# Apache Storm 流式计算框架

## 1、Storm 基础

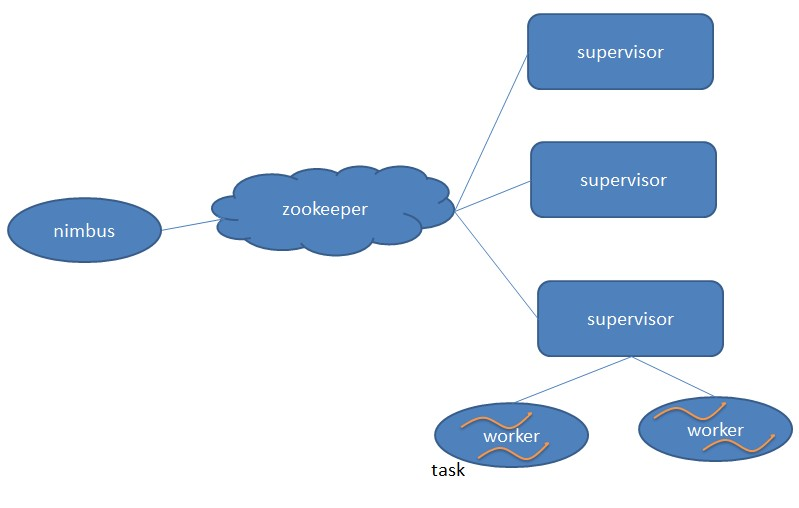
### 1.1、Storm是什么

* Hadoop在处理数据的时候，时效性不够，市场期望能够尽快得到处理后的数据。
* Storm是一个流式计算框架，数据源源不断的产生，源源不断的收集，源源不断的计算。

（一条数据一条数据的处理）

* Storm只负责数据的计算，不负责数据的存储。
* 2013年前后，阿里巴巴基于storm框架，使用java语言开发了类似的流式计算框架佳作，Jstorm。2016年年底阿里巴巴将源码贡献给了Apache storm，两个项目开始合并，新的项目名字叫做storm2.x。阿里巴巴团队专注flink开发。

### 1.2、Storm架构

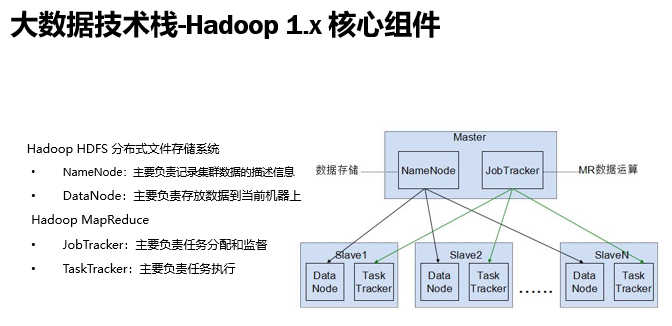


Nimbus：负责资源分配和任务调度。

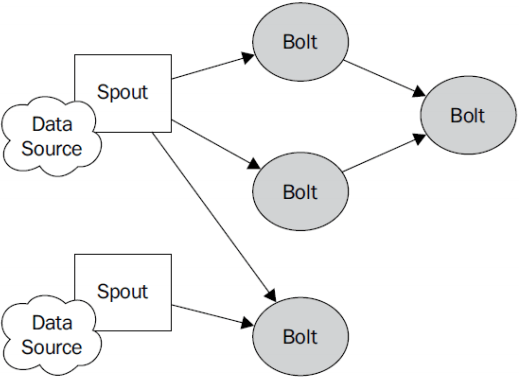
Supervisor：负责接受nimbus分配的任务，启动和停止属于自己管理的worker进程。

Worker：运行具体处理组件逻辑的进程。

Task：worker中每一个spout/bolt的线程称为一个task. 在storm0.8之后，task不再与物理线程对应，同一个spout/bolt的task可能会共享一个物理线程，该线程称为executor。



### 1.3、Storm编程模型



**Topology**：Storm中运行的一个实时应用程序，因为各个组件间的消息流动形成逻辑上的一个拓扑结构。

**Spout**：在一个topology中产生源数据流的组件。通常情况下spout会从外部数据源中读取数据，然后转换为topology内部的源数据。Spout是一个主动的角色，其接口中有个nextTuple()函数，storm框架会不停地调用此函数，用户只要在其中生成源数据即可。

**Bolt**：在一个topology中接受数据然后执行处理的组件。Bolt可以执行过滤、函数操作、合并、写数据库等任何操作。Bolt是一个被动的角色，其接口中有个execute(Tuple input)函数,在接受到消息后会调用此函数，用户可以在其中执行自己想要的操作。

**Tuple**：一次消息传递的基本单元。本来应该是一个key-value的map，但是由于各个组件间传递的tuple的字段名称已经事先定义好，所以tuple中只要按序填入各个value就行了，所以就是一个value list.

**Stream**：源源不断传递的tuple就组成了stream。

### 1.4、分组策略

Stream grouping：即消息的partition方法。

Stream Grouping定义了一个流在Bolt任务间该如何被切分。这里有Storm提供的6个Stream Grouping类型：

1. 随机分组(Shuffle grouping)：随机分发tuple到Bolt的任务，保证每个任务获得相等数量的tuple。 **跨服务器通信，浪费网络资源，尽量不适用**

2. 字段分组(Fields grouping)：根据指定字段分割数据流，并分组。例如，根据“user-id”字段，相同“user-id”的元组总是分发到同一个任务，不同“user-id”的元组可能分发到不同的任务。 **跨服务器，除非有必要，才使用这种方式。**

3. 全部分组(All grouping)：tuple被复制到bolt的所有任务。这种类型需要谨慎使用。 **人手一份，完全不必要**

4. 全局分组(Global grouping)：全部流都分配到bolt的同一个任务。明确地说，是分配给ID最小的那个task。 **欺负新人**

5. 无分组(None grouping)：你不需要关心流是如何分组。目前，无分组等效于随机分组。但最终，Storm将把无分组的Bolts放到Bolts或Spouts订阅它们的同一线程去执行(如果可能)。

6. 直接分组(Direct grouping)：这是一个特别的分组类型。元组生产者决定tuple由哪个元组处理者任务接收。 **点名分配 AckerBolt 消息容错**

**7.LocalOrShuffle 分组。 优先将数据发送到本地的Task，节约网络通信的资源。**

## 2、WordCount案例分析（重要）

### 2.1、 功能说明

设计一个topology，来实现对一个句子里面的单词出现的频率进行统计。

整个topology分为三个部分：

RandomSentenceSpout：数据源，在已知的英文句子中，随机发送一条句子出去。

SplitSentenceBolt:负责将单行文本记录（句子）切分成单词

WordCountBolt:负责对单词的频率进行累加

导入最新的依赖包

|  |
| --- |
| <**dependency**>  <**groupId**>org.apache.storm</**groupId**>  <**artifactId**>storm-core</**artifactId**>  <**version**>1.1.1</**version**> </**dependency**> |

### 2.2、TopologyMain 驱动类

|  |
| --- |
| **package** cn.itcast.realtime;   **import** org.apache.storm.Config; **import** org.apache.storm.LocalCluster; **import** org.apache.storm.StormSubmitter; **import** org.apache.storm.generated.AlreadyAliveException; **import** org.apache.storm.generated.AuthorizationException; **import** org.apache.storm.generated.InvalidTopologyException; **import** org.apache.storm.topology.TopologyBuilder;  */\*\*  \* 组装应用程序--驱动类  \*/* **public class** WordCountTopology {  **public static void** main(String[] args) **throws** InvalidTopologyException, AuthorizationException, AlreadyAliveException {  *//1、创建一个job(topology)* TopologyBuilder topologyBuilder = **new** TopologyBuilder();  *//2、设置job的详细内容* topologyBuilder.setSpout(**"ReadFileSpout"**,**new** ReadFileSpout(),1);  topologyBuilder.setBolt(**"SentenceSplitBolt"**,**new** SentenceSplitBolt(),1).shuffleGrouping(**"ReadFileSpout"**);  topologyBuilder.setBolt(**"WordCountBolt"**,**new** WordCountBolt(),1).shuffleGrouping(**"SentenceSplitBolt"**);  *//准备配置项* Config config = **new** Config();  config.setDebug(**false**);  *//3、提交job  //提交由两种方式：一种本地运行模式、一种集群运行模式。* **if** (args != **null** && args.**length** > 0) {  *//运行集群模式* config.setNumWorkers(1);  StormSubmitter.*submitTopology*(args[0],config,topologyBuilder.createTopology());  } **else** {  LocalCluster localCluster = **new** LocalCluster();  localCluster.submitTopology(**"wordcount"**, config, topologyBuilder.createTopology());  }  } } |

### 2.3、 ReadFileSpout

|  |
| --- |
| **package** cn.itcast.realtime;  **import** org.apache.storm.spout.SpoutOutputCollector; **import** org.apache.storm.task.TopologyContext; **import** org.apache.storm.topology.OutputFieldsDeclarer; **import** org.apache.storm.topology.base.BaseRichSpout; **import** org.apache.storm.tuple.Fields; **import** org.apache.storm.tuple.Values;  **import** java.util.Map;  */\*\*  \* Spout 需要继承一个模板  \*/* **public class** ReadFileSpout **extends** BaseRichSpout {  **private** SpoutOutputCollector **collector**;  */\*\*  \* Map conf 应用程序能够读取的配置文件  \* TopologyContext context 应用程序的上下文  \* SpoutOutputCollector collector Spout输出的数据丢给SpoutOutputCollector。  \*/* @Override  **public void** open(Map conf, TopologyContext context, SpoutOutputCollector collector) {  *//1、Kafka 连接 / MYSQL 连接 /Redis 连接  //****todo*** *//2、将SpoutOutputCollector复制给成员变量* **this**.**collector** = collector;  }   */\*\*  \* storm框架有个while循环，一直在nextTuple  \*/* @Override  **public void** nextTuple() {  *// 发送数据，使用collector.emit方法  // Values extends ArrayList<Object>* **collector**.emit(**new** Values(**"i love u"**));  **try** {  Thread.*sleep*(500);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }   @Override  **public void** declareOutputFields(OutputFieldsDeclarer declarer) {  declarer.declare(**new** Fields(**"biaobai"**));  } } |

### 2.4、 SentenceSplitBolt

|  |
| --- |
| **package** cn.itcast.realtime;  **import** org.apache.storm.task.OutputCollector; **import** org.apache.storm.task.TopologyContext; **import** org.apache.storm.topology.OutputFieldsDeclarer; **import** org.apache.storm.topology.base.BaseRichBolt; **import** org.apache.storm.tuple.Fields; **import** org.apache.storm.tuple.Tuple; **import** org.apache.storm.tuple.Values;  **import** java.util.Map;  */\*\*  \* 切割单词  \*/* **public class** SentenceSplitBolt **extends** BaseRichBolt {  **private** OutputCollector **collector**;  */\*\*  \* 初始化方法  \* Map stormConf 应用能够得到的配置文件  \* TopologyContext context 上下文 一般没有什么用  \* OutputCollector collector 数据收集器  \*/* @Override  **public void** prepare(Map stormConf, TopologyContext context, OutputCollector collector) {  **this**.**collector** = collector;  *//****todo 连接数据 连接redis 连接hdfs*** }   */\*\*  \* 有个while不停的调用execute方法，每次调用都会发一个数据进行来。  \*/* @Override  **public void** execute(Tuple input) { *// String sentence = input.getString(0);  // 底层先通过 biaobai 这个字段在map中找到对应的index角标值，然后再valus中获取对应数据。* String sentence = input.getStringByField(**"biaobai"**);  *//* ***todo 切割*** String[] strings = sentence.split(**" "**);  **for** (String word : strings) {  *//* ***todo 输出数据* collector**.emit(**new** Values(word,1));  }  }  @Override  **public void** declareOutputFields(OutputFieldsDeclarer declarer) {  *// 声明 输出的是什么字段* declarer.declare(**new** Fields(**"word"**,**"num"**));  } } |

### 2.5、 WordCountBolt

|  |
| --- |
| **package** cn.itcast.realtime;  **import** org.apache.storm.task.OutputCollector; **import** org.apache.storm.task.TopologyContext; **import** org.apache.storm.topology.OutputFieldsDeclarer; **import** org.apache.storm.topology.base.BaseRichBolt; **import** org.apache.storm.tuple.Fields; **import** org.apache.storm.tuple.Tuple;  **import** java.util.HashMap; **import** java.util.Map;  */\*\*  \* 计数  \*/* **public class** WordCountBolt **extends** BaseRichBolt {  **private** OutputCollector **collector**;  **private** HashMap<String, Integer> **wordCountMap**;   */\*\*  \* 初始化方法  \* Map stormConf 应用能够得到的配置文件  \* TopologyContext context 上下文 一般没有什么用  \* OutputCollector collector 数据收集器  \*/* @Override  **public void** prepare(Map stormConf, TopologyContext context, OutputCollector collector) {  **this**.**collector** = collector;  *//****todo 连接数据 连接redis 连接hdfs* wordCountMap** = **new** HashMap<String, Integer>();  }   */\*\*  \* 有个while不停的调用execute方法，每次调用都会发一个数据进行来。  \*/* @Override  **public void** execute(Tuple input) {  String word = input.getStringByField(**"word"**);  Integer num = input.getIntegerByField(**"num"**);  *// 先判断这个单词是否出现过* **if** (**wordCountMap**.containsKey(word)) {  Integer oldNum = **wordCountMap**.get(word);  **wordCountMap**.put(word, oldNum + num);  } **else** {  **wordCountMap**.put(word, num);  }  System.***out***.println(**wordCountMap**);  }   @Override  **public void** declareOutputFields(OutputFieldsDeclarer declarer) {  *// 声明 输出的是什么字段* declarer.declare(**new** Fields(**"fenshou"**));  } } |

### 2.6、 pom依赖

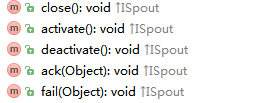
|  |
| --- |
| <**dependencies**>  <**dependency**>  <**groupId**>org.apache.storm</**groupId**>  <**artifactId**>storm-core</**artifactId**>  <**version**>1.1.1</**version**>  *<!-- 目前<scope>可以使用5个值：  \* compile，缺省值，适用于所有阶段，会随着项目一起发布。  \* provided，类似compile，期望JDK、容器或使用者会提供这个依赖。如servlet.jar。  \* runtime，只在运行时使用，如JDBC驱动，适用运行和测试阶段。  \* test，只在测试时使用，用于编译和运行测试代码。不会随项目发布。  \* system，类似provided，需要显式提供包含依赖的jar，Maven不会在Repository中查找它。 -->  <!--<scope>provided</scope>-->* </**dependency**> </**dependencies**> |

### 2.7、 项目编译

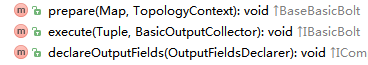
|  |
| --- |
| <**build**>  <**plugins**>  <**plugin**>  <**artifactId**>maven-assembly-plugin</**artifactId**>  <**configuration**>  <**descriptorRefs**>  <**descriptorRef**>jar-with-dependencies</**descriptorRef**>  </**descriptorRefs**>  <**archive**>  <**manifest**>  <**mainClass**>realtime.WordCountTopology</**mainClass**>  </**manifest**>  </**archive**>  </**configuration**>  <**executions**>  <**execution**>  <**id**>make-assembly</**id**>  <**phase**>package</**phase**>  <**goals**>  <**goal**>single</**goal**>  </**goals**>  </**execution**>  </**executions**>  </**plugin**>  <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-compiler-plugin</**artifactId**>  <**version**>3.7.0</**version**>  <**configuration**>  <**source**>1.8</**source**>  <**target**>1.8</**target**>  </**configuration**>  </**plugin**>  </**plugins**> </**build**> |

### 2.8、 Component生命周期

#### Spout生命周期



#### Bolt生命周期



#### Bolt的两个抽象类

BaseRichBolt 需要手动调ack方法

BaseBasicBolt由storm框架自动调ack方法

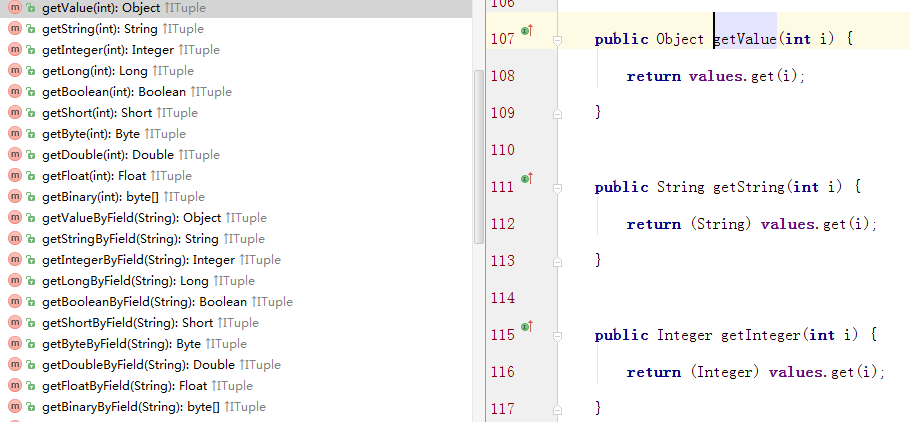
详见《第九章》

### 2.9、 StreamGrouping

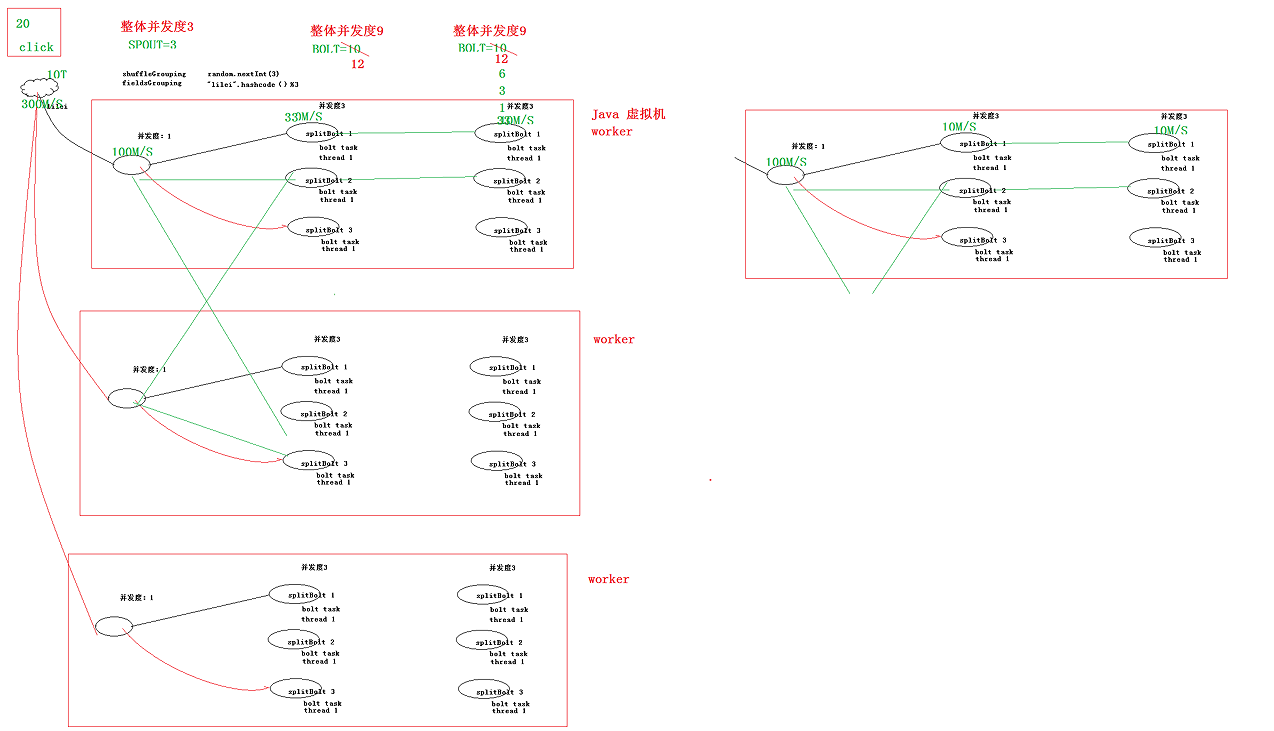
MKgrouper

|  |
| --- |
| **public** Map<List<Integer>, List<MsgInfo>> grouperBatch(List<MsgInfo> batch) {  Map<List<Integer>, List<MsgInfo>> ret = **new** HashMap<List<Integer>, List<MsgInfo>>();  *//optimize fieldGrouping & customGrouping* **if** (GrouperType.***local\_or\_shuffle***.equals(**grouptype**)) {  ret.put(**local\_shuffer\_grouper**.grouper(**null**), batch);  } **else if** (GrouperType.***global***.equals(**grouptype**)) {  *// send to task which taskId is 0* ret.put(JStormUtils.*mk\_list*(**out\_tasks**.get(0)), batch);  } **else if** (GrouperType.***fields***.equals(**grouptype**)) {  **fields\_grouper**.batchGrouper(batch, ret);  } **else if** (GrouperType.***all***.equals(**grouptype**)) {  *// send to every task* ret.put(**out\_tasks**, batch);  } **else if** (GrouperType.***shuffle***.equals(**grouptype**)) {  *// random, but the random is different from none* ret.put(**shuffer**.grouper(**null**), batch);  } **else if** (GrouperType.***none***.equals(**grouptype**)) {  **int** rnd = Math.*abs*(**random**.nextInt() % **out\_tasks**.size());  ret.put(JStormUtils.*mk\_list*(**out\_tasks**.get(rnd)), batch);  } **else if** (GrouperType.***custom\_obj***.equals(**grouptype**) || GrouperType.***custom\_serialized***.equals(**grouptype**)) {  **for** (**int** i = 0; i < batch.size(); i++ ) {  MsgInfo msg = batch.get(i);  List<Integer> out = **custom\_grouper**.grouper(msg.**values**);  List<MsgInfo> customBatch = ret.get(out);  **if** (customBatch == **null**) {  customBatch = JStormUtils.*mk\_list*();  ret.put(out, customBatch);  }  customBatch.add(msg);  }  } **else if** (GrouperType.***localFirst***.equals(**grouptype**)) {  ret.put(**localFirst**.grouper(**null**), batch);  } **else** {  ***LOG***.warn(**"Unsupportted group type"**);  }  **return** ret; } |

### 2.10、 Tuple是什么



### 并行度是什么



## 3、案例：实时交易数据统计

### 3.1 业务背景（重要）

根据订单mq，快速计算双11当天的订单量、销售金额。





### 3.2 架构设计及思路

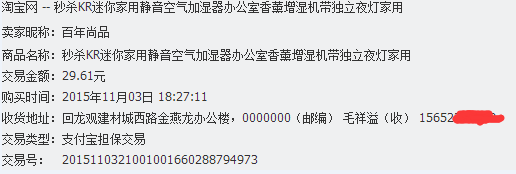
支付系统+kafka+storm/Jstorm集群+redis集群

1、支付系统发送mq到kafka集群中，编写storm程序消费kafka的数据并计算实时的订单数量、订单数量

2、将计算的实时结果保存在redis中

3、外部程序访问redis的数据实时展示结果

### 3.3 数据准备



订单编号、订单时间、支付编号、支付时间、商品编号、商家名称、商品价格、优惠价格、支付金额



### 3.4 业务口径



* 订单总数：一条支付信息当一条订单处理，假设订单信息不会重发（实际情况要考虑订单去重的情况，父子订单等多种情况），计算接收到MQ的总条数，即当做订单数。
* 销售额：累加所有的订单中商品的价格
* 支付金额：累加所有订单中商品的支付价格
* 用户人数：一条支付信息当一个人处理，假设订单一个人只下一单（实际情况要考虑用户去重的情况）。

整体淘宝的业务指标，每个品类，每个产品线，每个淘宝店，每个商品

Redis Key如何设计？

Index：{}pinlei}：{date} value

Index:1290:20160526 value

Index:1291:20160526 value

Index:1292:20160526 value

Index:1293:20160526 value

Index:1294:20160526 value

### 3.5 数据展示

读取redis中的数据，每秒进行展示，打印在控制台。

### 3.6 工程设计

* 数据产生：编写kafka数据生产者，模拟订单系统发送mq
* 数据输入：使用PaymentSpout消费kafka中的数据
* 数据计算：使用CountBolt对数据进行统计
* 数据存储：使用Sava2RedisBolt对数据进行存储，将结果数据存储到redis中
* 数据展示：编写java app客户端，访问redis，对数据进行展示，展示方式为打印在控制台。

1、获取外部数据源，MQSpout----Open(连接你的RMQ)---nextTuple()-----emit（json）

2、ParserPaymentInfoBolt()----execute(Tuple)------解析Json----JavaBean

productId,orderId,time,price（原价，订单价，优惠价，支付价），user，收货地址

total：原价、total：订单价、total：订单人数……

3、Save2ReidsBolt，保存相关业务指标

问题： 在redis中存放整个网站销售的原价， b:t:p:20160410 ---> value

redis: String----> value1+value2 + value3 + value4 incrBy

b:t:p:20160410

b:t:p:20161111

b:t:p:20160412

### 3.7 代码开发

#### 3.7.1 项目依赖

|  |
| --- |
| *<!-- storm core-->* <**dependencies**>  <**dependency**>  <**groupId**>org.apache.storm</**groupId**>  <**artifactId**>storm-core</**artifactId**>  <**version**>1.1.1</**version**>  <**scope**>provided</**scope**>  </**dependency**>  *<!-- storm kafka KafkaSpout-->  <!-- use new kafka spout code -->* <**dependency**>  <**groupId**>org.apache.storm</**groupId**>  <**artifactId**>storm-kafka-client</**artifactId**>  <**version**>1.1.1</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.apache.kafka</**groupId**>  <**artifactId**>kafka-clients</**artifactId**>  <**version**>0.10.0.0</**version**>  </**dependency**>  *<!-- redis jedis 依赖-->* <**dependency**>  <**groupId**>redis.clients</**groupId**>  <**artifactId**>jedis</**artifactId**>  <**version**>2.8.0</**version**>  </**dependency**>  *<!-- json Gson/fastjson-->* <**dependency**>  <**groupId**>com.google.code.gson</**groupId**>  <**artifactId**>gson</**artifactId**>  <**version**>2.4</**version**>  </**dependency**> </**dependencies**> |

#### 3.7.2 数据生产

|  |
| --- |
| **package** cn.itcast.realtime.kanban.producer; **import** cn.itcast.realtime.kanban.domain.PaymentInfo; **import** org.apache.kafka.clients.producer.KafkaProducer; **import** org.apache.kafka.clients.producer.ProducerRecord;  **import** java.util.Properties;  **public class** PaymentInfoProducer {  **public static void** main(String[] args){  *//1、准备配置文件* Properties props = **new** Properties();  props.put(**"bootstrap.servers"**, **"node01:9092"**);  */\*\*  \* 当生产者将ack设置为“全部”（或“-1”）时，min.insync.replicas指定必须确认写入被认为成功的最小副本数。  \* 如果这个最小值不能满足，那么生产者将会引发一个异常（NotEnoughReplicas或NotEnoughReplicasAfterAppend）。  \* 当一起使用时，min.insync.replicas和acks允许您执行更大的耐久性保证。  \* 一个典型的情况是创建一个复制因子为3的主题，将min.insync.replicas设置为2，并使用“全部”选项来产生。  \* 这将确保生产者如果大多数副本没有收到写入引发异常。  \*/* props.put(**"acks"**, **"all"**);  */\*\*  \* 设置一个大于零的值,将导致客户端重新发送任何失败的记录  \*/* props.put(**"retries"**, 0);  */\*\*  \* 只要有多个记录被发送到同一个分区，生产者就会尝试将记录一起分成更少的请求。  \* 这有助于客户端和服务器的性能。该配置以字节为单位控制默认的批量大小。  \*/* props.put(**"batch.size"**, 16384);  */\*\*  \*在某些情况下，即使在中等负载下，客户端也可能希望减少请求的数量。  \* 这个设置通过添加少量的人工延迟来实现这一点，即不是立即发出一个记录，  \* 而是等待达到给定延迟的记录，以允许发送其他记录，以便发送可以一起批量发送  \*/* props.put(**"linger.ms"**, 1);  */\*\*  \* 生产者可用于缓冲等待发送到服务器的记录的总字节数。  \* 如果记录的发送速度比发送给服务器的速度快，那么生产者将会阻塞，max.block.ms之后会抛出异常。  \* 这个设置应该大致对应于生产者将使用的总内存，但不是硬性限制，  \* 因为不是所有生产者使用的内存都用于缓冲。  \* 一些额外的内存将被用于压缩（如果压缩被启用）以及用于维护正在进行的请求。  \*/* props.put(**"buffer.memory"**, 33554432);  props.put(**"key.serializer"**, **"org.apache.kafka.common.serialization.StringSerializer"**);  props.put(**"value.serializer"**, **"org.apache.kafka.common.serialization.StringSerializer"**);  *//2、创建KafkaProducer* KafkaProducer<String, String> kafkaProducer = **new** KafkaProducer<String, String>(props);  **while** (**true**){  *//3、发送数据* kafkaProducer.send(**new** ProducerRecord<String, String>(**"itcast\_shop\_order"**,**new** PaymentInfo().random()));  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  } } |

#### 3.7.3 驱动类

|  |
| --- |
| **package** cn.itcast.realtime.kanban.Storm;  **import** org.apache.storm.Config; **import** org.apache.storm.LocalCluster; **import** org.apache.storm.StormSubmitter; **import** org.apache.storm.generated.AlreadyAliveException; **import** org.apache.storm.generated.AuthorizationException; **import** org.apache.storm.generated.InvalidTopologyException; **import** org.apache.storm.kafka.spout.KafkaSpout; **import** org.apache.storm.kafka.spout.KafkaSpoutConfig; **import** org.apache.storm.topology.TopologyBuilder;  */\*\*  \* 组装应用程序--驱动类  \*/* **public class** KanBanTopology {  **public static void** main(String[] args) **throws** InvalidTopologyException, AuthorizationException, AlreadyAliveException {  *//1、创建一个job(topology)* TopologyBuilder topologyBuilder = **new** TopologyBuilder();  *//2、设置job的详细内容* KafkaSpoutConfig.Builder<String, String> builder = KafkaSpoutConfig.*builder*(**"node01:9092"**,**"itcast\_shop\_order"**);  builder.setGroupId(**"bigdata\_kanban\_order"**);  KafkaSpoutConfig<String, String> kafkaSpoutConfig = builder.build();  topologyBuilder.setSpout(**"KafkaSpout"**,**new** KafkaSpout<String,String>(kafkaSpoutConfig), 1);  topologyBuilder.setBolt(**"ETLBolt"**,**new** ETLBolt(),1).shuffleGrouping(**"KafkaSpout"**);  topologyBuilder.setBolt(**"ProcessBolt"**,**new** ProcessBolt(),1).shuffleGrouping(**"ETLBolt"**);  *//准备配置项* Config config = **new** Config();  config.setDebug(**false**);  *//3、提交job  //提交由两种方式：一种本地运行模式、一种集群运行模式。* **if** (args != **null** && args.**length** > 0) {  *//运行集群模式* config.setNumWorkers(1);  StormSubmitter.*submitTopology*(args[0],config,topologyBuilder.createTopology());  } **else** {  LocalCluster localCluster = **new** LocalCluster();  localCluster.submitTopology(**"KanBanTopology"**, config, topologyBuilder.createTopology());  }  } } |

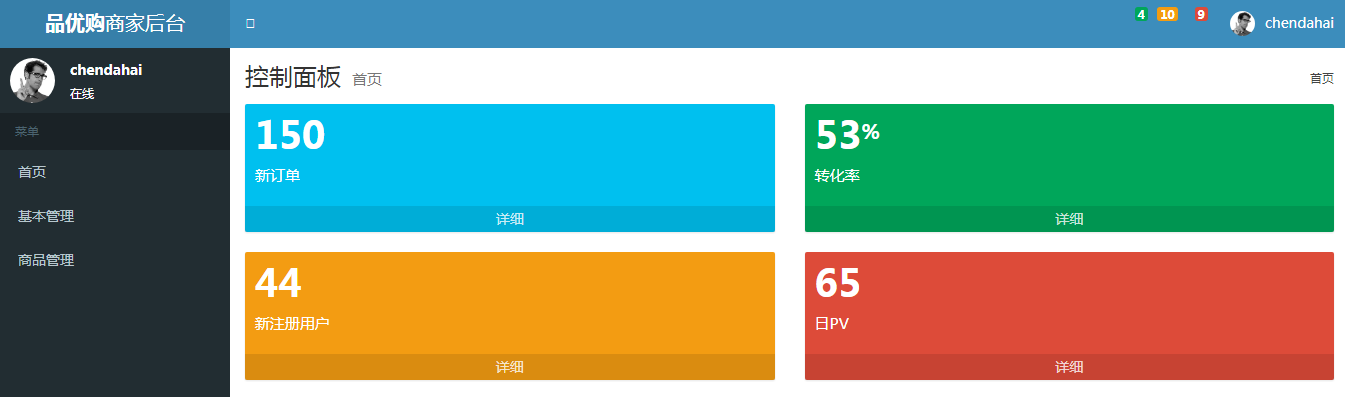
#### 3.7.4 ETLBolt

|  |
| --- |
| **package** cn.itcast.realtime.kanban.Storm;  **import** cn.itcast.realtime.kanban.domain.PaymentInfo; **import** com.google.gson.Gson; **import** org.apache.storm.task.OutputCollector; **import** org.apache.storm.task.TopologyContext; **import** org.apache.storm.topology.OutputFieldsDeclarer; **import** org.apache.storm.topology.base.BaseRichBolt; **import** org.apache.storm.tuple.Fields; **import** org.apache.storm.tuple.Tuple; **import** org.apache.storm.tuple.Values;  **import** java.util.Map;  **public class** ETLBolt **extends** BaseRichBolt {  **private** OutputCollector **collector**;   */\*\*  \* 初始化方法  \* Map stormConf 应用能够得到的配置文件  \* TopologyContext context 上下文 一般没有什么用  \* OutputCollector collector 数据收集器  \*/* @Override  **public void** prepare(Map stormConf, TopologyContext context, OutputCollector collector) {  **this**.**collector** = collector;  }   */\*\*  \* 有个while不停的调用execute方法，每次调用都会发一个数据进行来。  \*/* @Override  **public void** execute(Tuple input) {  String json = input.getString(4);  json = input.getStringByField(**"value"**);  *// 将json串转成 Java对象* Gson gson = **new** Gson();  PaymentInfo paymentInfo = gson.fromJson(json, PaymentInfo.**class**);  *// 其它的操作，比如说根据商品id查询商品的一级分类，二级分类，三级分类* **if**(paymentInfo!=**null**){  **collector**.emit(**new** Values(paymentInfo));  }  }   @Override  **public void** declareOutputFields(OutputFieldsDeclarer declarer) {  *// 声明 输出的是什么字段* declarer.declare(**new** Fields(**"paymentInfo"**));  } } |

#### 3.7.5 processBolt

|  |
| --- |
| **package** cn.itcast.realtime.kanban.Storm;  **import** cn.itcast.realtime.kanban.domain.PaymentInfo; **import** org.apache.storm.task.OutputCollector; **import** org.apache.storm.task.TopologyContext; **import** org.apache.storm.topology.OutputFieldsDeclarer; **import** org.apache.storm.topology.base.BaseRichBolt; **import** org.apache.storm.tuple.Tuple; **import** redis.clients.jedis.Jedis;  **import** java.util.Map;  **public class** ProcessBolt **extends** BaseRichBolt {  **private** Jedis **jedis**;   */\*\*  \* 初始化方法  \* Map stormConf 应用能够得到的配置文件  \* TopologyContext context 上下文 一般没有什么用  \* OutputCollector collector 数据收集器  \*/* @Override  **public void** prepare(Map stormConf, TopologyContext context, OutputCollector collector) {  **jedis** = **new** Jedis(**"redis"**, 6379);  }   */\*\*  \* 有个while不停的调用execute方法，每次调用都会发一个数据进行来。  \*/* @Override  **public void** execute(Tuple input) {  *//获取上游发送的javabean* PaymentInfo value = (PaymentInfo) input.getValue(0);  *//先计算总数据 来一条算一条* **jedis**.incrBy(**"kanban:total:ordernum"**,1);  **jedis**.incrBy(**"kanban:total:orderPrice"**,value.getPayPrice());  **jedis**.incrBy(**"kanban:total:orderuser"**,1);   *//计算商家（店铺的销售情况）* String shopId = value.getShopId();  **jedis**.incrBy(**"kanban:shop:"**+shopId+**":ordernum"**,1);  **jedis**.incrBy(**"kanban:shop:"**+shopId+**":orderPrice"**,value.getPayPrice());  **jedis**.incrBy(**"kanban:shop:"**+shopId+**":orderuser"**,1);   *//计算每个品类（品类id）一级品类* String Level1 = value.getLevel1();  **jedis**.incrBy(**"kanban:shop:"**+Level1+**":ordernum"**,1);  **jedis**.incrBy(**"kanban:shop:"**+Level1+**":orderPrice"**,value.getPayPrice());  **jedis**.incrBy(**"kanban:shop:"**+Level1+**":orderuser"**,1);  }   @Override  **public void** declareOutputFields(OutputFieldsDeclarer declarer) {  } } |

#### 3.7.6 看板



|  |
| --- |
| **package** cn.itcast.realtime.kanban.view;  **import** redis.clients.jedis.Jedis;  **public class** Kanban {  **public static void** main(String[] args) {  Jedis jedis = **new** Jedis(**"redis"**,6379);  **while** (**true**){  System.***out***.println(**"kanban:total:ordernum 指标是"**+jedis.get(**"kanban:total:ordernum"**));  System.***out***.println(**"kanban:total:orderPrice指标是"**+jedis.get(**"kanban:total:orderPrice"**));  System.***out***.println(**"kanban:total:orderuser指标是"**+jedis.get(**"kanban:total:orderuser"**));  System.***out***.println(**"---------------------------"**);  System.***out***.println();  **try** {  Thread.*sleep*(2000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  } } |

#### 3.7.7 打包运行

|  |
| --- |
| <**build**>  <**plugins**>  <**plugin**>  <**artifactId**>maven-assembly-plugin</**artifactId**>  <**configuration**>  <**descriptorRefs**>  <**descriptorRef**>jar-with-dependencies</**descriptorRef**>  </**descriptorRefs**>  <**archive**>  <**manifest**>  <**mainClass**>cn.itcast.realtime.kanban.Storm.KanBanTopology</**mainClass**>  </**manifest**>  </**archive**>  </**configuration**>  <**executions**>  <**execution**>  <**id**>make-assembly</**id**>  <**phase**>package</**phase**>  <**goals**>  <**goal**>single</**goal**>  </**goals**>  </**execution**>  </**executions**>  </**plugin**>  <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-compiler-plugin</**artifactId**>  <**version**>3.7.0</**version**>  <**configuration**>  <**source**>1.8</**source**>  <**target**>1.8</**target**>  </**configuration**>  </**plugin**>  </**plugins**> </**build**> |

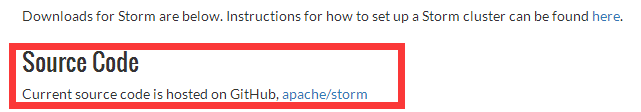
#### 3.7.8 数据对象

|  |
| --- |
| **import** com.google.gson.Gson;  **import** java.io.Serializable; **import** java.text.ParseException; **import** java.text.SimpleDateFormat; **import** java.util.Date; **import** java.util.Random; **import** java.util.UUID;  **public class** PaymentInfo **implements** Serializable {  **private static final long** serialVersionUID = -7958315778386204397L;  **private** String orderId;*//订单编号* **private** Date createOrderTime;*//订单创建时间* **private** String paymentId;*//支付编号* **private** Date paymentTime;*//支付时间* **private** String productId;*//商品编号* **private** String productName;*//商品名称* **private long** productPrice;*//商品价格* **private long** promotionPrice;*//促销价格* **private** String shopId;*//商铺编号* **private** String shopName;*//商铺名称* **private** String shopMobile;*//商品电话* **private long** payPrice;*//订单支付价格* **private int** num;*//订单数量   /\*\*  \* <Province>19</Province>  \* <City>1657</City>  \* <County>4076</County>  \*/* **private** String province; *//省* **private** String city; *//市* **private** String county;*//县   //102,144,114* **private** String catagorys;   **public** String getProvince() {  **return** province;  }   **public void** setProvince(String province) {  **this**.province = province;  }   **public** String getCity() {  **return** city;  }   **public void** setCity(String city) {  **this**.city = city;  }   **public** String getCounty() {  **return** county;  }   **public void** setCounty(String county) {  **this**.county = county;  }   **public** String getCatagorys() {  **return** catagorys;  }   **public void** setCatagorys(String catagorys) {  **this**.catagorys = catagorys;  }   **public** PaymentInfo() {  }   **public** PaymentInfo(String orderId, Date createOrderTime, String paymentId, Date paymentTime, String productId, String productName, **long** productPrice, **long** promotionPrice, String shopId, String shopName, String shopMobile, **long** payPrice, **int** num) {  **this**.orderId = orderId;  **this**.createOrderTime = createOrderTime;  **this**.paymentId = paymentId;  **this**.paymentTime = paymentTime;  **this**.productId = productId;  **this**.productName = productName;  **this**.productPrice = productPrice;  **this**.promotionPrice = promotionPrice;  **this**.shopId = shopId;  **this**.shopName = shopName;  **this**.shopMobile = shopMobile;  **this**.payPrice = payPrice;  **this**.num = num;  }   **public** String getOrderId() {  **return** orderId;  }   **public void** setOrderId(String orderId) {  **this**.orderId = orderId;  }   **public** Date getCreateOrderTime() {  **return** createOrderTime;  }   **public void** setCreateOrderTime(Date createOrderTime) {  **this**.createOrderTime = createOrderTime;  }   **public** String getPaymentId() {  **return** paymentId;  }   **public void** setPaymentId(String paymentId) {  **this**.paymentId = paymentId;  }   **public** Date getPaymentTime() {  **return** paymentTime;  }   **public void** setPaymentTime(Date paymentTime) {  **this**.paymentTime = paymentTime;  }   **public** String getProductId() {  **return** productId;  }   **public void** setProductId(String productId) {  **this**.productId = productId;  }   **public** String getProductName() {  **return** productName;  }   **public void** setProductName(String productName) {  **this**.productName = productName;  }   **public long** getProductPrice() {  **return** productPrice;  }   **public void** setProductPrice(**long** productPrice) {  **this**.productPrice = productPrice;  }   **public long** getPromotionPrice() {  **return** promotionPrice;  }   **public void** setPromotionPrice(**long** promotionPrice) {  **this**.promotionPrice = promotionPrice;  }   **public** String getShopId() {  **return** shopId;  }   **public void** setShopId(String shopId) {  **this**.shopId = shopId;  }   **public** String getShopName() {  **return** shopName;  }   **public void** setShopName(String shopName) {  **this**.shopName = shopName;  }   **public** String getShopMobile() {  **return** shopMobile;  }   **public void** setShopMobile(String shopMobile) {  **this**.shopMobile = shopMobile;  }   **public long** getPayPrice() {  **return** payPrice;  }   **public void** setPayPrice(**long** payPrice) {  **this**.payPrice = payPrice;  }   **public int** getNum() {  **return** num;  }   **public void** setNum(**int** num) {  **this**.num = num;  }   @Override  **public** String toString() {  **return "PaymentInfo{"** +  **"orderId='"** + orderId + **'\''** +  **", createOrderTime="** + createOrderTime +  **", paymentId='"** + paymentId + **'\''** +  **", paymentTime="** + paymentTime +  **", productId='"** + productId + **'\''** +  **", productName='"** + productName + **'\''** +  **", productPrice="** + productPrice +  **", promotionPrice="** + promotionPrice +  **", shopId='"** + shopId + **'\''** +  **", shopName='"** + shopName + **'\''** +  **", shopMobile='"** + shopMobile + **'\''** +  **", payPrice="** + payPrice +  **", num="** + num +  **'}'**;  }   **public** String random() {  **this**.orderId = UUID.randomUUID().toString().replaceAll(**"-"**, **""**);  **this**.paymentId = UUID.randomUUID().toString().replaceAll(**"-"**, **""**);  **this**.productPrice = **new** Random().nextInt(1000);  **this**.promotionPrice = **new** Random().nextInt(500);  **this**.payPrice = **new** Random().nextInt(480);  **this**.shopId = **new** Random().nextInt(200000)+**""**;   **this**.catagorys = **new** Random().nextInt(10000)+**","**+**new** Random().nextInt(10000)+**","**+**new** Random().nextInt(10000);  **this**.province = **new** Random().nextInt(23)+**""**;  **this**.city = **new** Random().nextInt(265)+**""**;  **this**.county = **new** Random().nextInt(1489)+**""**;   String date = **"2015-11-11 12:22:12"**;  SimpleDateFormat simpleDateFormat = **new** SimpleDateFormat(**"yyyy-MM-dd HH:mm:ss"**);  **try** {  **this**.createOrderTime = simpleDateFormat.parse(date);  } **catch** (ParseException e) {  e.printStackTrace();  }   **return new** Gson().toJson(**this**);  } } |

## 4、Storm源码下载及目录熟悉

### 4.1 在Storm官方网站上寻找源码地址

<http://storm.apache.org/downloads.html>



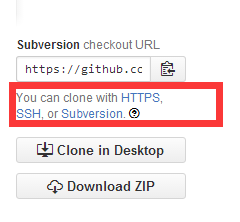
### 4.2 点击文字标签进入github

点击Apache/storm文字标签，进入github

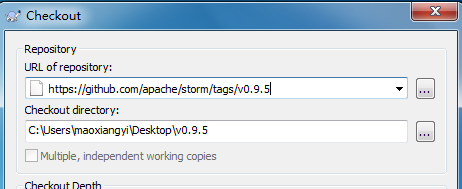
<https://github.com/apache/storm>

### 4.3 拷贝storm源码地址

在网页右侧，拷贝storm源码地址



### 4.4 使用Subversion客户端下载

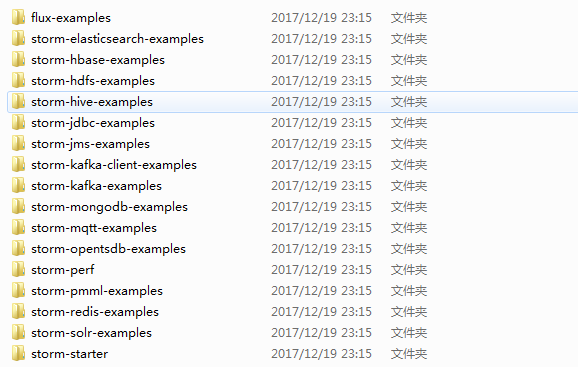


<https://github.com/apache/storm/tags/v0.9.5>

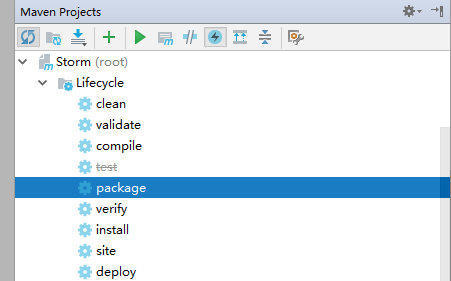
### 4.5 Storm源码目录分析



扩展包中的三个项目，使storm能与hbase、hdfs、kafka交互

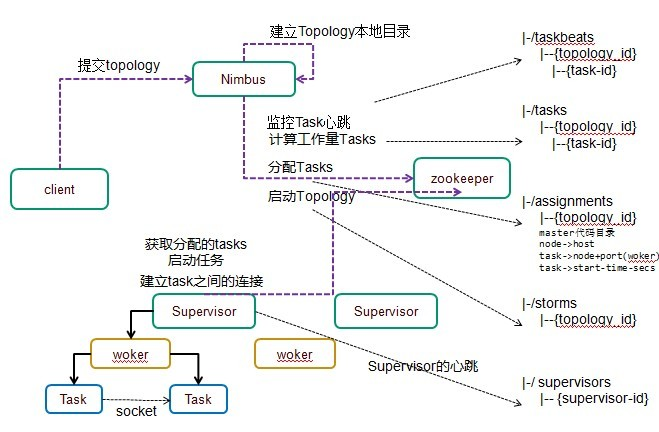


### 4.6 Storm源码编译

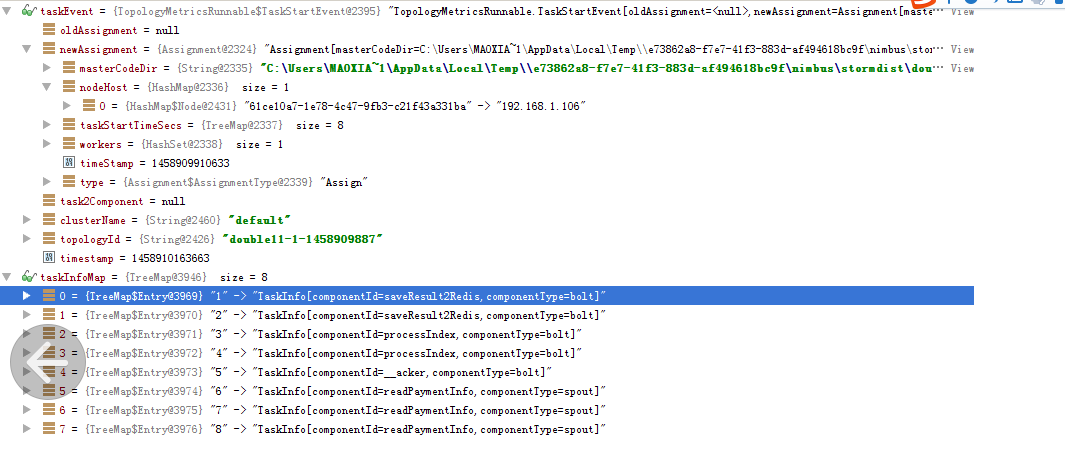


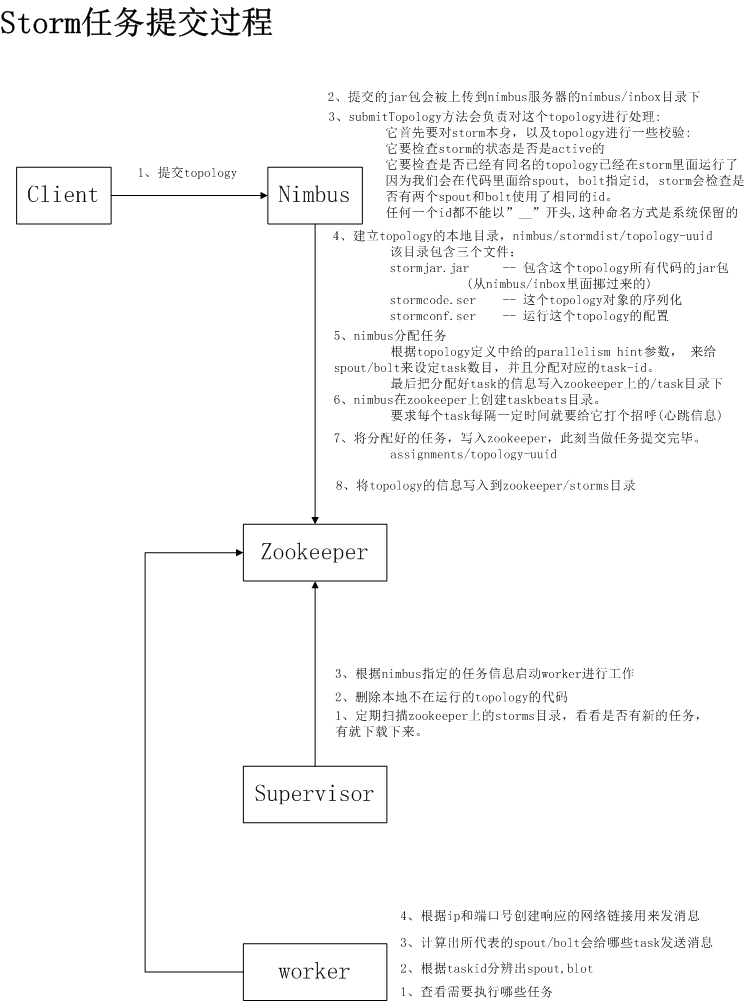
## 5、Storm原理

### 5.1 Storm 任务提交的过程



|  |
| --- |
| TopologyMetricsRunnable.TaskStartEvent[oldAssignment=<null>,newAssignment=Assignment[masterCodeDir=C:\Users\MAOXIA~1\AppData\Local\Temp\\e73862a8-f7e7-41f3-883d-af494618bc9f\nimbus\stormdist\double11-1-1458909887,nodeHost={61ce10a7-1e78-4c47-9fb3-c21f43a331ba=192.168.1.106},taskStartTimeSecs={1=1458909910, 2=1458909910, 3=1458909910, 4=1458909910, 5=1458909910, 6=1458909910, 7=1458909910, 8=1458909910},workers=[ResourceWorkerSlot[hostname=192.168.1.106,memSize=0,cpu=0,tasks=[1, 2, 3, 4, 5, 6, 7, 8],jvm=<null>,nodeId=61ce10a7-1e78-4c47-9fb3-c21f43a331ba,port=6900]],timeStamp=1458909910633,type=Assign],task2Component=<null>,clusterName=<null>,topologyId=double11-1-1458909887,timestamp=0] |

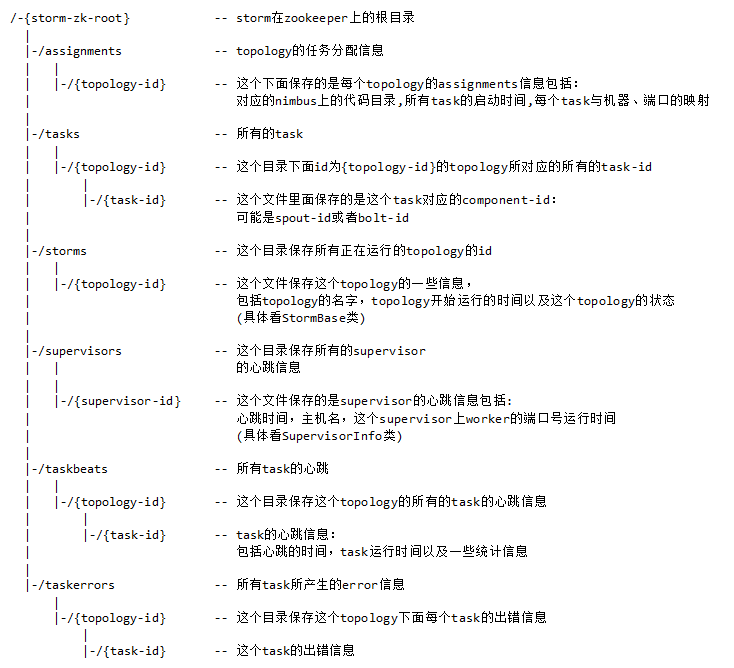




### 5.2 Storm组件本地目录树



### 5.3 Storm zookeeper目录树



### 5.4 Storm启动流程分析

------------程序员client------------------

1、客户端运行storm nimbus时，会调用storm的python脚本，该脚本中为每个命令编写一个方法，每个方法都可以生成一条相应的java命令。

命令格式如下：java -server xxxx.ClassName -args

nimbus---> Running: /export/servers/jdk/bin/java -server backtype.storm.daemon.nimbus

supervisor---> Running: /export/servers/jdk/bin/java -server backtype.storm.daemon.supervisor

--------------nimbus---------------------

2、nibums启动之后，接受客户端提交任务

命令格式：storm jar xxx.jar xxx驱动类 参数

Running: /export/servers/jdk/bin/java -client -Dstorm.jar=/export/servers/storm/examples/storm-starter/storm-starter-topologies-0.9.6.jar storm.starter.WordCountTopology wordcount-28

该命令会执行 storm-starter-topologies-0.9.6.jar 中的storm-starter-topologies-0.9.6.jar的main方法，main方法中会执行以下代码：

StormSubmitter.submitTopology("mywordcount",config,topologyBuilder.createTopology());

topologyBuilder.createTopology()，会将程序猿编写的spout对象和bolt对象进行序列化。

会将用户的jar上传到 nimbus物理节点的 /export/data/storm/workdir/nimbus/inbox目录下。并且改名，改名的规则是添加了一个UUID字符串。

在nimbus物理节点的 /export/data/storm/workdir/nimbus/stormdist目录下。有当前正在运行的topology的jar包和配置文件，序列化对象文件。

3、nimbus接受到任务之后，会将任务进行分配，分配会产生一个assignment对象，该对象会保存到zk中，目录是/storm/assignments ，该目录只保存正在运行的topology任务。

--------supervisor------------------

4、supervisor通过watch机制，感知到nimbus在zk上的任务分配信息，从zk上拉取任务信息，分辨出属于自己任务。

ResourceWorkerSlot[hostname=192.168.1.106,memSize=0,cpu=0,tasks=[1, 2, 3, 4, 5, 6, 7, 8],jvm=<null>,nodeId=61ce10a7-1e78-4c47-9fb3-c21f43a331ba,port=6900]

5、supervisor 根据自己的任务信息，启动自己的worker，并分配一个端口。

'/export/servers/jdk/bin/java' '-server' '-Xmx768m' export/data/storm/workdir/supervisor/stormdist/wordcount1-3-1461683066/stormjar.jar' 'backtype.storm.daemon.worker' 'wordcount1-3-1461683066' 'a69bb8fc-e08e-4d55-b51f-e539b066f90b' '6701' '9fac2805-7d2b-4e40-aabc-1c85c9856d64'

---------worker----------------------

6、worker启动之后，连接zk，拉取任务

ResourceWorkerSlot[hostname=192.168.1.106,memSize=0,cpu=0,tasks=[1, 2, 3, 4, 5, 6, 7, 8],jvm=<null>,nodeId=61ce10a7-1e78-4c47-9fb3-c21f43a331ba,port=6900]

假设任务信息：

1--->spout---type:spout

2--->bolt ---type:bolt

3--->acker---type:bolt

得到对象有几种方式？ new ClassName 创建对象、class.forName 反射对象、clone 克隆对象、序列化反序列化

worker通过反序列化，得到程序员自己定义的spout和bolt对象。

7、worker根据任务类型，分别执行spout任务或者bolt任务。

spout的声明周期是：open、nextTuple、outPutFiled

bolt的生命周期是：prepare、execute(tuple)、outPutFiled

### 5.5 启动流程代码说明

jstorm supervisor如何启动worker，worker如何启动task

1、下载Jstorm源码，在源码包下找到 daemon包，在这个包下有三个子包，分别是nimbus，supervisor，worker。

2、通过架构图，我们已知nimbus分配任务，并将任务信息写入到zk上，supervisor读取zk上的任务后启动自己的worker。所以我们分析supervisor如何启动worker，worker如何启动task。

3、supervisor如何启动worker。打开 com.alibaba.jstorm.daemon.supervisor.Supervisor 发现supervisor有几个方法，方法中有个mkSupervisor方法。

4、进去Supervisor中的mkSupervisor方法，在第144行有以下的代码，改代码创建了SyncSupervisorEvent 对象。

SyncSupervisorEvent syncSupervisorEvent =

new SyncSupervisorEvent(supervisorId, conf, syncSupEventManager, stormClusterState, localState, syncProcessEvent, hb);

5、SyncSupervisorEvent对象实现了RunnableCallback接口，该接口有个run方法会被定时执行。在run方法的191行，有代码如下，主要是要supervisor获取到任务信息，要开始准备启动worker了。

syncProcesses.run(zkAssignment, downloadFailedTopologyIds);

6、syncProcesses是com.alibaba.jstorm.daemon.supervisor.SyncProcessEvent的

引用变量，该类中有个自定义的run方法中有段代码如下，调用的startNewWorkers方法

startNewWorkers(keepPorts, localAssignments, downloadFailedTopologyIds);

7、SyncProcessEvent的startNewWorkers方法有代码片段如下，主要是根据集群模式启动不同模式下的worker。我们跟踪分布式集群模式下的worker启动。

for (Entry<Integer, LocalAssignment> entry : newWorkers.entrySet()) {

if (clusterMode.equals(“distributed”)) {

launchWorker(conf, sharedContext, assignment.getTopologyId(), supervisorId, port, workerId, assignment);

} else if (clusterMode.equals(“local”)) {

launchWorker(conf, sharedContext, assignment.getTopologyId(), supervisorId, port, workerId, workerThreadPids);

}

}

8、在分布式模式下worker启动最终会调用一个类似于java -server xxx.worker 启动worker。由于第7步中，有个for循环，该for循环会迭代出属于当前supervisor的所有worker任务并启动。

JStormUtils.launchProcess(cmd, environment, true);

9、java -server xxx.worker，命令执行之后，会执行Worker的mian方法。worker的main方法有代码如下，其实调用了worker自己内部的静态方法，叫做mk\_worker方法。

WorkerShutdown sd = mk\_worker(conf, null, topology\_id, supervisor\_id, Integer.parseInt(port\_str), worker\_id, jar\_path);

sd.join();

10、mk\_worker静态方法，会执行以下代码，创建一个worker的实例，并立即执行execute方法。

Worker w = new Worker(conf, context, topology\_id, supervisor\_id, port, worker\_id, jar\_path);

return w.execute();

11、execute方法会执行以下代码创建一个RefreshConnections 的实例。

RefreshConnections refreshConn = makeRefreshConnections();

12、makeRefreshConnections 方法会执行以下代码创建一个RefreshConnections 实例。

RefreshConnections refresh\_connections = new RefreshConnections(workerData);

13、RefreshConnections 是继承了 RunnableCallback，该实例的会有一个run方法会被定时执行。run方法中有以下代码，其中createTasks(addedTasks)方法用来创建Task任务。

shutdownTasks(removedTasks);

createTasks(addedTasks);

updateTasks(updatedTasks);

14、createTasks方法有代码如下，循环启动属于该worker的Task任务，启动Task任务主要调用Task.mk\_task(workerData, taskId);

for (Integer taskId : tasks) {

try {

TaskShutdownDameon shutdown = Task.mk\_task(workerData, taskId);

workerData.addShutdownTask(shutdown);

} catch (Exception e) {

LOG.error(“Failed to create task-” + taskId, e);

throw new RuntimeException(e);

}

}

15、Task.mk\_task(workerData, taskId)方法实现如下，创建一个Task对象并立即调用execute方法。

Task t = new Task(workerData, taskId);

return t.execute();

16、execute方法实现如下,用来初始化一个Executor，我们知道在默认情况下一个task等于一个executor。

RunnableCallback baseExecutor = prepareExecutor();

17、进入prepareExecutor()方法，代码如下，发现代码调用了mkExecutor方法。

final BaseExecutors baseExecutor = mkExecutor();

18、mkExecutor方法，代码如下，如果当前taskObj是Bolt就创建Bolt的executor，如果当前taskObj是Spout就创建相应的Spout executor。

public BaseExecutors mkExecutor() {

BaseExecutors baseExecutor = null;

if (taskObj instanceof IBolt) {

baseExecutor = new BoltExecutors(this);

} else if (taskObj instanceof ISpout) {

if (isSingleThread(stormConf) == true) {

baseExecutor = new SingleThreadSpoutExecutors(this);

} else {

baseExecutor = new MultipleThreadSpoutExecutors(this);

}

}

return baseExecutor;

}

19、创建完了executor，现在有两条线，分别是bolt executor和spout executor。以

bolt executor 为例，这个executor会实现Disruptor的EventHandler接口。 接口onevent方法需要实现，实现代码中会调用processTupleEvent()方法。下面节选onevent中的部分代码。

if (event instanceof Tuple) {

processControlEvent();

processTupleEvent((Tuple) event);

} else if (event instanceof BatchTuple) {

for (Tuple tuple : ((BatchTuple) event).getTuples()) {

processControlEvent();

processTupleEvent((Tuple) tuple);

}

}

20、进入processTupleEvent方法，发现有代码如下，其实最终是调用了bolt.execute()方法。

private void processTupleEvent(Tuple tuple) {

try {

if (xxx) {

backpressureTrigger.handle(tuple);

} else {

bolt.execute(tuple);

}

} catch (Throwable e) {

error = e;

LOG.error(“bolt execute error “, e);

report\_error.report(e);

}

}

## 6、Storm通信机制（了解）

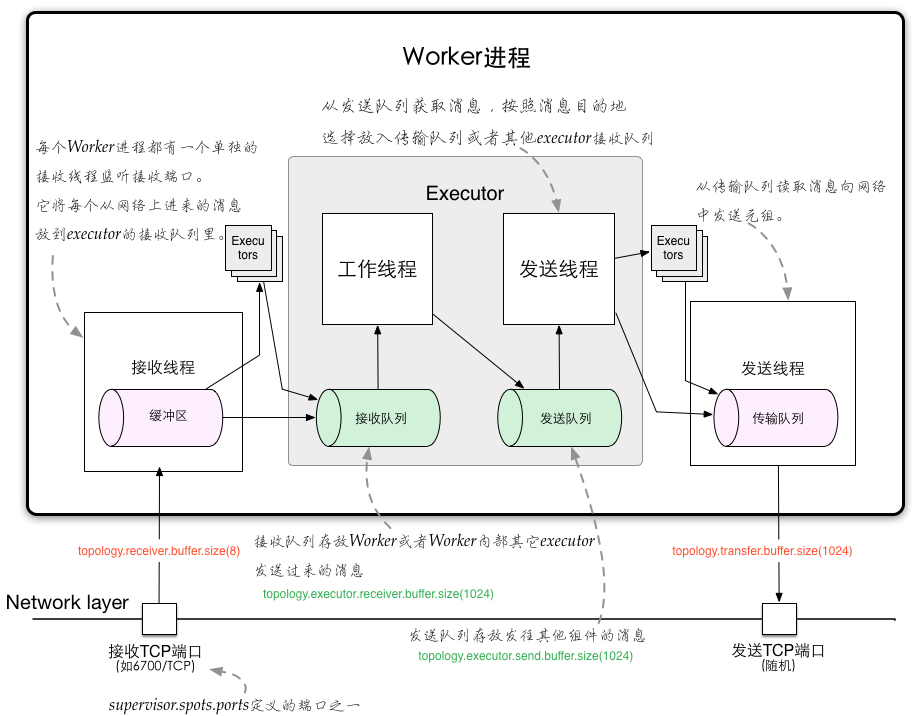
Worker间的通信经常需要通过网络跨节点进行，Storm使用ZeroMQ或Netty(0.9以后默认使用)作为进程间通信的消息框架。

Worker进程内部通信：**不同worker的thread通信使用LMAX Disruptor来完成。**

  不同topologey之间的通信，Storm不负责，需要自己想办法实现，例如使用kafka等；

### 6.1、 Worker进程间通信

worker进程间消息传递机制，消息的接收和处理的大概流程见下图



* + 对于worker进程来说，为了管理流入和传出的消息，每个worker进程有一个独立的接收线程(对配置的TCP端口supervisor.slots.ports进行监听);

对应Worker接收线程，每个worker存在一个独立的发送线程，它负责从worker的transfer-queue中读取消息，并通过网络发送给其他worker

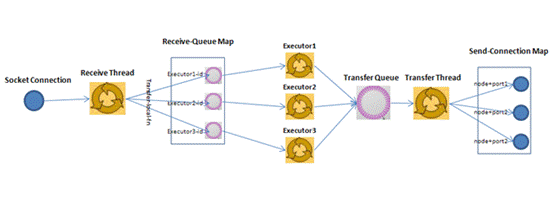
* + 每个executor有自己的incoming-queue和outgoing-queue。

Worker接收线程将收到的消息通过task编号传递给对应的executor(一个或多个)的incoming-queues;

每个executor有单独的线程分别来处理spout/bolt的业务逻辑，业务逻辑输出的中间数据会存放在outgoing-queue中，当executor的outgoing-queue中的tuple达到一定的阀值，executor的发送线程将批量获取outgoing-queue中的tuple,并发送到transfer-queue中。

* + 每个worker进程控制一个或多个executor线程，用户可在代码中进行配置。**其实就是我们在代码中设置的并发度个数。**

### 6.2、 Worker进程间通信分析



1. Worker接受线程通过网络接受数据，并根据Tuple中包含的taskId，匹配到对应的executor；然后根据executor找到对应的incoming-queue，将数据存发送到incoming-queue队列中。
2. 业务逻辑执行现成消费incoming-queue的数据，通过调用Bolt的execute(xxxx)方法，将Tuple作为参数传输给用户自定义的方法
3. 业务逻辑执行完毕之后，将计算的中间数据发送给outgoing-queue队列，当outgoing-queue中的tuple达到一定的阀值，executor的发送线程将批量获取outgoing-queue中的tuple,并发送到Worker的transfer-queue中
4. Worker发送线程消费transfer-queue中数据，计算Tuple的目的地，连接不同的node+port将数据通过网络传输的方式传送给另一个的Worker。
5. 另一个worker执行以上步骤1的操作。

### 6.3、Worker进程间技术(Netty、ZeroMQ)

#### Netty

Netty是一个NIO client-server(客户端服务器)框架，使用Netty可以快速开发网络应用，例如服务器和客户端协议。Netty提供了一种新的方式来使开发网络应用程序，这种新的方式使得它很容易使用和有很强的扩展性。Netty的内部实现时很复杂的，但是Netty提供了简单易用的api从网络处理代码中解耦业务逻辑。Netty是完全基于NIO实现的，所以整个Netty都是异步的。

书籍：Netty权威指南

#### ZeroMQ

ZeroMQ是一种基于消息队列的多线程网络库，其对套接字类型、连接处理、帧、甚至路由的底层细节进行抽象，提供跨越多种传输协议的套接字。ZeroMQ是网络通信中新的一层，介于应用层和传输层之间（按照TCP/IP划分），其是一个可伸缩层，可并行运行，分散在分布式系统间。

ZeroMQ定位为：一个简单好用的传输层，像框架一样的一个socket library，他使得Socket编程更加简单、简洁和性能更高。是一个消息处理队列库，可在多个线程、内核和主机盒之间弹性伸缩。ZMQ的明确目标是“成为标准网络协议栈的一部分，之后进入Linux内核”。

### 6.4、Worker 内部通信技术(Disruptor)

#### Disruptor的来历

* 一个公司的业务与技术的关系，一般可以分为三个阶段。第一个阶段就是跟着业务跑。第二个阶段是经历了几年的时间，才达到的驱动业务阶段。第三个阶段，技术引领业务的发展乃至企业的发展。所以我们在学习Disruptor这个技术时，不得不提LMAX这个机构，因为**Disruptor这门技术就是由LMAX公司开发并开源的**。
  + LMAX是在英国注册并受到FSA监管（监管号码为509778）的外汇黄金交易所。LMAX也是欧洲第一家也是唯一一家采用多边交易设施Multilateral Trading Facility（MTF）拥有交易所牌照和经纪商牌照的欧洲顶级金融公司
  + LAMX拥有最迅捷的交易平台，顶级技术支持。LMAX交易所使用“（MTF）分裂器Disruptor”技术，可以在极短时间内（一般在3百万秒之一内）处理订单，**在一个线程里每秒处理6百万订单**。所有订单均为撮合成交形式，无一例外。多边交易设施（MTF）曾经用来设计伦敦证券交易 所（london Stock Exchange）、德国证券及衍生工具交易所（Deutsche Borse）和欧洲证券交易所（Euronext）。
  + 2011年LMAX凭借该技术获得了金融行业技术评选大赛的最佳交易系统奖和甲骨文“公爵杯”创新编程框架奖。

#### Disruptor是什么

* 1. 简单理解：Disruptor是一个Queue。Disruptor是实现了“队列”的功能，而且是一个有界队列。而队列的应用场景自然就是“生产者-消费者”模型。
  2. 在JDK中Queue有很多实现类，包括不限于ArrayBlockingQueue、LinkBlockingQueue，这两个底层的数据结构分别是数组和链表。数组查询快，链表增删快，能够适应大多数应用场景。
  3. 但是ArrayBlockingQueue、LinkBlockingQueue都是线程安全的。涉及到线程安全，就会有synchronized、lock等关键字，这就意味着CPU会打架。
  4. Disruptor一种线程之间信息无锁的交换方式（使用CAS（Compare And Swap/Set）操作）。

#### Disruptor主要特点

1. 没有竞争=没有锁=非常快。
2. 所有访问者都记录自己的序号的实现方式，允许多个生产者与多个消费者共享相同的数据结构。
3. 在每个对象中都能跟踪序列号（ring buffer，claim Strategy，生产者和消费者），加上神奇的cache line padding，就意味着没有为伪共享和非预期的竞争。

#### Disruptor 核心技术点

Disruptor可以看成一个事件监听或消息机制，在队列中一边生产者放入消息，另外一边消费者并行取出处理.

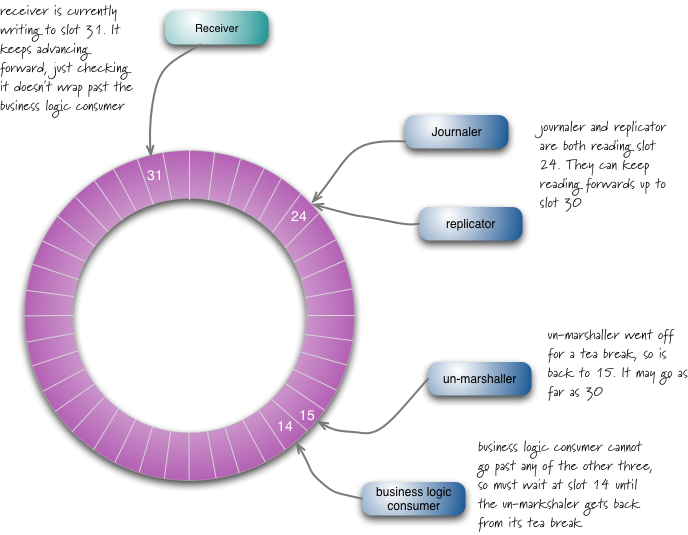
底层是单个数据结构：一个ring buffer。

每个生产者和消费者都有一个次序计算器，以显示当前缓冲工作方式。

每个生产者消费者能够操作自己的次序计数器的能够读取对方的计数器，生产者能够读取消费者的计算器确保其在没有锁的情况下是可写的。

**核心组件**

* Ring Buffer 环形的缓冲区，负责对通过 Disruptor 进行交换的数据（事件）进行存储和更新。
* Sequence 通过顺序递增的序号来编号管理通过其进行交换的数据（事件），对数据(事件)的处理过程总是沿着序号逐个递增处理。
* RingBuffer底层是个数组，次序计算器是一个64bit long 整数型，平滑增长。



1. 接受数据并写入到脚标31的位置，之后会沿着序号一直写入，但是不会绕过消费者所在的脚标。
2. Joumaler和replicator同时读到24的位置，他们可以批量读取数据到30

3、消费逻辑线程读到了14的位置，但是没法继续读下去，因为他的sequence暂停在15的位置上，需要等到他的sequence给他序号。如果sequence能正常工作，就能读取到30的数据。

## 7、消息不丢失机制

### 7.1、 ack是什么

ack 机制是storm整个技术体系中非常闪亮的一个创新点。

通过Ack机制，spout发送出去的每一条消息，都可以确定是被成功处理或失败处理， 从而可以让开发者采取动作。比如在Meta中，成功被处理，即可更新偏移量，当失败时，重复发送数据。

因此，通过Ack机制，很容易做到保证所有数据均被处理，一条都不漏。

**另外需要注意的，当spout触发fail动作时，不会自动重发失败的tuple，需要spout自己重新获取数据，手动重新再发送一次**

ack机制即， spout发送的每一条消息，

* 在规定的时间内，spout收到Acker的ack响应，即认为该tuple 被后续bolt成功处理
* 在规定的时间内，没有收到Acker的ack响应tuple，就触发fail动作，即认为该tuple处理失败，
* 或者收到Acker发送的fail响应tuple，也认为失败，触发fail动作

另外Ack机制还常用于限流作用： 为了避免spout发送数据太快，而bolt处理太慢，常常设置pending数，当spout有等于或超过pending数的tuple没有收到ack或fail响应时，跳过执行nextTuple， 从而限制spout发送数据。

通过conf.put(Config.TOPOLOGY\_MAX\_SPOUT\_PENDING, pending);设置spout pend数。

**这个timeout时间可以通过Config.TOPOLOGY\_MESSAGE\_TIMEOUT\_SECS来设定。Timeout的默认时长为30秒**

### 7.2、 如何使用Ack机制

spout 在发送数据的时候带上msgid

设置acker数至少大于0；Config.setNumAckers(conf, ackerParal);

在bolt中完成处理tuple时，执行OutputCollector.ack(tuple), 当失败处理时，执行OutputCollector.fail(tuple);

推荐使用IBasicBolt， 因为IBasicBolt 自动封装了OutputCollector.ack(tuple), 处理失败时，请抛出FailedException，则自动执行OutputCollector.fail(tuple)

### 7.3、 如何关闭Ack机制

有2种途径

spout发送数据是不带上msgid

设置acker数等于0

### 7.4、 基本实现

Storm 系统中有一组叫做"acker"的特殊的任务，它们负责跟踪DAG（有向无环图）中的每个消息。

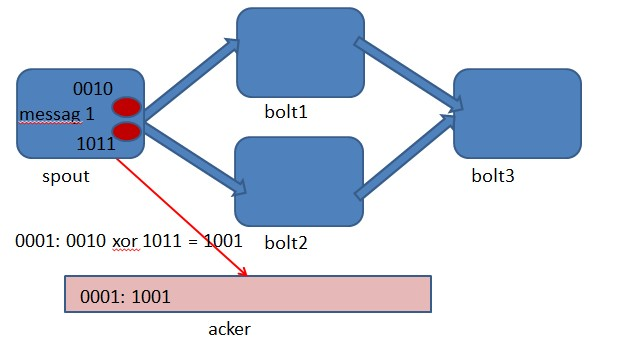
acker任务保存了spout id到一对值的映射。第一个值就是spout的任务id，通过这个id，acker就知道消息处理完成时该通知哪个spout任务。第二个值是一个64bit的数字，我们称之为"ack val"， 它是树中所有消息的随机id的异或计算结果。

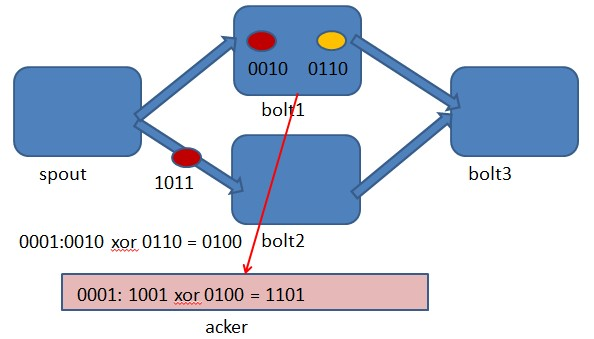
<TaskId,<RootId,ackValue>>

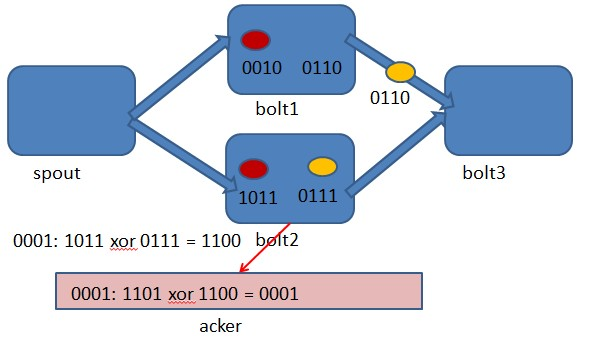
Spoutid,<系统生成的id,ackValue>

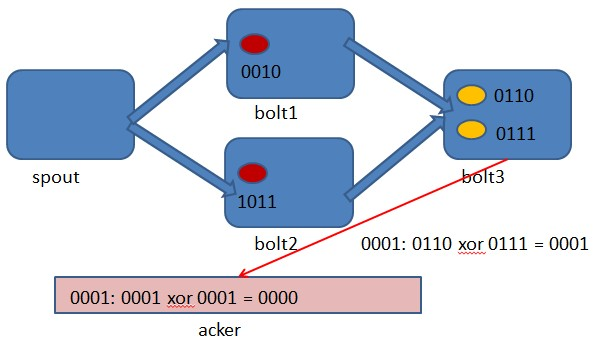
Task-0,64bit,0

ack val表示了整棵树的的状态，无论这棵树多大，只需要这个固定大小的数字就可以跟踪整棵树。当消息被创建和被应答的时候都会有相同的消息id发送过来做异或。 每当acker发现一棵树的ack val值为0的时候，它就知道这棵树已经被完全处理了









## 8、案例：实时日志监控告警系统

### 8.1、课程目标

* 介绍企业监控系统的一种实现方式
* 集成Flume+Kafka+Storm+Redis
* 配置Flume脚本及自定义Flume的拦截器
* 规划Kafka集群及设计Topic的分片&副本数量
* Storm整合Mysql数据库
* Storm集成发送短信、邮件功能
* 定时将数据从mysql数据中更新到Storm程序
* 学生动手开发一套流式计算的程序

### 8.2、开发步骤

1）创建数据库的表架构、初始化业务系统名称、业务系统需要监控的字段、业务系统的开发人员（手机号、邮箱地址）

2）编写Flume脚本去收集数据

3）创建Kafka的topic

4）编写Storm程序。

### 8.3、数据库表结构

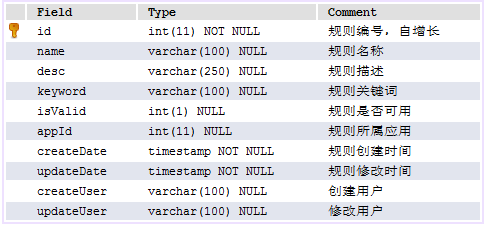
#### 1）用户表（开发人员表）

用户编号、用户名称、用户手机号、用户邮箱地址、是否可用

#### 2）应用程序表

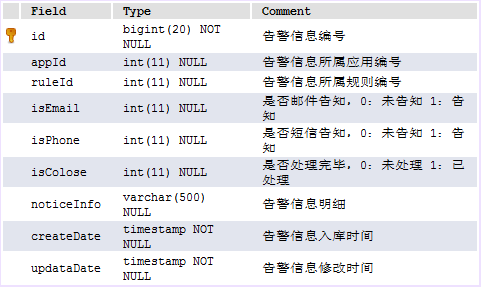
用来保存应用的信息，包括应用名称、应用描述、应用是否在线等信息

#### 3）规则表（每个应用系统要监控哪些规则）



#### 4）结果表

用来保存触发规则后的记录，包括告警编号、是否短信告知、是否邮件告知、告警明细等信息。



### 8.4、Flume+Kafka整合

1）启动zookeeper



2）启动 Kafka



3）创建 Kafka Topic

kafka-topics.sh --create --zookeeper zk01:2181 --topic system\_log --partitions 6 --replication-factor 2

4）配置 Flume配置文件

|  |
| --- |
| [root@node01 nginx\_flume\_kafka]# cat exec.conf  a1.sources = r1  a1.channels = c1  a1.sinks = k1  a1.sources.r1.type = exec  a1.sources.r1.command = tail -F /export/data/flume/click\_log/data.log  a1.sources.r1.channels = c1  a1.channels.c1.type=memory  a1.channels.c1.capacity=10000  a1.channels.c1.transactionCapacity=100  a1.sinks.k1.type = org.apache.flume.sink.kafka.KafkaSink  a1.sinks.k1.topic = system\_log  a1.sinks.k1.brokerList = kafka01:9092  a1.sinks.k1.requiredAcks = 1  a1.sinks.k1.batchSize = 20  a1.sinks.k1.channel = c1 |
|  |

5）启动 Flume

|  |
| --- |
| flume-ng agent -n a1 -c /export/servers/flume/conf -f /export/servers/flume/myconfig/nginx\_flume\_kafka/exec.conf -Dflume.root.logger=INFO,console |

6）启动模拟器生成日志

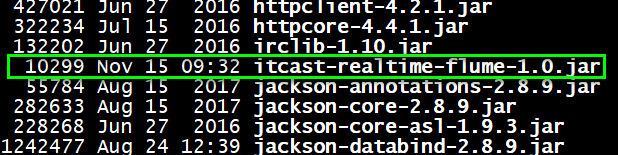
|  |
| --- |
| [root@node01 nginx\_flume\_kafka]# pwd  /export/servers/flume/myconfig/nginx\_flume\_kafka  [root@node01 nginx\_flume\_kafka]# cat click\_log\_out.sh  for((i=0;i<=500000;i++));  do echo "i am lilei "+$i >> /export/data/flume/click\_log/data.log;  done |

7）启动Kafka Consumer

|  |
| --- |
| kafka-console-consumer.sh --zookeeper zk01:2181 --from-beginning –topic system\_log |

### 8.5、Flume拦截器使用

1）上传资料中的文件到 flume的lib包下

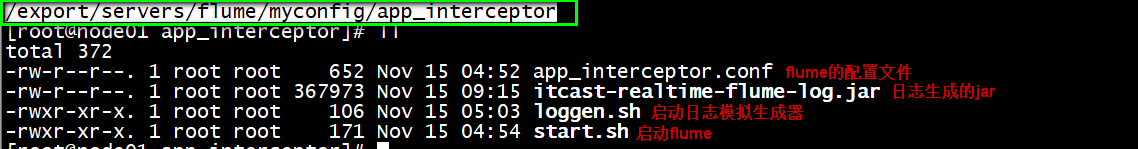


2）生产模拟日志生成器和flume配置文件到集群上



上传到（没有目录就创建）

/export/servers/flume/myconfig/app\_interceptor



3）启动模拟日志生成器



4）创建topic

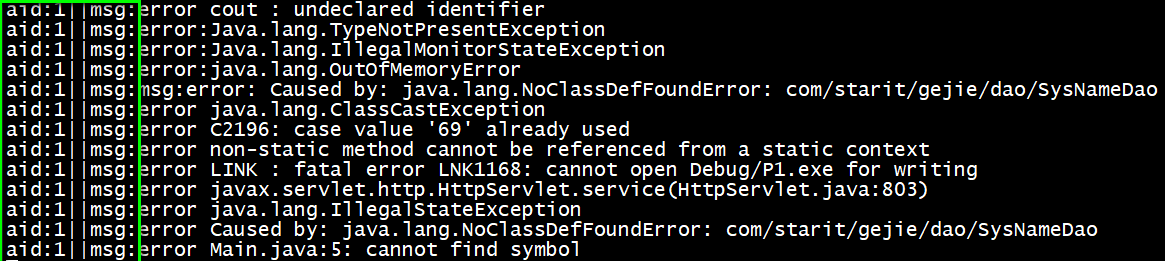
kafka-topics.sh --create --zookeeper zk01:2181 --topic log\_monitor --partitions 6 --replication-factor 2

5）启动Flume



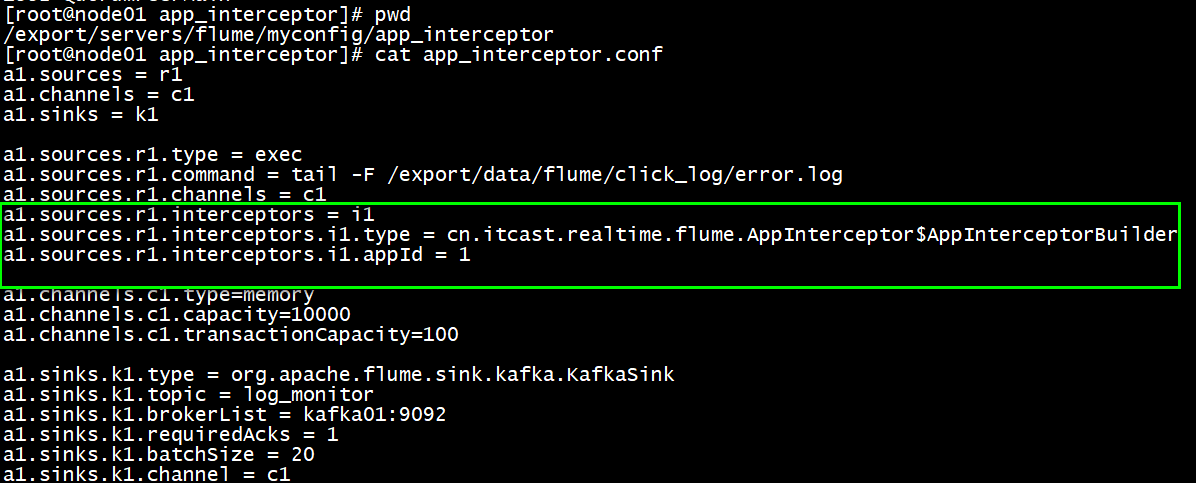
6）启动Kafka 消费者

kafka-console-consumer.sh --zookeeper zk01:2181 --from-beginning --topic log\_monitor



### 8.6、Flume拦截器实现

1）配置 在flumesource上配置拦截器

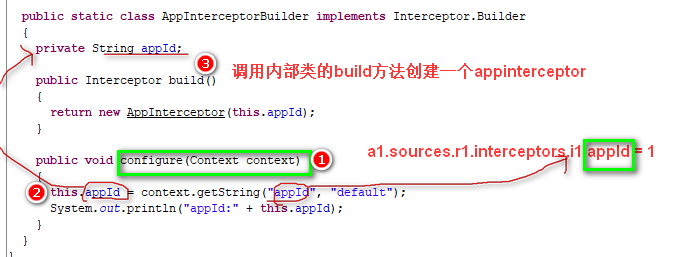


2）使用反编译工具查看 AppInterceptorBuilder

Configure方法在类初始化之后会被调用，由flume框架传入context上下文对象

从context上下文中获取配置文件中配置的appid配置项

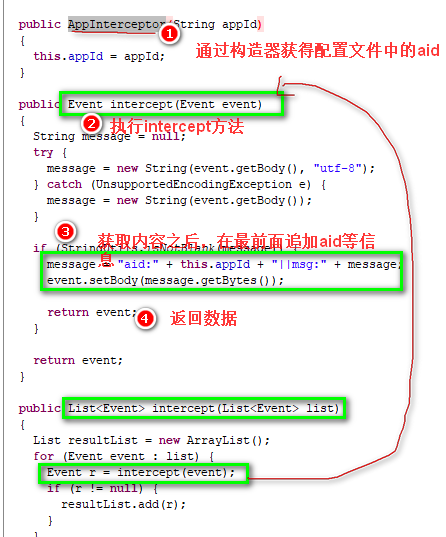
使用build方法构建AppInterceptor类，并将appid传入到AppInterceptor



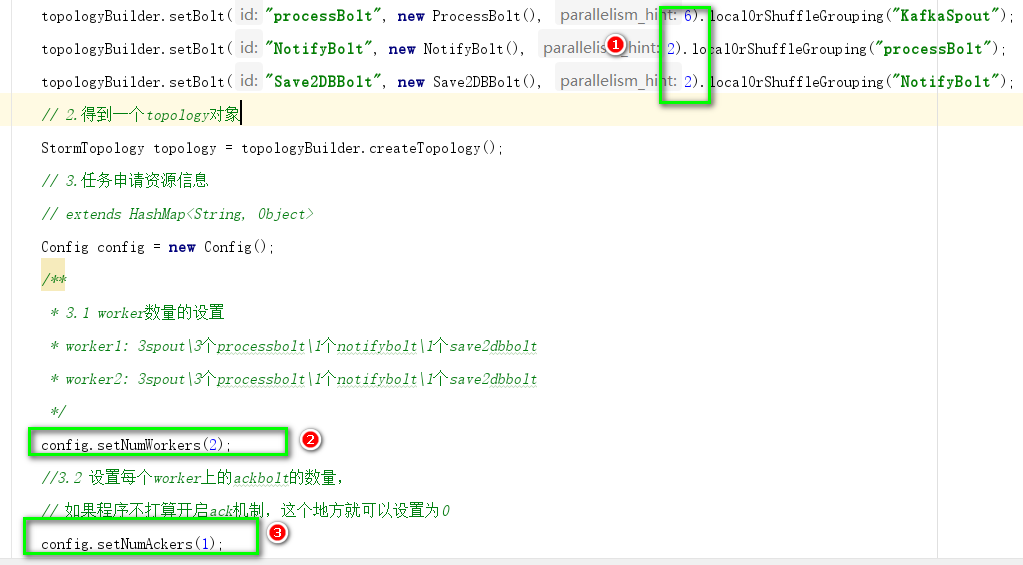
3) 使用 反编译工具查看AppInterceptor

通过构造器获得AppInterceptorBuilder传入的appid

在拦截器方法调用的时候，对原始数据进行获取，然后添加内容。



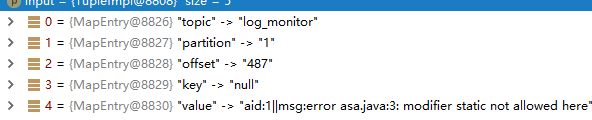
### 8.7 Storm程序worker&Task数量设置



1） KafkaSpout并行度初始化状态



2) KafkaSpout读取数据的5个字段



### 8.8 业务逻辑

1）KafkaSpout 负责读取数据。并行度设置6个，在一个消费组中启动6个消费者（Task）

2）ProcessBolt 负责检验数据中是否包含关键词

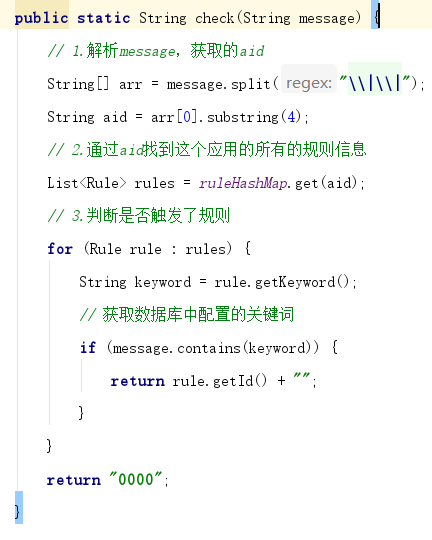
读取上游发送的数据，上游的数据会有五个字段。Value字段就是我们需要的。

*aid:1||msg:error java.lang.ArrayIndexOutOfBoundsException*

对数据进行分割，得到应用的编号和错误的信息。

通过应用编号获得应用所有的规则信息。

迭代所有的规则信息，如果有一个规则被触发了，直接返回这个规则的编号。



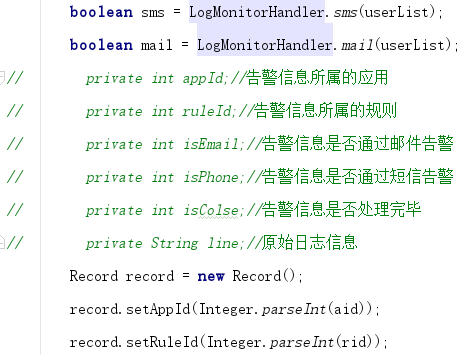
返回规则编号，如果不是0000，就向下游发送数据。

3）notifyBolt 发送短信和发送邮件等信息

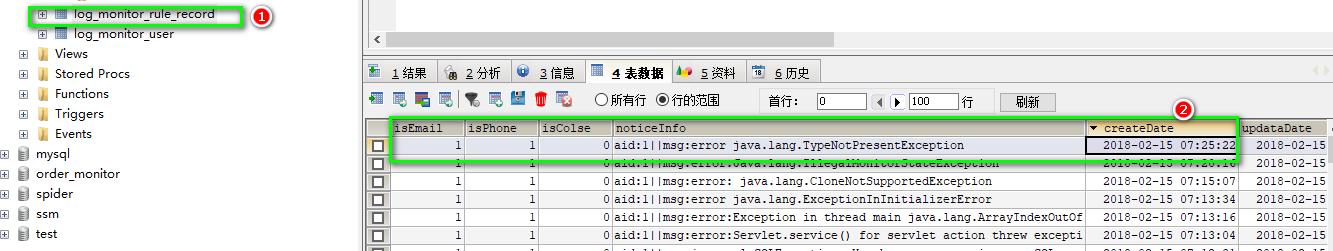
先判断这个应用的这个规则，是否是在五分钟内已经发送过短信。

如果没有发过，发送短信（聚合数据API,3分钱一条）、发送邮件（自己编写）

拼装以下触发规则的信息，准备保存到数据库。



4）save2db 将触发规则的信息保存到数据库



## 9、案例：点击流日志分析系统

用户在网站上操作的任何行为，都会被悄悄记录下来（以数据的方式）。

### 9.1、课程目标

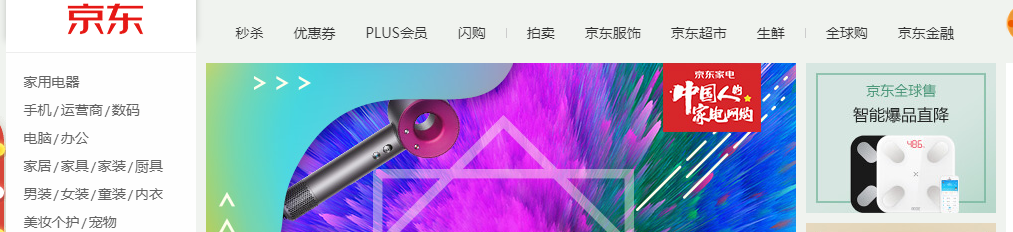
* 介绍点击流日志产生、收集、部署
* 集成点击流系统+Flume+Kafka+Storm+Redis
* 数据分析业务知识【重点】
* 学生动手开发一套流式计算的程序

我们想知道用户在网页中点击了哪些标签、每个标签被点击了多少次。

分析场景（一）-黑马程序员



分析场景（二）-京东商城



分析场景（三）-京东专题页分析



### 9.2、如何进行点击流日志分析

1. 用户访问<http://yun.itheima.com/open/c-139.html> 是否可以理解为是从‘云计算大数据’标签过来的？

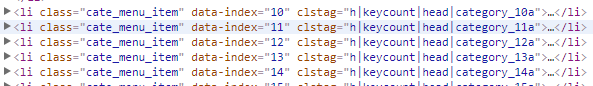
答案：不一定，因为用户可以从其他地方访问这个url。

解决：最好的办法，当用户点击了‘云计算大数据’的标签，通过js代码向后台发送一个数据，这个数据就记录了用户点击过这个标签。

User info、标签唯一标识、点击事件、浏览器尺寸等等。

1. 如何给网页中一个标签设置一个唯一的标志？

京东：在京东上，通过给每个标签设置一个clstag用来标志标签的唯一性。



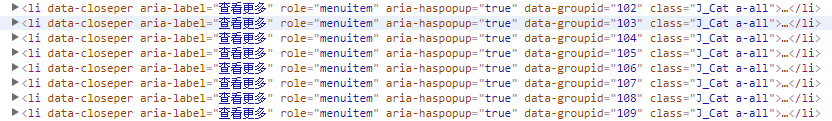
苏宁：苏宁是通过在每个标签上设置一个cms-name属性来标志标签的唯一性



国美：国美是通过在每个标签上这是一个data-code属性来标志标签的唯一性

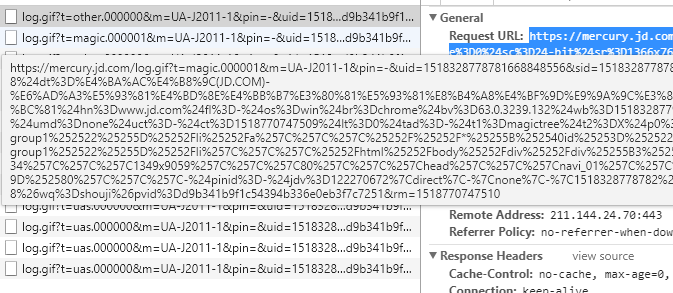


淘宝：疑似通过data-groupid进行唯一标志



1. 如何通过js代码发送日志

京东：当用户点击或者浏览某个页面的时候，后台有js会发送类似于以下的信息给服务器。



1. 解析出点击流日志中的字段信息

1、来源URL ref

2、当前URL req http://www.itcast.cn/121212.html

3、请求时间 reqTime （用户打开网站的时间）

4、操作类型 type （0：浏览操作，1：点击操作）

9、页面停留时间

10、用户IP地址

11、sessionID

12、用户的账号信息

13、点击什么东西 clstag

5、操作系统

6、浏览器的信息 br

7、屏幕尺寸

8、鼠标点击的位置

### 9.3、一些简单的数据分析需求

1）计算“京东首页”中来自于百度的流量信息

Select count(\*) from 点击表 where requestUrl = “京东首页” and refUrl =“百度”；

2）计算‘“京东首页”中来自于百度的流量信息中chrome浏览器占比

Select count(浏览器类型) from 点击流表 where requestUrl=”京东首页” and refUrl=“百度” group by 浏览器类型。

3）计算“京东首页”中来自于百度流量的信息中各省份的流量占比情况

需要些一个UDF函数，将IP地址转化成省市县信息。

新的挑战：

虽然通过sql语句可以很好的计算出各种数据分析的需求。

但是，一个网站需要分析的页面会很多需要分析、**很多标签需要计算指标、很多新上线的页面和标签……**

**解决办法：**

**开发一个自动计算指标的系统，由产品经理输入相关的判断条件，程序自动给出结果。解放程序员的双手。**

### 9.4、业务逻辑梳理

1）Storm部分 主要计算pv，uv

KafkaSpout:读取点击流收集系统收集的日志信息。不同公司的日志信息不一样，需要通过nginx+lua对日志进行标准化。 \t分割。

EtlBolt ：将数据解析成一个Java对象。Java对象中有很多字段，这些字段名称会显示在网页，给产品经理去选择。

ProcessBolt：读取数据库的所有Job信息，判断Job是否被触发（所有条件是否被满足）。如果触发了Job，就开始为Job计算pv，uv。 将结果保存到Redis中

2）app部分 主要计算增量数据

计算每个Job的pv的增量数据、uv的增量数据。以便展示在网页上称为一个趋势图。

1. 读取数据库中所有的Job信息。得到JobId，通过一定的规则生成redis的key，去获取最新的值。
2. 用最新的值，减去上一个时间点（分钟、十五分钟、半小时、1小时）的值。得到差值之后，将数据保存到数据库。

作用：解放了程序员的双手。要分析指标的时候，只需要产品经理在网页配置一下即可。