 步骤4

处理逻辑很简单，先将数据缓存，然后在达到阀值之后，一起传送给transfer-thread.

start-batch-transfer– >worker-handler

```
(defn start-batch-transfer->worker-handler! [worker executor-data]
(let [worker-transfer-fn (:transfer-fn worker)
      cached-emit (MutableObject. (ArrayList.))
      storm-conf (:storm-conf executor-data)
      serializer (KryoTupleSerializer. storm-conf (:worker-context executor-data))
      ]
  (disruptor/consume-loop*
    (:batch-transfer-queue executor-data)
    (disruptor/handler [o seq-id batch-end?]
      (let [^ArrayList alist (.getObject cached-emit)]
        (.add alist o)
        (when batch-end?
          (worker-transfer-fn serializer alist)
          (.setObject cached-emit (ArrayList.))
          )))
          :kill-fn (:report-error-and-die executor-data))))
```

worker-transfer-fn是worker中的transfer-fn，由mk-transfer-fn生成。

```
       (defn mk-transfer-fn [worker]
(let [local-tasks (-> worker :task-ids set)
      local-transfer (:transfer-local-fn worker)
      ^DisruptorQueue transfer-queue (:transfer-queue worker)]
  (fn [^KryoTupleSerializer serializer tuple-batch]
    (let [local (ArrayList.)
          remote (ArrayList.)]
      (fast-list-iter [[task tuple :as pair] tuple-batch]
        (if (local-tasks task)
          (.add local pair)
          (.add remote pair)
          ))
      (local-transfer local)
      ;; not using map because the lazy seq shows up in perf profiles
      (let [serialized-pairs (fast-list-for [[task ^TupleImpl tuple] remote] [task (.serialize serializer tuple)])]
        (disruptor/publish transfer-queue serialized-pairs)
        )))))
```

## 步骤5

处理函数mk-transfer-tuples-handler，主要进行序列化，将序列化后的数据发送给目的地址。

```
          (defn mk-transfer-tuples-handler [worker]
(let [^DisruptorQueue transfer-queue (:transfer-queue worker)
    drainer (ArrayList.)
    node+port->socket (:cached-node+port->socket worker)
    task->node+port (:cached-task->node+port worker)
    endpoint-socket-lock (:endpoint-socket-lock worker)
    ]
(disruptor/clojure-handler
  (fn [packets _ batch-end?]
    (.addAll drainer packets)
    (when batch-end?
      (read-locked endpoint-socket-lock
        (let [node+port->socket @node+port->socket
              task->node+port @task->node+port]
          ;; consider doing some automatic batching here (would need to not be serialized at this point to remove per-tuple ove
          ;; try using multipart messages ... first sort the tuples by the target node (without changing the local ordering)

          (fast-list-iter [[task ser-tuple] drainer]
            ;; TODO: consider write a batch of tuples here to every target worker
            ;; group by node+port, do multipart send
            (let [node-port (get task->node+port task)]
              (when node-port
                (.send ^IConnection (get node+port->socket node-port) task ser-tuple))
                ))))
                (.clear drainer))))))
```

tuple发送的时候需要用到connection,但目前只知道task-id, 所以在worker中需要保存task-id到node+port的映射, node+port与outgoing connections之间的映射。

worker中与步骤5相关的数据项:

1. :cached-node+port->socket
2. :cached-task->node+port
3. :component->stream->fields
4. :component->sorted-tasks
5. :endpoint-socket-lock
6. :transfer-queue (线程内部的消息队列)
7. :task->component

## 7.1 Tuple接收

Spout的输出消息到达Bolt, 作为Bolt的输入会经过这么几个阶段。

- spout的输出通过该spout所处worker的消息输出线程, 将tuple输入到Bolt所属的worker。它们之间的通路是socket连接, 用ZeroMQ实现
- bolt所处的worker有一个专门处理socket消息的receive thread 接收到spout发送来的tuple
- receive thread将接收到的消息传送给对应的bolt所在的executor。 在worker内部 (即同一process内部),消息传递使用的是Lmax Disruptor pattern.
- executor接收到tuple之后, 由event-handler进行处理

### 7.1.1 worker创建接收线程

```clojure
(defn launch-receive-thread [worker]
  (log-message "Launching receive-thread for " (:assignment-id worker) ":" (:port worker))
  (msg-loader/launch-receive-thread!
    (:mq-context worker)
    (:storm-id worker)
    (:port worker)
    (:transfer-local-fn worker)
    (-> worker :storm-conf (get TOPOLOGY-RECEIVER-BUFFER-SIZE))
    :kill-fn (fn [t] (halt-process! 11))))
```

```clojure
(defmethod mk-threads :bolt [executor-data task-datas]
  (let [execute-sampler (mk-stats-sampler (:storm-conf executor-data))
        executor-stats (:stats executor-data)
        {:keys [storm-conf component-id worker-context transfer-fn report-error sampler
                open-or-prepare-was-called?]} executor-data
        rand (Random. (Utils/secureRandomLong))
        tuple-action-fn (fn [task-id ^TupleImpl tuple]
   (let [stream-id (.getSourceStreamId tuple)]
     (condp = stream-id
       Constants/METRICS_TICK_STREAM_ID (metrics-tick executor-data task-datas tuple)
       (let [task-data (get task-datas task-id)
             ^IBolt bolt-obj (:object task-data)
             user-context (:user-context task-data)
             sampler? (sampler)
             execute-sampler? (execute-sampler)
             now (if (or sampler? execute-sampler?) (System/currentTimeMillis))]
         (when sampler?
           (.setProcessSampleStartTime tuple now))
         (when execute-sampler?
           (.setExecuteSampleStartTime tuple now))
         (.execute bolt-obj tuple)
         (let [delta (tuple-execute-time-delta! tuple)]
           (task/apply-hooks user-context .boltExecute (BoltExecuteInfo. tuple task-id delta))
           (when delta
             (builtin-metrics/bolt-execute-tuple! (:builtin-metrics task-data)
                                                   executor-stats
                                                   (.getSourceComponent tuple)
                                                   (.getSourceStreamId tuple)
                                                   delta)
             (stats/bolt-execute-tuple! executor-stats
                                        (.getSourceComponent tuple)
                                        (.getSourceStreamId tuple)
                                        delta)))))))]

    [(async-loop
      (fn []
        ;; If topology was started in inactive state, don't call prepare bolt until it's activated first.
        (while (not @(:storm-active-atom executor-data))
          (Thread/sleep 100))

        (log-message "Preparing bolt " component-id ":" (keys task-datas))
        (doseq [[task-id task-data] task-datas
                :let [^IBolt bolt-obj (:object task-data)
                      tasks-fn (:tasks-fn task-data)
```

```
                    user-context (:user-context task-data)
                    bolt-emit (fn [stream anchors values task]
(let [out-tasks (if task
                  (tasks-fn task stream values)
                  (tasks-fn stream values))]
  (fast-list-iter [t out-tasks]
                  (let [anchors-to-ids (HashMap.)]
                    (fast-list-iter [^TupleImpl a anchors]
                                    (let [root-ids (-> a .getMessageId .getAnchorsToIds .keySet)]
                                      (when (pos? (count root-ids))
                                        (let [edge-id (MessageId/generateId rand)]
                                          (.updateAckVal a edge-id)
                                          (fast-list-iter [root-id root-ids]
                                                          (put-xor! anchors-to-ids root-id edge-id))
                                          ))))
                    (transfer-fn t
                                 (TupleImpl. worker-context
                                             values
                                             task-id
                                             stream
                                             (MessageId/makeId anchors-to-ids)))))
  (or out-tasks [])))]]
        (builtin-metrics/register-all (:builtin-metrics task-data) storm-conf user-context)
        (if (= component-id Constants/SYSTEM_COMPONENT_ID)
          (builtin-metrics/register-queue-metrics {:sendqueue (:batch-transfer-queue executor-data)
                                                   :receive (:receive-queue executor-data)
                                                   :transfer (:transfer-queue (:worker executor-data))}
                                                  storm-conf user-context)
          (builtin-metrics/register-queue-metrics {:sendqueue (:batch-transfer-queue executor-data)
                                                   :receive (:receive-queue executor-data)}
                                                  storm-conf user-context)
          )

        (.prepare bolt-obj
                  storm-conf
                  user-context
                  (OutputCollector.
                   (reify IOutputCollector
                     (emit [this stream anchors values]
                       (bolt-emit stream anchors values nil))
                     (emitDirect [this task stream anchors values]
                       (bolt-emit stream anchors values task))
                     (^void ack [this ^Tuple tuple]
                       (let [^TupleImpl tuple tuple
                             ack-val (.getAckVal tuple)]
                         (fast-map-iter [[root id] (.. tuple getMessageId getAnchorsToIds)]
                                        (task/send-unanchored task-data
                                                              ACKER-ACK-STREAM-ID
                                                              [root (bit-xor id ack-val)])
                                        ))
                       (let [delta (tuple-time-delta! tuple)]
                         (task/apply-hooks user-context .boltAck (BoltAckInfo. tuple task-id delta))
                         (when delta
                           (builtin-metrics/bolt-acked-tuple! (:builtin-metrics task-data)
                                                              executor-stats
                                                              (.getSourceComponent tuple)
                                                              (.getSourceStreamId tuple)
```

```
                                                         delta)
                             (stats/bolt-acked-tuple! executor-stats
                                                       (.getSourceComponent tuple)
                                                       (.getSourceStreamId tuple)
                                                       delta))))
                       (^void fail [this ^Tuple tuple]
                         (fast-list-iter [root (.. tuple getMessageId getAnchors)]
                                         (task/send-unanchored task-data
                                                               ACKER-FAIL-STREAM-ID
                                                               [root]))
                         (let [delta (tuple-time-delta! tuple)]
                           (task/apply-hooks user-context .boltFail (BoltFailInfo. tuple task-id delta))
                           (when delta
                             (builtin-metrics/bolt-failed-tuple! (:builtin-metrics task-data)
                                                                  executor-stats
                                                                  (.getSourceComponent tuple)
                                                                  (.getSourceStreamId tuple))
                             (stats/bolt-failed-tuple! executor-stats
                                                        (.getSourceComponent tuple)
                                                        (.getSourceStreamId tuple)
                                                        delta))))
                       (reportError [this error]
                         (report-error error)
                         )))))
        (reset! open-or-prepare-was-called? true)
        (log-message "Prepared bolt " component-id ":" (keys task-datas))
        (setup-metrics! executor-data)

        (let [receive-queue (:receive-queue executor-data)
              event-handler (mk-task-receiver executor-data tuple-action-fn)]
          (disruptor/consumer-started! receive-queue)
          (fn []
            (disruptor/consume-batch-when-available receive-queue event-handler)
            0)))
      :kill-fn (:report-error-and-die executor-data)
      :factory? true
      :thread-name component-id)]))
```

### 7.1.2 worker从socket接收消息

```
vthread (async-loop
  (fn []
    (let [socket (.bind ^IContext context storm-id port)]
      (fn []
        (let [batched (ArrayList.)
              init (.recv ^IConnection socket 0)]
          (loop [packet init]
            (let [task (if packet (.task ^TaskMessage packet))
                  message (if packet (.message ^TaskMessage packet))]
              (if (= task -1)
                (do (log-message "Receiving-thread:["
                                 storm-id ", " port "]
                                 received shutdown notice")
                  (.close socket)
                  nil )
                (do
                  (when packet (.add batched [task message]))
                  (if (and packet (< (.size batched) max-buffer-size))
                    (recur (.recv ^IConnection socket 1))
                    (do (transfer-local-fn batched)
                      0 ))))))))))
```

transfer-local-fn将接收的TaskMessage传递给相应的executor

### 7.1.3 disruptor在线程之间进行消息传递

mk-transfer-local-fn表示将外部世界的消息传递给本进程内的线程。而 mk-transfer-fn则刚好在方向上反过来。

```
(defn mk-transfer-local-fn [worker]
  (let [short-executor-receive-queue-map (:short-executor-receive-queue-map worker)
        task->short-executor (:task->short-executor worker)
        task-getter (comp #(get task->short-executor %) fast-first)]
    (fn [tuple-batch]
      (let [grouped (fast-group-by task-getter tuple-batch)]
        (fast-map-iter [[short-executor pairs] grouped]
          (let [q (short-executor-receive-queue-map short-executor)]
            (if q
              (disruptor/publish q pairs)
              (log-warn "Received invalid messages for unknown tasks. Dropping... ")
              )))))))
```

### 7.1.4 消息被executor处理

进程接收到的消息最终是被每个executor即相应的线程所消费。消息在线程之间的传递使用的是lmax提供的disrutptor库。

有关disruptor的详细介绍可以参考lmax distruptor

```
(defn mk-task-receiver [executor-data tuple-action-fn]
  (let [^KryoTupleDeserializer deserializer (:deserializer executor-data)
        task-ids (:task-ids executor-data)
        debug? (= true (-> executor-data :storm-conf (get TOPOLOGY-DEBUG)))
        ]
    (disruptor/clojure-handler
      (fn [tuple-batch sequence-id end-of-batch?]
        (fast-list-iter [[task-id msg] tuple-batch]
          (let [^TupleImpl tuple (if (instance? Tuple msg) msg (.deserialize deserializer msg))]
            (when debug? (log-message "Processing received message " tuple))
            (if task-id
              (tuple-action-fn task-id tuple)
              ;; null task ids are broadcast tuples
              (fast-list-iter [task-id task-ids]
                (tuple-action-fn task-id tuple)
                ))
            ))))))
```

加亮行中tuple-action-fn定义于mk-threads(源文件executor.clj)中。因为当前以Bolt为例，所以会调用的tuple-action-fn会使用如下的定义

```
defmethod mk-threads :bolt [executor-data task-datas]
```

那么mk-task-receiver是如何与disruptor关联起来的呢，可以见定义于mk-threads中的下述代码

```
(let [receive-queue (:receive-queue executor-data)
              event-handler (mk-task-receiver executor-data tuple-action-fn)]
          (disruptor/consumer-started! receive-queue)
          (fn []
            (disruptor/consume-batch-when-available receive-queue event-handler)
            0)))
```

到了这里，消息的接收处理路径打通。

## 7.2 Tuple发送

### 7.2.1 IConnection

```
bolt-emit (fn [stream anchors values task]
   (let [out-tasks (if task
                    (tasks-fn task stream values)
                    (tasks-fn stream values))]
     (fast-list-iter [t out-tasks]
     (let [anchors-to-ids (HashMap.)]
       (fast-list-iter [^TupleImpl a anchors]
                      (let [root-ids (-> a .getMessageId .getAnchorsToIds .keySet)]
                        (when (pos? (count root-ids))
                          (let [edge-id (MessageId/generateId rand)]
                            (.updateAckVal a edge-id)
                            (fast-list-iter [root-id root-ids]
                                            (put-xor! anchors-to-ids root-id edge-id))
                            ))))
       (transfer-fn t
                    (TupleImpl. worker-context
                                values
                                task-id
                                stream
                                (MessageId/makeId anchors-to-ids)))))
                    (or out-tasks [])))
```

到了这，事情还没有结，我们知道消息最终是通过调用zeromq接口朝外发送的，所以还要继续打通到调用zeromq的通路才行。在bolt-emit代码中，看到消息通过transfer-fn进行发送，那么transfer-fn又是在哪定义的呢?

```
:transfer-fn (mk-executor-transfer-fn batch-transfer->worker)
```

继续看mk-executor-transfer-fn函数实现

```
(defn mk-executor-transfer-fn [batch-transfer->worker]
  (fn this
    ([task tuple block? ^List overflow-buffer]
      (if (and overflow-buffer (not (.isEmpty overflow-buffer)))
        (.add overflow-buffer [task tuple])
        (try-cause
          (disruptor/publish batch-transfer->worker [task tuple] block?)
        (catch InsufficientCapacityException e
          (if overflow-buffer
            (.add overflow-buffer [task tuple])
            (throw e))
          ))))
    ([task tuple overflow-buffer]
      (this task tuple (nil? overflow-buffer) overflow-buffer))
    ([task tuple]
      (this task tuple nil)
      )))
```

transfer-fn将bolt或是spout所在运行线程产生的数据通过disruptor机制发送给进程中对外发送数据的线程。向进程外发送数据的线程由mk-transfer-fn函数生成，来看一下其定义。

Figure 10: woker.clj/mk-transfer-fn

```
(defn mk-transfer-fn [worker]
  (let [local-tasks (-> worker :task-ids set)
        local-transfer (:transfer-local-fn worker)
        ^DisruptorQueue transfer-queue (:transfer-queue worker)]
    (fn [^KryoTupleSerializer serializer tuple-batch]
      (let [local (ArrayList.)
            remote (ArrayList.)]
        (fast-list-iter [[task tuple :as pair] tuple-batch]
          (if (local-tasks task)
            (.add local pair)
            (.add remote pair)
            ))
        (local-transfer local)
        ;; not using map because the lazy seq shows up in perf profiles
        (let [serialized-pairs (fast-list-for [[task ^TupleImpl tuple] remote] [task (.serialize serializer tuple
          (disruptor/publish transfer-queue serialized-pairs)
          )))))
```

在worker中向专门负责向外发送数据的线程称为transfer-thread，它的运行逻辑见下述代码

```
transfer-thread (disruptor/consume-loop* (:transfer-queue worker) transfer-tuples)
```

上述代码的含义表示如果transfer-queue中有数据的话，那么使用transfer-tuples函数来进行处理。好，接下来就是看transfer-tuples的定义了。

```
(defn mk-transfer-tuples-handler [worker]
  (let [^DisruptorQueue transfer-queue (:transfer-queue worker)
        drainer (ArrayList.)
        node+port->socket (:cached-node+port->socket worker)
        task->node+port (:cached-task->node+port worker)
        endpoint-socket-lock (:endpoint-socket-lock worker)
        ]
    (disruptor/clojure-handler
      (fn [packets _ batch-end?]
        (.addAll drainer packets)
        (when batch-end?
          (read-locked endpoint-socket-lock
            (let [node+port->socket @node+port->socket
                  task->node+port @task->node+port]
              ;; consider doing some automatic batching here (would need to not be serialized at this point to rem
              ;; try using multipart messages ... first sort the tuples by the target node (without changing the l

              (fast-list-iter [[task ser-tuple] drainer]
                ;; TODO: consider write a batch of tuples here to every target worker
                ;; group by node+port, do multipart send
                (let [node-port (get task->node+port task)]
                  (when node-port
                    (.send ^IConnection (get node+port->socket node-port) task ser-tuple))
                    ))))
          (.clear drainer))))))
```

当追述到上面的这行代码的时候，应该可以确认向外发送数据的通路已经打通

```
(.send ^IConnection (get node+port->socket node-port) task ser-tuple))
```

IConnection又是在什么时候建立的呢？这个不复杂就是有点绕而已。在storm.yaml中有这么一个配置项

```
storm.messaging.transport: "backtype.storm.messaging.zmq"
```

这个配置项与worker中的mqcontext相对应，所以在worker中以mqcontext为线索，就能够一步步找到IConnection的实现。connections在函数mk-refresh-connections中建立

```
refresh-connections (mk-refresh-connections worker)
```

在mk-refresh-connections中能够找到这么一段代码

```
(swap! (:cached-node+port->socket worker)
 #(HashMap. (merge (into {} %1) %2))
 (into {}
   (dofor [endpoint-str new-connections
           :let [[node port] (string->endpoint endpoint-str)]]
     [endpoint-str
      (.connect
       ^IContext (:mq-context worker)
       storm-id
       ((:node->host assignment) node)
       port)
      ]
     )))
```

可以看到此外与刚才所说的mq-context挂起来了，默认是使用zmq来作为transport的。再打开zmq.clj来看看connection函数的实现

Figure 11: code snippet in zmq.clj

```
(^IConnection connect [this ^String storm-id ^String host ^int port]
    (require 'backtype.storm.messaging.zmq)
    (-> context
      (mq/socket mq/push)
      (mq/set-hwm hwm)
      (mq/set-linger linger-ms)
      (mq/connect (get-connect-zmq-url local? host port))
     mk-connection))
```

代码走到这，终于找到connection建立的源头了。

### 7.2.2 grouping

从一个bolt中产生的tuple可以有多个bolt接收，到底发送给哪一个bolt呢?这牵扯到分发策略问题，其实在twitter storm中有两个层面的分发策略问题，一个是对于task level的，在讲topology submit的时候已经涉及到。另一个就是现在要讨论的针对tuple level的分发。

再次将视线拉回到bolt-emit中，这次将目光集中在变量t的前前后后。

```
(let [out-tasks (if task
                  (tasks-fn task stream values)
                 (tasks-fn stream values))]
(fast-list-iter [t out-tasks]
                (let [anchors-to-ids (HashMap.)]
                  (fast-list-iter [^TupleImpl a anchors]
                                  (let [root-ids (-> a .getMessageId .getAnchorsToIds .keySet)]
                                    (when (pos? (count root-ids))
                                      (let [edge-id (MessageId/generateId rand)]
                                        (.updateAckVal a edge-id)
                                        (fast-list-iter [root-id root-ids]
                                                        (put-xor! anchors-to-ids root-id edge-id))
                                        ))))
```

```
                      (transfer-fn t
                                   (TupleImpl. worker-context
                                               values
                                               task-id
                                               stream
                                               (MessageId/makeId
                                               anchors-to-ids)))))
```

上述代码显示t从out-tasks来，而out-tasks是tasks-fn的返回值

```
tasks-fn (:tasks-fn task-data)
```

一谈tasks-fn，原来从未涉及的文件task.clj这次被挂上了，task-data与由task/mk-task创建。将中间环节跳过，调用关系如下所列。

- mk-task
- mk-task-data
- mk-tasks-fn

tasks-fn中会使用到grouping,处理代码如下

```
fn ([^Integer out-task-id ^String stream ^List values]
          (when debug?
            (log-message "Emitting direct: " out-task-id "; " component-id " " stream " " values))
          (let [target-component (.getComponentId worker-context out-task-id)
                component->grouping (get stream->component->grouper stream)
                grouping (get component->grouping target-component)
                out-task-id (if grouping out-task-id)]
            (when (and (not-nil? grouping) (not= :direct grouping))
              (throw (IllegalArgumentException. "Cannot emitDirect to a task expecting a regular grouping")))
            (apply-hooks user-context .emit (EmitInfo. values stream task-id [out-task-id]))
            (when (emit-sampler)
              (builtin-metrics/emitted-tuple! (:builtin-metrics task-data) executor-stats stream)
              (stats/emitted-tuple! executor-stats stream)
              (if out-task-id
                (stats/transferred-tuples! executor-stats stream 1)
                (builtin-metrics/transferred-tuple! (:builtin-metrics task-data) executor-stats stream 1)))
            (if out-task-id [out-task-id])
            ))
```

而每个topology中的grouping策略又是如何被executor知道的呢，这从另一端executor-data说起。
在mk-executor-data中有下面一行代码

```
:stream->component->grouper (outbound-components worker-context
  component-id)
```

outbound-components的定义如下

```
(defn outbound-components
  "Returns map of stream id to component id to grouper"
  [^WorkerTopologyContext worker-context component-id]
  (->> (.getTargets worker-context component-id)
       clojurify-structure
       (map (fn [[stream-id component->grouping]]
        [stream-id
         (outbound-groupings
          worker-context
          component-id
          stream-id
          (.getComponentOutputFields worker-context component-id stream-id)
          component->grouping)]))
       (into {})
       (HashMap.)))
```

Part IV

# Reliability Mechanism

## 8 Reliability Mechanism

### 8.1 Acking Framework

怎么知道一个从spout出来的tuple已经被topology中的所有结点处理过了呢，也就是说如何来保证处理结果是可信的呢。

一切需要从topology被提交到storm cluster的那一刻说起，在storm cluster上真正运行起来的topology与用户提交的topology还是有一些区别的.第一个是有隐式的acker bolt被创建.

这个转换后的topology由system-topology!创建.

acker bolt的目的是让spout知道从自己这发送出去的每一个tuple有没有被topology中的所有结点处理过，如果全部处理过，则向spout发送ack，如果失败则向spout发送fail.

acker bolt又是如何实现这点的呢，其利用的数学原理非常简单

$A \oplus A = 0$

任何数与自己相异或，其结果为零。

结合代码再来看一下其具体实现

```
class SplitSentence implements IRichBolt {
  public void execute(Tuple tuple) {
    String sentence = tuple.getstring(0);
    for (String word: sentence.split(" ")) {
      collector.emit(tuple, new Values(word));
    }
    collector.ack(tuple);
  }
}
```

上面代码中的最后一句对于处理的可靠性至为关键。

```
collector.ack(tuple)
```

IOutputCollector的接口定义

```
public interface IOutputCollector extends IErrorReporter {
  List<Integer> emit(String streamId,
                        Collection<Tuple> anchors, List<Object> tuple);
  void emitDirect(int taskId, String streamId, Collection<Tuple> anchors,
                              List<Object> tuple);
  void ack(Tuple input);
  void fail(Tuple input);
}
```

在executor.clj中利用reify对IOutputCollector进行实现。每向下发送一个tuple都会给其一个独立的编号，然后与上游的源tuple进行异或。ack会将在bolt产生的最后ack val发送给acker bolt。

acker bolt接收到最新的ack val，与该ack val对应的message的最新acker val进行异或，如果异或结果为真，表示tuple处理完成，否则表示处理失败。

```
(^void ack [this ^Tuple tuple]
  (let [^TupleImpl tuple tuple
        ack-val (.getAckVal tuple)]
    (fast-map-iter [[root id] (.. tuple getMessageId getAnchorsToIds)]
                   (task/send-unanchored task-data
                                         ACKER-ACK-STREAM-ID
                                         [root (bit-xor id ack-val)])
                   ))
  (let [delta (tuple-time-delta! tuple)]
    (task/apply-hooks user-context .boltAck (BoltAckInfo. tuple task-id delta))
    (when delta
      (builtin-metrics/bolt-acked-tuple! (:builtin-metrics task-data)
                                         executor-stats
                                         (.getSourceComponent tuple)
                                         (.getSourceStreamId tuple)
                                         delta)
      (stats/bolt-acked-tuple! executor-stats
                               (.getSourceComponent tuple)
                               (.getSourceStreamId tuple)
                               delta))))
```

acker bolt在什么时候会将更新的结果发送给Spout呢，如果update-ack的结果为0,表示已经处理完毕。如果不为0,则继续等待.

```
(when (and curr (:spout-task curr))
 (cond (= 0 (:val curr))
   (do
     (.remove pending id)
```

```
    (acker-emit-direct output-collector
                       (:spout-task curr)
                       ACKER-ACK-STREAM-ID
                       [id]
                       ))
  (:failed curr)
  (do
    (.remove pending id)
    (acker-emit-direct output-collector
                       (:spout-task curr)
                       ACKER-FAIL-STREAM-ID
                       [id]
                       ))
  ))
```

ISpout的定义

```
public interface ISpout extends Serializable {
  void open(Map conf, TopologyContext context, SpoutOutputCollector collector);
  void close();
  void activate();
  void deactivate();
  void nextTuple();
  void ack(Object msgId);
  void fail(Object msgId);
```

Spout最终收到通知了解到发送出去的tuple已经处理完毕或失败了，接下来该怎么做取决于用户自己。像在kestrel topology中的处理逻辑是，如果tuple处理成功，将tuple从缓存中移除；如果处理失败，则将处理失败的tuple再次emit出去，进行处理。

# 9 Trident Topology

TridentTopology是storm提供的高级调用接口，屏蔽了很多spout和bolt的细节，开发人员根据其提供的api就能用非常简短的程序写出许多很有实用价值的应用。

构成TridentTopology的几个重要概念

- stream
- batch
- TridentState

1. Operations that apply locally to each partition and cause no network transfer
2. Repartitioning operations that repartition a stream but otherwise don't change the contents (involves network transfer)
3. Aggregation operations that do network transfer as part of the operation
4. Operations on grouped streams
5. Merges and join

TridentTopology通过transactional spout与transactional state相结合，能够做到tuple“只被处理一次，不多也不少”。也就是做到事务性处理exactly-once,要么成功，要么失败。

而一般的storm topology是无法保证eactly-once的处理的，它们要么是at-least-once(至少被处理一次，有可能被处理多次)；要么是at-most-once（最多被处理一次，这样就存在遗漏的可能）.

TridentTopology在设计中借鉴和保留了目前已经过期的transactional topology的设计思想。

## 9.1 应用举例

百闻不如一见，举一个例子吧。

```
TridentTopology topology = new TridentTopology();
TridentState wordCounts =
     topology.newStream("spout1", spout)
       .each(new Fields("sentence"), new Split(), new Fields("word"))
       .groupBy(new Fields("word"))
       .persistentAggregate(new MemoryMapState.Factory(), new Count(),
           new Fields("count"))
       .parallelismHint(6);
```

Listing 1: TridentTopoloy WordCount

上述代码的含义解释如下

- 从spout中读取数据
- 将句子分隔成各个独立的句子
- 将单词分组
- 将统计单词个数，将其存入内存MemoryMap
- 并行数为6

上述的WordCount如果用Spark来写的话，代码如下

```scala
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
                  .map(word => (word, 1))
                  .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

## 9.2 TridentTopology创建过程详解

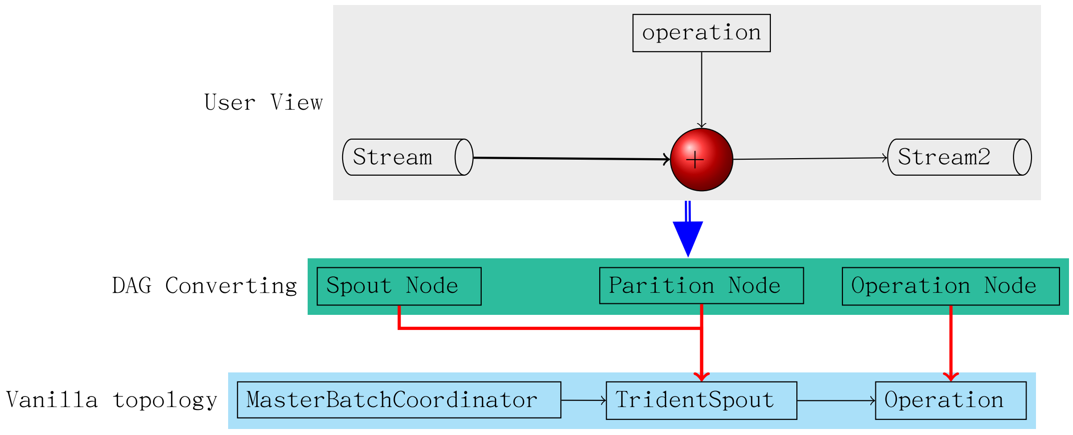
### 9.2.1 引言

TridentTopology是storm提供的高层使用接口，常见的一些SQL中的操作在tridenttopology提供的api中都有类似的影射。

从TridentTopology到vanilla topology(普通的topology)由三个层次组成：

1. 面向最终用户的概念stream，operation
2. 利用planner将tridenttopology转换成vanilla topology
3. 执行vanilla topology

从TridentTopology到基本的Topology有三层，下图是一个全局视图。 以



wordcount为例，使用TridentTopology上层接口来实现的源码如下所示

```
  TridentTopology topology = new TridentTopology();
    topology.newStream("spout1", spout).parallelismHint(16).each(new
        Fields("sentence"),
        new Split(), new Fields("word")).groupBy(new Fields("word")).
            persistentAggregate(new MemoryMapState.Factory(),
        new Count(), new Fields("count")).parallelismHint(16);

        return topology.build();
```

Listing 2: WordCount

上述代码的newStream一行，分两大部分，一是使用newStream来创建一个stream对象，然后针对该Stream进行各种操作，each/shuffle/persistentAggregate等就是各种operation.

用户在使用TridentTopology的时候，只需要熟悉Stream和TridentTopology中的API函数即可。

从用户层面来看TridentTopology，有两个重要的概念一是Stream,另一个是作用于Stream上的各种Operation。在实现层面来看，无论是stream，还是后续的operation都会转变成为各个Node，这些Node之间的关系通过重要的数据结构图来维护。具体到TridentTopology，实现图的各种操作的组件是jgrapht。

说到图，两个基本的概念会闪现出来，一是结点，二是描述结点之间关系的边。要想很好的理解TridentTopology就需要紧盯图中结点和边的变化。

TridentTopology在转换成为普通的StormTopology时，需要将原始的图分成各个group，每个group将运行于一个独立的bolt中。TridentTopology又是如何知道哪些node应该在同一个group,哪些应该处在另一个group中的呢；如何来确定每个group的并发度(parallismHint)的呢。这些问题的解决都与jgrapht分不开。

在TridentTopology中向图中添加结点的api有三种：

1. addNode

2. addSourcedNode

3. addSourcedStateNode

其中addNode在创建stream是使用，addSourcedStateNode在partitionPersist时使用到，其它的operation使用到的是addSourcedNode.

addNode与其它两个方法的一个重要区别还在于，addNode是不需要添加边(Edge)，而其它两个API需要往图中添加edge，以确定该node的源是哪个。

### 9.2.2 TridentTopoloy

```
public TridentTopology() {
        _graph = new DefaultDirectedGraph(new ErrorEdgeFactory());
        _gen = new UniqueIdGen();
    }
```

Listing 3: TridentTopology

在TridentTopology的构造函数中，创建了DAG(有向无环图)。利用这个_graph来作为容器以存储后续过程中创建的各个node及它们之间的关系。

### 9.2.3 newStream

newStream会为DAG(有向无环图)中创建源结点，其调用关系如下所示。

```
newStream
    addNode
        registerNode
```

```
protected void registerNode(Node n) {
        _graph.addVertex(n);
        if(n.stateInfo!=null) {
            String id = n.stateInfo.id;
            if(!_colocate.containsKey(id)) {
                _colocate.put(id, new ArrayList());
            }
            _colocate.get(id).add(n);
        }
    }
```
Listing 4: registerNode

### 9.2.4 each

作用于stream上的Operation有很多，以each为例来看新的operation是如何转换成为node添加到_graph中的。

```
     //Stream.java
public Stream each(Fields inputFields, Function function, Fields
    functionFields) {
  projectionValidation(inputFields);
  return _topology.addSourcedNode(this,
          new ProcessorNode(_topology.getUniqueStreamId(),
              _name,
              TridentUtils.fieldsConcat(getOutputFields(),
                  functionFields),
              functionFields,
              new EachProcessor(inputFields, function)));
            }
```
Listing 5: each

调用关系描述如下

**Stream::each**
**TridentTopology::addSourcedNode**
**TridentTopology::registerSourcedNode**

registerSourcedNode的实现如下

```
  protected void registerSourcedNode(List<Stream> sources, Node newNode
      ) {
        registerNode(newNode);
        int streamIndex = 0;
        for(Stream s: sources) {
            _graph.addEdge(s._node, newNode, new IndexedEdge(s._node,
                newNode, streamIndex));
            streamIndex++;
        }
      }
```
Listing 6: registerSourceNode

注意此处添加edge是，是有索引的，这样可以区别处理的先后顺序。

在Stream中含有成员变量_node，表示stream最近停泊的node,有了该变量添加edge才成为了可能。

### 9.2.5 partitionPersist

```
public TridentState partitionPersist(StateSpec stateSpec, Fields
    inputFields, StateUpdater updater, Fields functionFields) {
  projectionValidation(inputFields);
  String id = _topology.getUniqueStateId();
  ProcessorNode n = new ProcessorNode(_topology.getUniqueStreamId(),
              _name,
              functionFields,
              functionFields,
              new PartitionPersistProcessor(id, inputFields, updater))
                  ;
  n.committer = true;
  n.stateInfo = new NodeStateInfo(id, stateSpec);
  return _topology.addSourcedStateNode(this, n);
}
```

Listing 7: partitionPersist

调用关系

**Stream::partitionPersist**
**TridentTopology::addSourcedStateNode**
**TridentTopology::registerSourcedNode**

与addNode及addSourcedNode不同的是，addSourcedStateNode返回的是TridentState而非Stream。

既然谈到了TridentState就不得不谈到其另一面Stream::stateQuery，

```
public Stream stateQuery(TridentState state, Fields inputFields,
    QueryFunction function, Fields functionFields) {
      projectionValidation(inputFields);
      String stateId = state._node.stateInfo.id;
      Node n = new ProcessorNode(_topology.getUniqueStreamId(),
                      _name,
                      TridentUtils.fieldsConcat(getOutputFields(),
                          functionFields),
                      functionFields,
                      new StateQueryProcessor(stateId, inputFields,
                          function));
      _topology._colocate.get(stateId).add(n);
      return _topology.addSourcedNode(this, n);
  }
```

Listing 8: Stream::stateQuery

从此处可以看出stateQueryNode最起码有两个inputStream，一是从TridentState而来表示状态已经改变，另一个是处于drpcStream这个方面的上一跳结点。

### 9.2.6 build

TridentTopology::build是将TridentTopology转变为StormTopology的过程，这一过程中最重要的一环就是将_graph中含有的node进行分组。

### 9.2.7 grouping

算法逻辑概述

1. 将boltNodes中的每个boltNode作为一个group加入全部加入initial-Groups

2. 以graph和initialGroups作为入参创建GraphGrouper

3. 分组的过程其实就是进行合并的过程，详见GraphGrouper::mergeFully()

   - 如果从当前group1的输出目的地都是属于group2，则将group1,group2合并
   - 如果当前group1的所有输入源都是来自于group2，则将group1，group2合并
   - 将需要合并的group1,group2作为入参创建新的group，同时将group1,group2从已有的集合出移除

```
    public void mergeFully() {
     boolean somethingHappened = true;
     while(somethingHappened) {
         somethingHappened = false;
         for(Group g: currGroups) {
             Collection<Group> outgoingGroups = outgoingGroups(g);
             if(outgoingGroups.size()==1) {
                 Group out = outgoingGroups.iterator().next();
                 if(out!=null) {
                     merge(g, out);
                     somethingHappened = true;
                     break;
                 }
             }

             Collection<Group> incomingGroups = incomingGroups(g);
             if(incomingGroups.size()==1) {
                 Group in = incomingGroups.iterator().next();
                 if(in!=null) {
                     merge(g, in);
                     somethingHappened = true;
                     break;
                 }
             }
         }
     }
  }
```

Listing 9: mergeFully

GraphGrouper::merge()

```
  private void merge(Group g1, Group g2) {
    Group newGroup = new Group(g1, g2);
    currGroups.remove(g1);
    currGroups.remove(g2);
```

```
        currGroups.add(newGroup);
        for(Node n: newGroup.nodes) {
            groupIndex.put(n, newGroup);
        }
    }
```

Listing 10: merge

在group之间添加partitionNode

```
// add identity partitions between groups
  for(IndexedEdge<Node> e: new HashSet<IndexedEdge>(graph.edgeSet())) {
      if(!(e.source instanceof PartitionNode) && !(e.target instanceof
          PartitionNode)) {
          Group g1 = grouper.nodeGroup(e.source);
          Group g2 = grouper.nodeGroup(e.target);
          // g1 being null means the source is a spout node
          if(g1==null && !(e.source instanceof SpoutNode))
              throw new RuntimeException("Planner exception: Null source
                  group must indicate a spout node at this phase of
                  planning");
          if(g1==null || !g1.equals(g2)) {
              graph.removeEdge(e);
              PartitionNode pNode = makeIdentityPartition(e.source);
              graph.addVertex(pNode);
              graph.addEdge(e.source, pNode, new IndexedEdge(e.source,
                  pNode, 0));
              graph.addEdge(pNode, e.target, new IndexedEdge(pNode, e.
                  target, e.index));
          }
      }
  }
```

Listing 11: add PartitionNode

_graph中所有的node在变换过后，变成两组元素，一是spoutNodes，另一个是合并后的mergedGroup.

spoutNodes中的每个元素作为spout添加到TridentTopologyBuilder的_spouts数组中，mergedGroup中的每个group添加到TridentTopologyBuilder的_bolt数组中。在TridentTopologyBuilder::build()中最主要的事情是为每个_spouts和_bolts数组中的成员添加grouping关系。

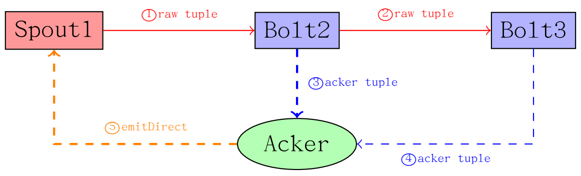
## 9.3 Storm Topology的ack机制

在进行TridentTopology的可靠性分析之前，我们先回顾一下在storm topology中的ack机制。ack bolt是在提交到storm cluster中，由系统自动产生的，一般来说一个topology只有一个ack bolt（当然可以通过配置参数指定多个）。

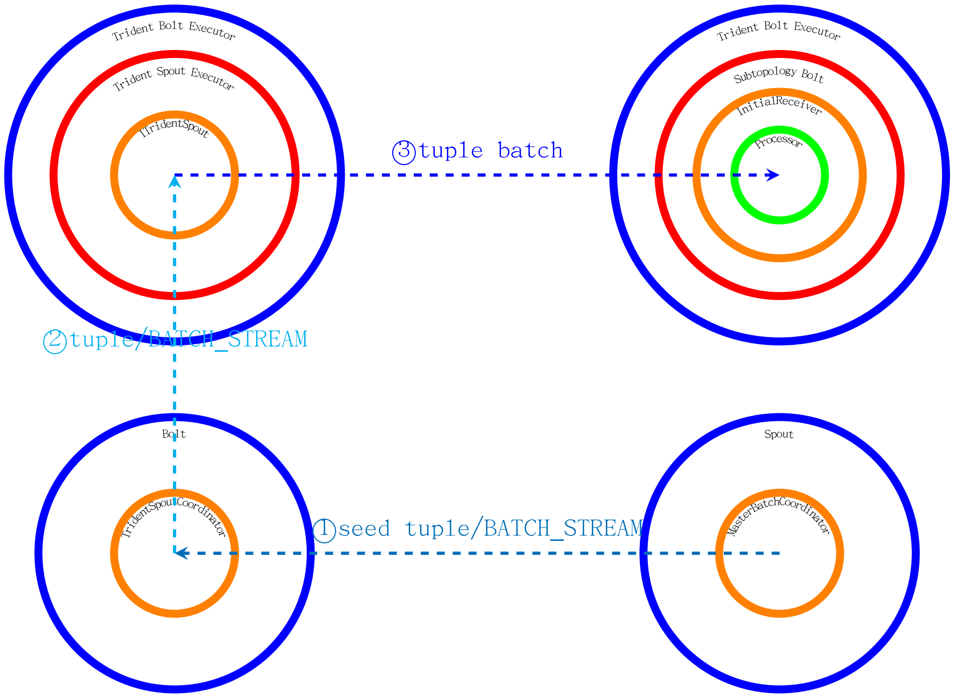
当bolt处理并下发完tuple给下一跳的bolt时，会发送一个ack给ack bolt。ack bolt通过简单的异或原理(即同一个数与自己异或结果为零)来判定从spout发出的某一个bolt是否已经被完全处理完毕。如果结果为真，ack bolt发送消息给spout，spout中的ack函数被调用并执行。如果超时，则发送fail消息给spout，spout中的fail函数被调用并执行，spout中的ack和fail的处理逻辑由用户自行填写。

如在github上的kerstel spout就能做到只有当某一个tuple被成功处理之后，它才会从缓存中移除，否则继续放入到处理队列再次进行处理。

下图是对上述文字所描述过程的形像展示



## 9.4 TridentTopology调用关系



上图是TridentTopology在转换成为vanilla topology在运行过程中的示意图，要点如下

1. 一个tridenttopoloy会至少引入一个MasterBatchCoordinator，这个MBC就类似于storm topology中的spout
2. newStream时使用的入参spout会裂变成两个bolt，一是TridentSpoutCoordinator,另一个是TridentSpoutExecutor
3. 针对stream的各种操作则被分散到各个Bolt中，它们的执行上下文是TridentBoltExecutor

可以看出使用TridentTopology Api进行操作时，所有的东西其实都运行在bolt context中，而真正的spout是在调用TridentTopologyBuilder.buildTopology()的时候被添加的。

MasterBatchCoordinator使用batch_stream发送一个类似于seeder tuple的东西给tridentspoutcoordinator，tridentspoutcoordinator将该信号继续下发给TridentSpoutExecutor，TridentSpout是如何一步步被调用到的呢。 调用关系如下所示

```
TridentBoltExecutor::execute
      TridentSpoutExecutor::execute
          BatchSpoutExecutor::execute
              ITridentSpout::emitBatch
```

emitBatch是产生真正需要被处理的tuple的，这些tuple会被各个Operation所在的bolt所接收。它们的调用顺序是

```
TridentBoltExecutor::execute
      SubtopologyBolt::execute
          InitialReceiver::receive
              TridentProcessor::execute
```

### 9.4.1 处理结束的判断依据

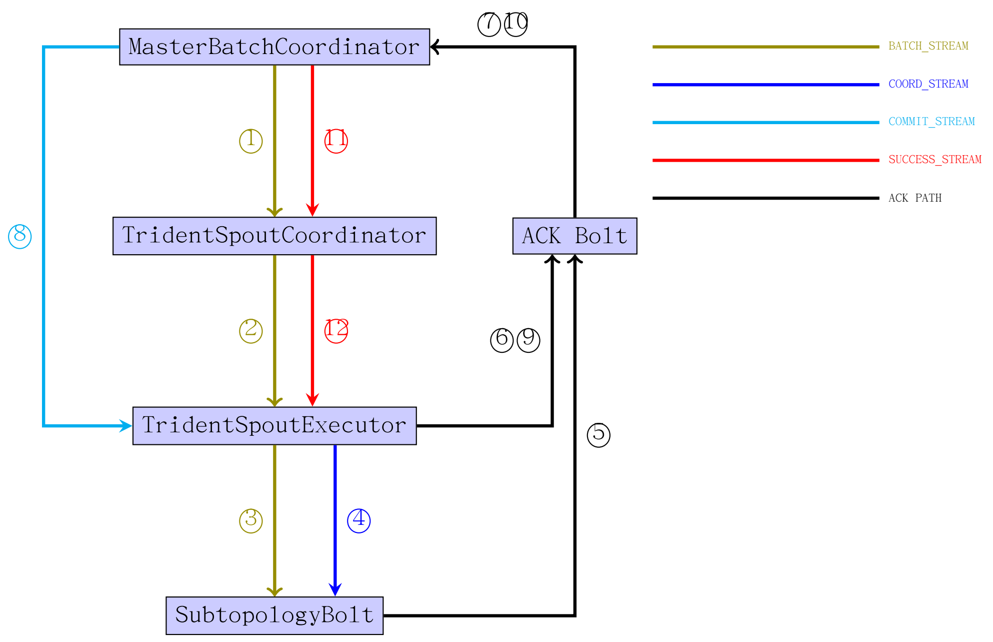
在TridentSpout中是如何判断所有的tuple都已经被处理的呢。

1. 在每跳中认为自己处理完毕的时候，它都会告诉下一跳，即下游，我给你发送了多少tuple，如果下游将上游发送过来的确认消息与自身确实已经处理的消息比对一致的话，则认为处理都完成，于是发送ack.
2. 问题的关键变成每一个bolt是如何判断自己已经处理完毕的呢，请看步骤3
3. 总有一个bolt是没有上游的，即TridentSpoutExecutor，它只会收到启动指令，但不接收真正的业务数据，于是它会告诉下一跳，我发了多少tuple给你。

在MasterBatchCoordinator中定义了三种不同的stream,这三种stream分别是

1. BATCH_STREAM
2. COMMIT_STREAM
3. SUCCESS_STREAM

这些stream分别在什么时候被使用呢，下图给出一个大概的时序



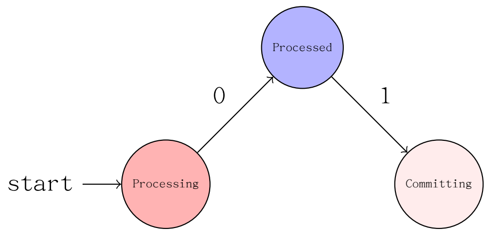
针对上图的简要说明

1. masterbatchcoordinator通过batch_stream发送seeder tuple给tridentspout-coordinator
2. tridentspoutcoordinator给tridentspoutexecutor继续传递该指令
3. TridentSpoutExecutor在收到启动指令后，调用ITridentSpout接口的实现类进行emitBatch
4. TridentSpoutExecutor在发送完一批batch后，finishBatch被调用，通过emitDirect会给下一跳通过coord_stream发送trackedinfo，即我已经发送了多少消息给你
5. TridentSpoutExecutor紧接着还会给ack bolt发送ack消息，ack bolt将其传达到MasterBatchCoordinator
6. MasterBatchCoordinator在收到第一个ack后，将状态置为processed
7. 当MasterBatchCoordinator再次收到ack后，会将状态转为committing，同时通过commit_stream发送tuple给TridentSpoutExecutor
8. 收到commit_stream上传来的tuple后，TridentSpoutExecutor会调用ITri-dentSpout中的emmitter，emmitter::commit()被执行，TridentSpoutEx-ecutor会再次ack收到tuple
9. MasterBatchCoordinator在收到这个tuple之后，会认为针对某一个seeder tuple的处理已经完全实现，于是通过SUCCESS_STREM告知TridentSpout-Coordinator，所有的活都已经都完成了，收工。

10. 收到Success_stream上传来的信号后，ITridentSpout中的内嵌子类Emmit和Coordinator中相应的success方法会被调用执行。

## 9.5 seeder tuple的状态机

在MasterBatchCoordinator中，针对每一个seeder tuple，其状态机如下图所示。注意这些状态是会被保存到zookeeper server中的，使用的api定义在TransactionalState中。



通过上面的分析可以看出，TridentTopology实现了一个比较好的框架，但真正要做到exactly-once的处理，还需要用户自己去实现ITridentSpout中的两个重要内嵌类，Emmitter和Coordinator。

具体如何实现该接口，可以查看storm-core/src/jvm/storm/trident/testing目录下的FixedBatchSpout.java和FeederCommitterBatchSpout.java