

第1章 Storm是什么

1.1 背景-流式计算与 storm

2011 年在海量数据处理领域，Hadoop 是人们津津乐道的技术，Hadoop 不仅可以用来存储海量数据，还可以用来计算海量数据。因为其高吞吐、高可靠等特点，很多互联网公司都已经使用 Hadoop 来构建数据仓库，高频使用并促进了 Hadoop 生态圈的各项技术的发展。一般来讲，根据业务需求，数据的处理可以分为离线处理和实时处理，在离线处理方面 Hadoop 提供了很好的解决方案，但是针对海量数据的实时处理却一直没有比较好的解决方案。

就在人们翘首以待的时间节点，Storm 横空出世，与生俱来的分布式、高可靠、高吞吐的特性，横扫市面上的一些流式计算框架，渐渐的成为了流式计算的首选框架。

如果庞麦郎在的话，他一定会说，这就是我要的滑板鞋！

The screenshot shows the GitHub search interface with 'storm' entered in the search bar. The results show 7,753 repository results. On the left, there are filters for Repositories (7,753), Code (4,230,406), Issues (17,969), and Users (2,272). Below these are language filters: Java (2,572), JavaScript (644), Python (331), HTML (331), Swift (226), and PHP (208). The main results list 'nathanmarz/storm' as the top result, described as 'Distributed and fault-tolerant realtime computation: stream processing, continuous computation, distributed RPC, and more', with 8,829 stars and 1,773 forks. Below it is 'apache/storm', described as 'Mirror of Apache Storm', with 3,402 stars and 2,530 forks. A bar chart at the bottom shows the commit activity for these repositories over time.

在 2013 年，阿里巴巴开源了基于 storm 的设计思路使用 java 重现编写的流式计算框架 jstorm。那 jstorm 是什么呢？

在 jstorm 早期的介绍中，一般会出现下面的语句：JStorm 比 Storm 更稳定，更强大，更快，Storm 上跑的程序，一行代码不变可以运行在 JStorm 上。

在最新的介绍中，jstorm 的团队是这样介绍的：JStorm 是一个类似 Hadoop MapReduce 的系统，用户按照指定的接口实现一个任务，然后将这个任务递交给 JStorm 系统，Jstorm 将这个任务跑起来，并且按 7 * 24 小时运行起来，一旦中间一个 Worker 发生意外故障，调度器立即分配一个新的 Worker 替换这个失效的 Worker。

因此，从应用的角度，JStorm 应用是一种遵守某种编程规范的分布式应用。

从系统角度，JStorm 一套类似 MapReduce 的调度系统。从数据的角度，是一套基于流水线的消息处理机制。实时计算现在是大数据领域中最火爆的一个方向，因为人们对数据的要求越来越高，实时性要求也越来越快，传统的 Hadoop MapReduce，逐渐满足不了需求，因此在这个领域需求不断。现在，Jstorm 在淘宝海量的数据和大量的业务场景的锤炼下，从开始的追随者，使用者慢慢的演变成了流式计算技术的领导者。当下，还有很多企业并不知道 jstorm，他们的生产环境依然是 storm，

并且 storm 也在不断更新，在笔者成文的时间点上，storm 发布了 1.0 的 beta 版。

鉴于大多数企业的生产环境还在使用 storm，我们学习的目标还是切换到 Apache 基金会的 storm 上来。

1.2 背景-Storm 是为了解决什么样的问题

伴随着信息科技日新月异的发展，信息呈现出爆发式的膨胀，人们获取信息的途径也更加多样、更加便捷，同时对于信息的时效性要求也越来越高。

举个搜索场景中的例子，当一个卖家发布了一条宝贝信息时，**他希望的当然是这个宝贝马上就可以被卖家搜索出来、点击、购买啦**，相反，如果这个宝贝要等到第二天或者更久才可以被搜出来，估计这个大哥就要骂娘了。

再举一个推荐的例子，**如果用户昨天在淘宝上买了一双袜子，今天想买一副泳镜去游泳**，但是却发现系统在不遗余力地给他推荐袜子、鞋子，根本对他今天寻找泳镜的行为视而不见，估计这哥们心里就会想推荐你妹呀。其实稍微了解点背景知识的码农们都知道，这是因为后台系统做的是每天一次的全量处理，而且大多是在夜深人静之时做的，那么你今天白天做的事情当然要明天才能反映出来啦。

1.3 背景-实现实时计算系统需要解决那些问题

如果让我们自己设计一个实时计算系统，我们要解决哪些问题。

- (1) 低延迟：都说了是实时计算系统了，延迟是一定要低的。
- (2) 高性能：性能不高就是浪费机器，浪费机器是要受批评的哦。
- (3) 分布式：系统都是为应用场景而生的，如果你的应用场景、你的数据和计算单机就能搞定，那么不用考虑这些复杂的问题了。我们所说的是单机搞不定的情况。
- (4) 可扩展：伴随着业务的发展，我们的数据量、计算量可能会越来越大，所以希望这个系统是可扩展的。
- (5) 容错：这是分布式系统中通用问题。一个节点挂了不能影响我的应用。
- (6) 通信：设计的系统需要应用程序开发人员考虑各个处理组件的分布、消息的传递吗？如果是，发人员可能会用不好，也不会想去用。
- (7) **消息不丢失：用户发布的一个宝贝消息不能在实时处理的时候给丢了**，对吧？

1.4 离线计算是什么

离线计算：批量获取数据、批量传输数据、**周期性**批量计算数据、数据展示

代表技术：Sqoop 批量导入数据、HDFS 批量存储数据、MapReduce 批量计算数据、Hive 批量计算数据、***任务调度

日常业务：

- 1, hivesql
- 2、调度平台

- 3、Hadoop 集群运维
- 4、数据清洗（脚本语言）
- 5、元数据管理
- 6、数据稽查
- 7、数据仓库模型架构

1.5 流式计算是什么

流式计算：数据实时产生、数据实时传输、数据实时计算、实时展示

代表技术：Flume 实时获取数据、Kafka/**metaq** 实时数据存储、**Storm/JStorm** 实时数据计算、Redis 实时**结果**缓存、持久化存储(mysql)。

一句话总结：将源源不断产生的数据实时收集并实时计算，尽可能快的得到计算结果，用来支持决策。

1.6 离线计算与实时计算的区别

最大的区别：实时收集、实时计算、实时展示

离线计算，一次计算很多条数据

实时计算，数据被一条一条的计算

1.7 Storm 是什么？

Storm 用**来实时处理数据**，特点：低延迟、高可用、分布式、可扩展、**数据不丢失**。提供简单容易理解的接口，便于开发。

Spout Bolt

数据输入 数据计算 数据输出 数据计算 数据输出

Spout Bolt 1.... Bolt N BoltN+1. BoltN....

1.8 Storm 的应用场景

Storm 处理数据的方式是基于消息的流水线处理，因此特别适合无状态计算，也就是计算单元的依赖的数据全部在接受的消息中可以找到，并且最好一个数据流不依赖另外一个数据流。

因此，常常用于

- 日志分析，从海量日志中分析出特定的数据，并将分析的结果存入外部存储器用来辅助决策。
- 管道系统，将一个数据从一个系统传输到另外一个系统，比如将数据库同步到 Hadoop
- 消息转化器，将接受到的消息按照某种格式进行转化，存储到另外一个系统如消息中间件
- 统计分析器，从日志或消息中，提炼出某个字段，然后做 count 或 sum 计算，最后将统计值存入外部存储器。中间处理过程可能更复杂。

1.8.1 案例：一淘-实时分析系统

一淘-实时分析系统：实时分析用户的属性，并反馈给搜索引擎。最初，用户属性分析是通过每天在云梯上定时运行的 MR job 来完成的。为了满足实时性的要求，希望能够实时分析用户的行为日志，将最新的用户属性反馈给搜索引擎，能够为用户展现最贴近其当前需求的结果。

1.8.2 案例：携程-网站性能监控

携程-网站性能监控：实时分析系统监控携程网的网站性能。利用 HTML5 获得可用的指标，并记录日志。Storm 集群实时分析日志和入库。使用 DRPC 聚合成报表，通过历史数据对比等判断规则，触发预警事件。

1.8.3 案例：游戏实时运营

一个游戏新版本上线，有一个实时分析系统，收集游戏中的数据，运营或者开发者可以在上线后几秒钟得到持续不断更新的游戏监控报告和分析结果，然后马上针对游戏的参数 和平衡性进行调整。这样就能够大大缩短游戏迭代周期，加强游戏的生命力。

1.8.4 案例：实时计算在腾讯的运用

实时计算在腾讯的运用：精准推荐（广点通广告推荐、新闻推荐、视频推荐、游戏道具推荐）；实时分析（微信运营数据门户、效果统计、订单画像分析）；实时监控（实时监控平台、游戏内接口调用）

1.8.5 案例：实时计算在阿里的运用

为了更加精准投放广告，阿里妈妈后台计算引擎需要维护每个用户的兴趣点（理想状态是，你对什么感兴趣，就向你投放哪类广告）。用户兴趣主要基于用户的历史行为、用户的实时查询、用户的实时点击、用户的地理信息而得，其中实时查询、实时点击等用户行为都是实时数据。考虑到系统的实时性，阿里妈妈使用 Storm 维护用户兴趣数据，并在此基础上进行受众定向的广告投放。

1.9 Storm 在互联网公司

本节内容来自《Storm 技术内幕与大数据实践》，并得到作者授权使用。



1. 新浪的实时分析平台

新浪实时分析平台的计算引擎是 Storm，整个实时计算平台包括可视化的任务提交 Portal 界面、对实时计算任务的管理监控平台以及核心处理实时计算平台。

Storm 作为核心处理，待处理数据来源为 Kafka。对于实时性要求比较高的应用、数据会直接发送到 Kafka，然后由 Storm 中的应用进行实时分析处理；而对实时性要求不太高的应用，则由 Scribe 收集数据，然后转发到 Kafka 中，再由 Storm 进行处理。

任务提交到 Portal 之前，作业的提交者需要确定数据源、数据的每个处理逻辑，同时确定处理完成后数据的存储、获取和展示方式。在任务提交后，可以完成对任务的管理：编辑、停止、暂停和恢复等。

2. 腾讯的实时计算平台

腾讯的实时计算平台 Tencent Real-time Computing 主要由两部分组成：分布式 K/V 存储引擎 TDEngine 和支持数据流计算的 TDProcess。TDProcess 是基于 Storm 的计算引擎，提供了通用的计算模型，如 Sum、Count、PV/UV 计算和 TopK 统计等。整个平台修复了运行中发现的 Storm 的问题，同时引入 YARN 进行资源管理。

据称，整个计算平台每天承载了超过 1000 亿数据量的计算，支持广点通、微信、视频、易迅、秒级监控、电商和互娱等业务上百个实时统计的需求。

3. 奇虎 360 实时平台

奇虎 360 从 2012 年开始引入 Storm，Storm 主要应用场景包括云盘缩略图、日志实时分析、搜索热词推荐、在线验证码识别、实时网络入侵检测等包括网页、图片、安全等应用。在部署中，使用了 CGroup 进行资源隔离，并向 Storm 提交了很多补丁，如 log UI (<https://github.com/nathanmarz/storm/pull/598>) 等。在部署上，Storm 集群复用了其他机器的空闲资源（Storm 部署在其他服务的服务器上，每台机器贡献 1~2 核处理器、1~2 GB 内存），整个规模达到 60 多个集群，15 000 多台物理机，服务于 170 多个业务。每天处理数据量约几百 TB、几百亿条记录。

4. 京东的实时平台

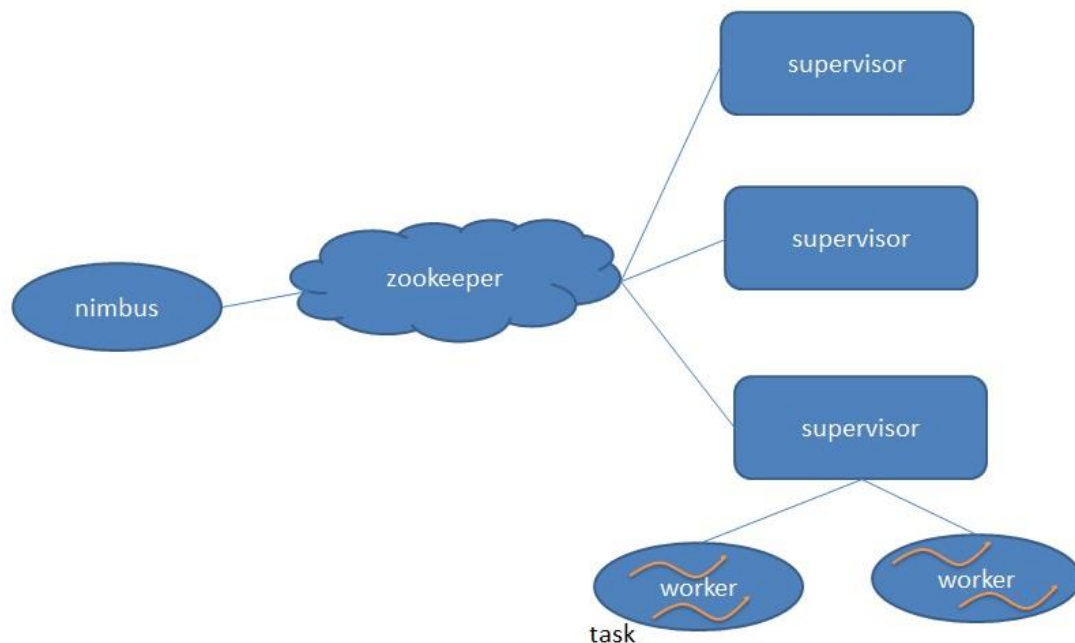
京东的实时平台基于 LinkedIn 开源的 Samza，整个 Samza 包括流处理层 Kafka，执行层 YARN 和处理层 Samza API。一个流式处理由一个或多个作业组成，作业之间的信息交互借助 Kafka 实现，一个作业在运行状态表现为一个或者多个 Task，整个处理过程实际上是在 Task 中完成的。在 Samza 中，Kafka 主要的角色是消息的缓冲、作业交互信息的存储，同一个业务流程中使用 YARN 进行任务调度。在其整个架构中，引入了 Redis 作为数据处理结果的存储，并通过 Comet 技术将实时分析的数据推送到前台展示，整个业务主要应用于京东大家电的订单处理，实时分析统计出待定区域中各个状态的订单量（包括待定位、待派工、待拣货、待发货、待配送、待妥投等）。

5. 百度的实时系统

相对而言，百度在实时系统上开展的比较早，在其流计算平台 DStream 开发时业界尚未有类似的开源系统。截至 2014 年，从公开的资料可以发现，DStream 平台的集群规模已超千台，单集群最大处理数据量超过 50 TB/天，集群峰值 QPS 193W/S，系统稳定性、计算能力已完全满足海量数据时效性处理需求。另一个平台 TM 则保证数据不重不丢，主要用于报表生成系统、计费流计算等。

第2章 Storm核心组件（重要）

2.1 Storm 架构图



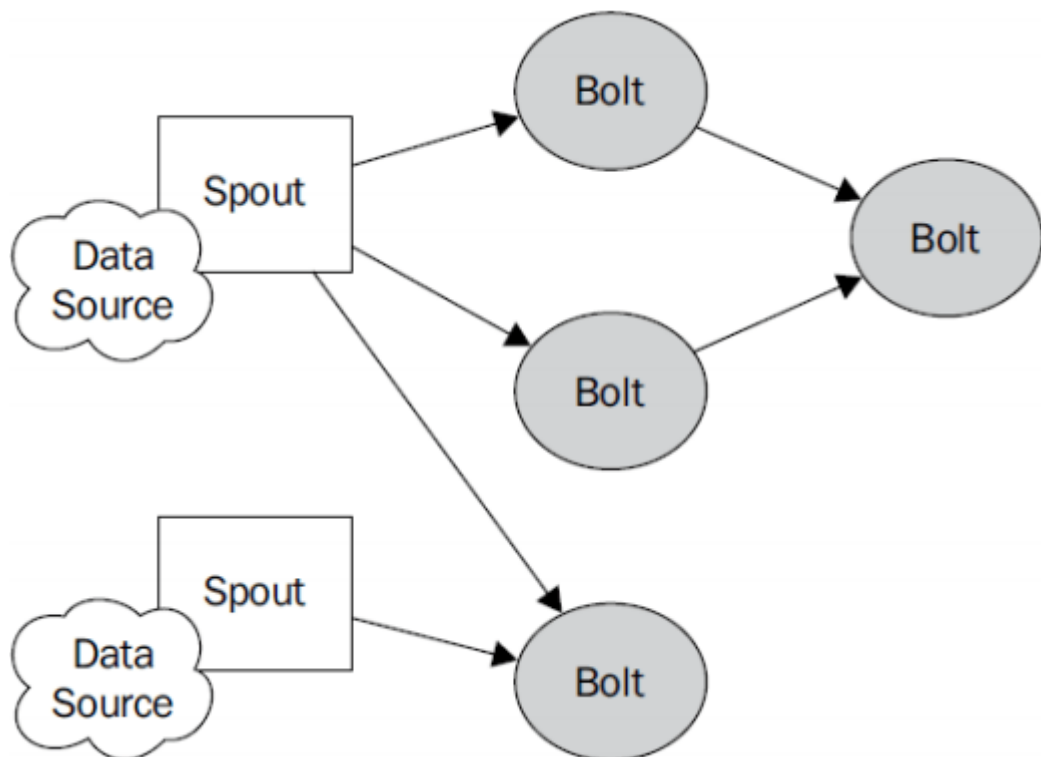
Nimbus：负责资源分配和任务调度。

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。

2.2 Storm 编程模型



2.2.1 基本介绍

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。

2.2.2 Stream grouping

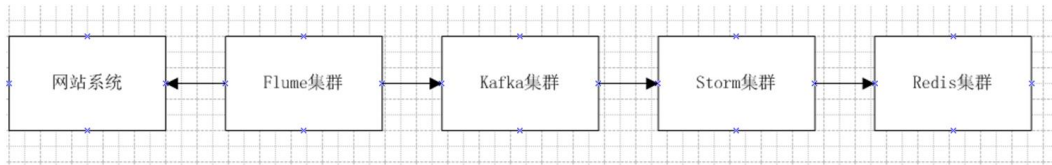
Stream grouping：即消息的 partition 方法。

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。

2.3 流式计算整体结构



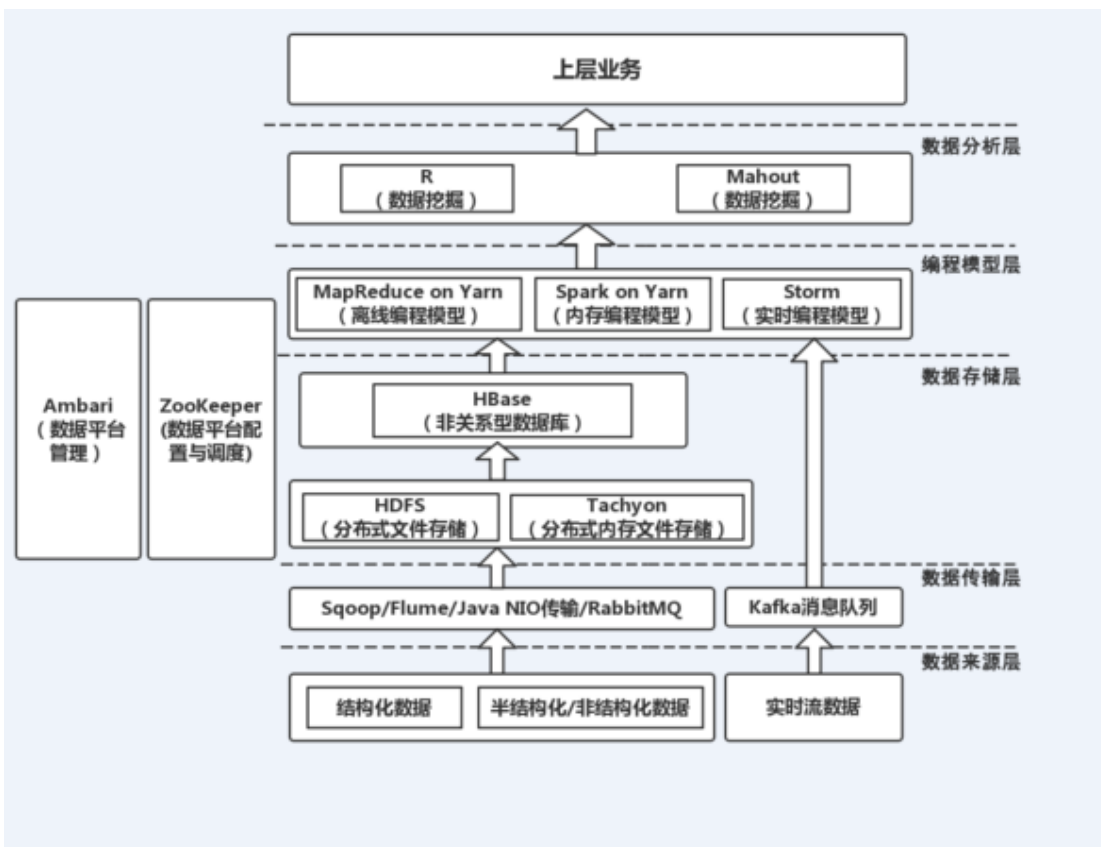
flume 用来获取数据

Kafka 用来临时保存数据

Strom 用来计算数据

Redis 是个内存数据库，用来保存数据

2.4 Storm 在综合项目中



第3章 WordCount案例分析（重要）

3.1 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>
```

3.2 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.createTo
pology());
} else {
    LocalCluster localCluster = new LocalCluster();
    localCluster.submitTopology("wordcount", config,
topologyBuilder.createTopology());
}
}
```

3.3 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"));
}
}
```

3.4 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 value 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"));  
}  
}
```

3.5 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"));
    }
}

```

3.6 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>

```

3.7 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>
```

3.8 Component 生命周期

3.8.1 Spout 生命周期

```
m 📄 close(): void ↑ISpout  
m 📄 activate(): void ↑ISpout  
m 📄 deactivate(): void ↑ISpout  
m 📄 ack(Object): void ↑ISpout  
m 📄 fail(Object): void ↑ISpout
```

3.8.2 Bolt 生命周期

```
m 📄 prepare(Map, TopologyContext): void ↑BaseBasicBolt  
m 📄 execute(Tuple, BasicOutputCollector): void ↑IBasicBolt  
m 📄 declareOutputFields(OutputFieldsDeclarer): void ↑ICom
```

3.8.3 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》

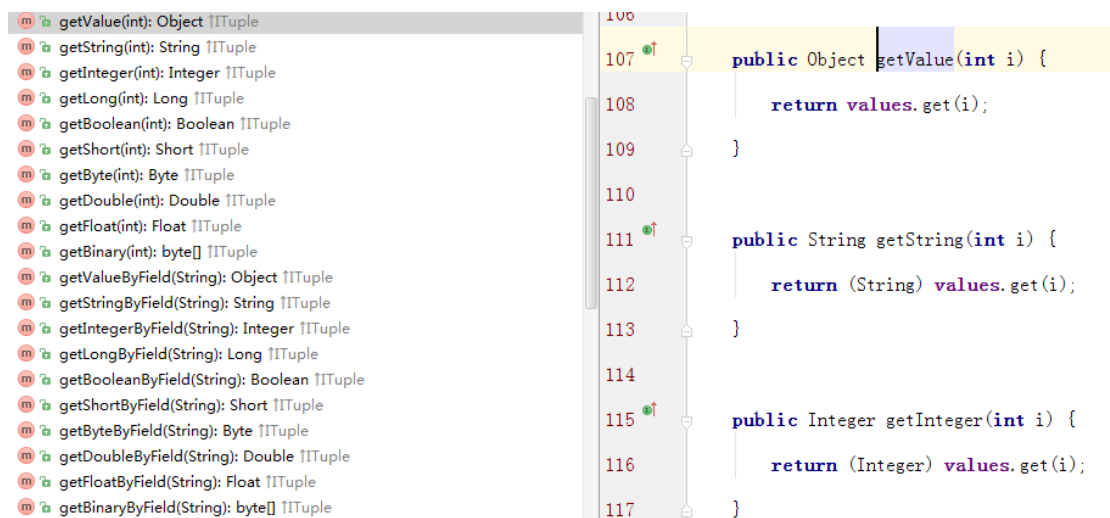
3.9 StreamGrouping

MKgrouper

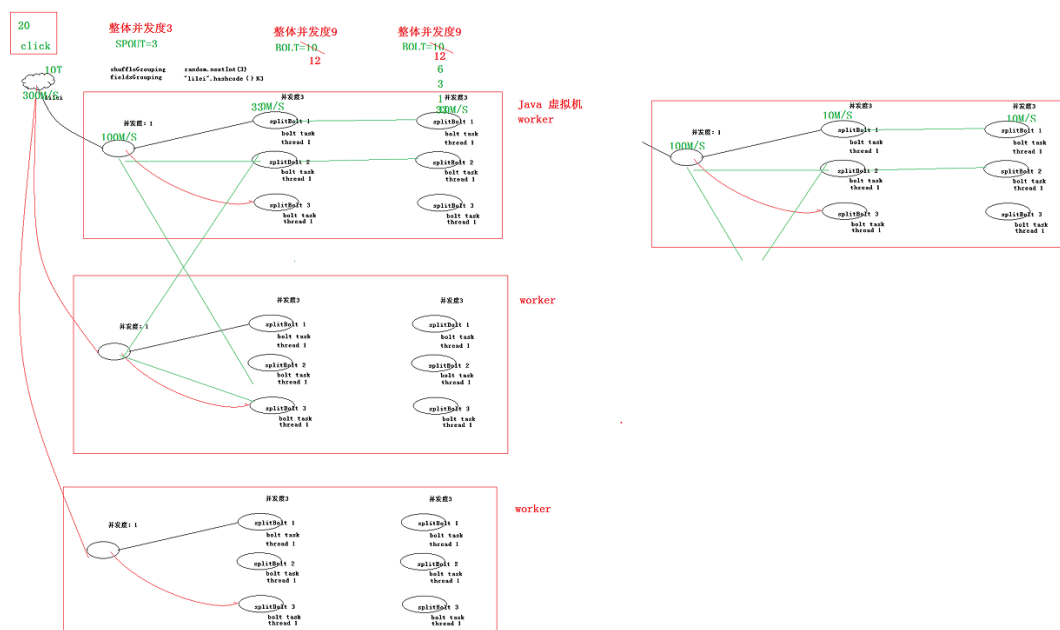
```
public Map<List<Integer>, List<MsgInfo>> grouperBatch(List<MsgInfo>  
batch) {  
    Map<List<Integer>, List<MsgInfo>> ret = new HashMap<List<Integer>,  
List<MsgInfo>>();  
    //optimize fieldGrouping & customGrouping  
    if (GrouperType.local_or_shuffle.equals(grouptype)) {  
        ret.put(local_shuffer_grouper.grouper(null), 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;
}
```

3.10 Tuple 是什么



3.11 并行度是什么



第4章 Storm 集群安装部署（必备）

4.1 集群规划

参见《附 2、Storm 集群部署详细手册.docx》

4.2 基础环境安装

参见《附 2、Storm 集群部署详细手册.docx》

4.3 Storm 集群安装

参见《附 2、Storm 集群部署详细手册.docx》

参见《Apache Storm 集群搭建命令.html》

4.4 常用命令

有许多简单且有用的命令可以用来管理拓扑，它们可以提交、杀死、禁用、再平衡拓扑。

- 提交任务命令格式：storm jar 【jar 路径】 【拓扑包名.拓扑类名】 【拓扑名称】
bin/storm jar examples/storm-starter/storm-starter-topologies-0.10.0.jar
storm.starter.WordCountTopology wordcount
- 杀死任务命令格式：storm kill 【拓扑名称】 -w 10（执行 kill 命令时可以通过-w [等待秒数]
指定拓扑停用以后的等待时间）
storm kill topology-name -w 10
- 停用任务命令格式：storm deactivate 【拓扑名称】
storm deactivate topology-name

我们能够挂起或停用运行中的拓扑。当停用拓扑时，所有已分发的元组都会得到处理，但是 spouts 的 nextTuple 方法不会被调用。销毁一个拓扑，可以使用 kill 命令。它会以一种安全的方式销毁一个拓扑，首先停用拓扑，在等待拓扑消息的时间段内允许拓扑完成当前的数据流。

- 启用任务命令格式：storm activate 【拓扑名称】
storm activate topology-name
- 重新部署任务命令格式：storm rebalance 【拓扑名称】
storm rebalance topology-name

再平衡使你重分配集群任务。这是个很强大的命令。比如，你向一个运行中的集群增加了节点。再平衡命令将会停用拓扑，然后在相应超时时间之后重分配 worker，并重启拓扑。

第5章 实时交易数据统计

5.1 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。





5.2 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

5.3 数据准备

淘宝网 -- 秒杀KR迷你家用静音空气加湿器办公室香薰增湿机带独立夜灯家用

卖家昵称：百年尚品

商品名称：秒杀KR迷你家用静音空气加湿器办公室香薰增湿机带独立夜灯家用

交易金额：29.61元

购买时间：2015年11月03日 18:27:11

收货地址：回龙观建材城西路金燕龙办公楼，0000000（邮编）毛祥溢（收）15652

交易类型：支付宝担保交易

交易号：2015110321001001660288794973

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

订单信息

收货地址：北京 北京市海淀区四季青
电话：13810406418 000000

订单编号：1250879274589179

支付宝交易号：20151103210010016602721584
成交时间：2015-11-03 00:09:38
付款时间：2015-11-03 00:09:48
发货时间：2015-11-03 11:29:51

商家：家比家居专营店
真实姓名：泉州市鲤城区家比贸易有限公司
城市：泉州市
联系电话：18160911202

购买用户
用户电话

订单编号
订单时间
支付编号
支付时间
商品编号
商家简称
商家电话

```
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 BigDecimal productPrice;//商品价格
private BigDecimal promotionPrice;//促销价格
private String shopId;//店铺编号
private String shopName;//店铺名称
private String shopMobile;
private BigDecimal payPri;
private int num;//订单数量
```

注:实际业务更复杂,比如父子订单,一个订单多个商品,一个支付订单多个

商品名称、商品价格、商品数量、优惠价格、支付价格	数量	优惠(元)	状态	运费(元)
包裹1 优惠快递 运单号:900119621692				
相框世家简易衣帽架落地 衣服挂衣架 落地... 颜色分类: 908E粉色	1	159.00	售后无忧省 136.20	待确认收货 9天18时
				0.00 (快递)

5.4 业务口径

秒杀 优惠券 闪购 拍卖 京东服饰 京东超市 生鲜 全球购 京东金融

玩3C > 手机通讯 > 手机配件 > 摄影摄像 > 数码配件 > 影音娱乐 > 智能设备 > 电子教育 >

手机 > 对讲机 > 以旧换新 > 手机维修

合约机 > 选号码 > 国话宽带 > 办套餐 > 充话费/流量 > 中国电信 > 中国移动 > 中国联通 > 京东通信 > 170选号

手机壳 > 贴膜 > 手机存储卡 > 数据线 > 充电器 > 手机耳机 > 创意配件 > 手机饰品 > 手机电池 > 苹果周边

移动电源 > 蓝牙耳机 > 手机支架 > 车载配件 > 拍照配件

数码相机 > 单反/微单相机 > 单反相机 > 拍立得 > 运动相机 > 摄像机 > 镜头 > 户外器材 > 影棚器材 > 冲印服务

数码相机

存储卡 > 三脚架/云台 > 相机包 > 滤镜 > 闪光灯/手柄 > 相机清洁/贴膜 > 机身附件 > 镜头附件 > 读卡器 > 支架

电池/充电器

耳机/耳麦 > 音箱/音响 > 智能音箱 > 便携/无线音箱 > 收音机 > 麦克风 > MP3/MP4 > 专业音频

智能手环 > 智能手表 > 智能眼镜 > 智能机器人 > 运动跟踪器 > 健康监测 > 智能配饰 > 智能家居 > 体感车 > 无人机

其他配件

学生平板 > 点读机/笔 > 早教益智 > 录音笔 > 电子书 > 电子词典 > 复读机

- 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
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
```

5.5 数据展示

读取 redis 中的数据，每秒进行展示，打印在控制台。

5.6 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

5.7 代码开发

5.7.1 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>
```

5.7.2 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 允许您执行更大的持久性保
         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();
    }
}
}
```

5.7.3 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());  
    }  
}  
}
```

5.7.4 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"));  
    }  
}
```

5.7.5 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来进行来。
    */
    @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);

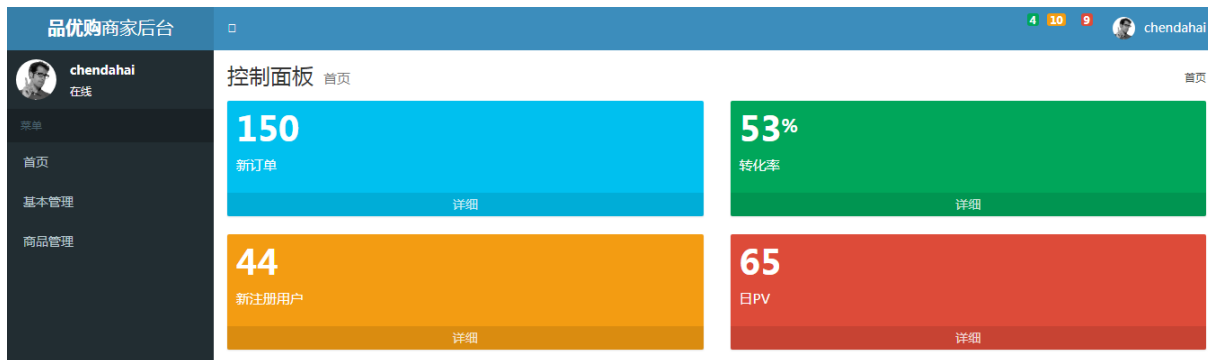
        jedis.incrBy("kanban:shop:"+shopId+":orderPrice", value.getPayPrice());
        jedis.incrBy("kanban:shop:"+shopId+":orderuser", 1);

        //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) {
    }
}
```

5.7.6 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();
            }
        }
    }
}
```

5.7.7 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>
```

5.7.8 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 + '\'' +
        ", createTime='" + createTime +
        ", 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);
}
```

第6章 Storm源码下载及目录熟悉

6.1 在 Storm 官方网站上寻找源码地址

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

Downloads for Storm are below. Instructions for how to set up a Storm cluster can be found [here](#).

Source Code

Current source code is hosted on GitHub, [apache/storm](#)

6.2 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>

6.3 拷贝 storm 源码地址

在网页右侧，拷贝 storm 源码地址

Subversion checkout URL

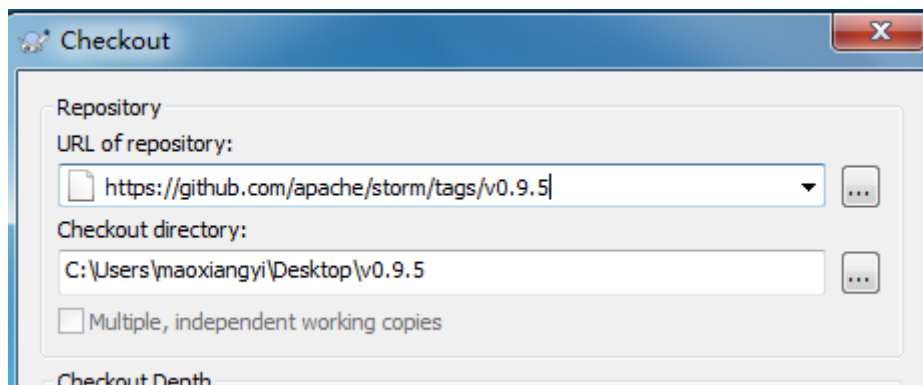
<https://github.cc>

You can clone with [HTTPS](#),
[SSH](#), or Subversion. ⓘ

Clone in Desktop

Download ZIP

6.4 使用 Subversion 客户端下载




















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

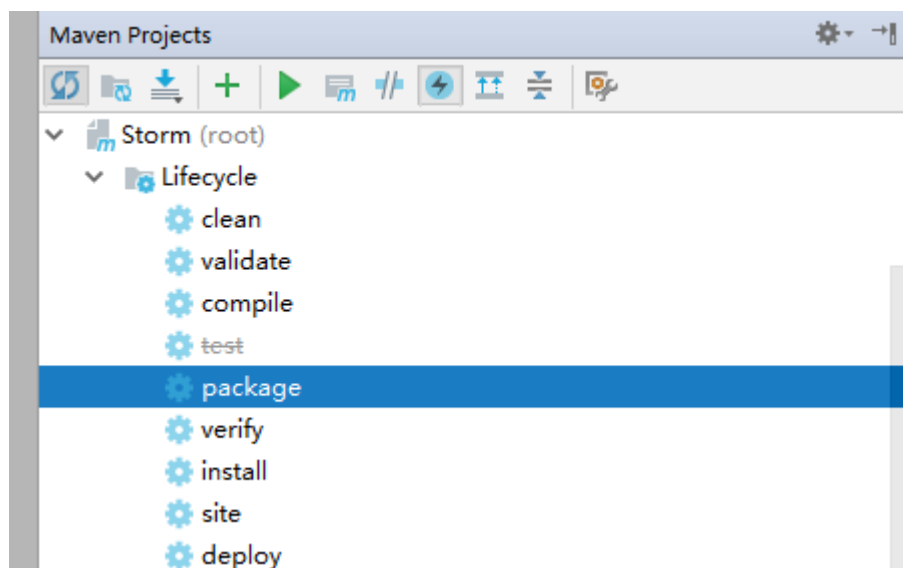
6.5 Storm 源码目录分析

bin	2015/10/29 15:20	文件夹	
conf	2015/10/29 15:20	文件夹	
dev-tools	2015/10/29 15:20	文件夹	
examples	2015/10/29 15:20	文件夹	各种特性的Demo
external	2015/10/29 15:20	文件夹	扩展插件, kafka, hbase, hdfs
logback	2015/10/29 15:20	文件夹	
storm-buildtools	2015/10/29 15:20	文件夹	
storm-core	2015/10/29 17:24	文件夹	核心代码
storm-dist	2015/10/29 15:20	文件夹	

扩展包中的三个项目，使 storm 能与 hbase、hdfs、kafka 交互

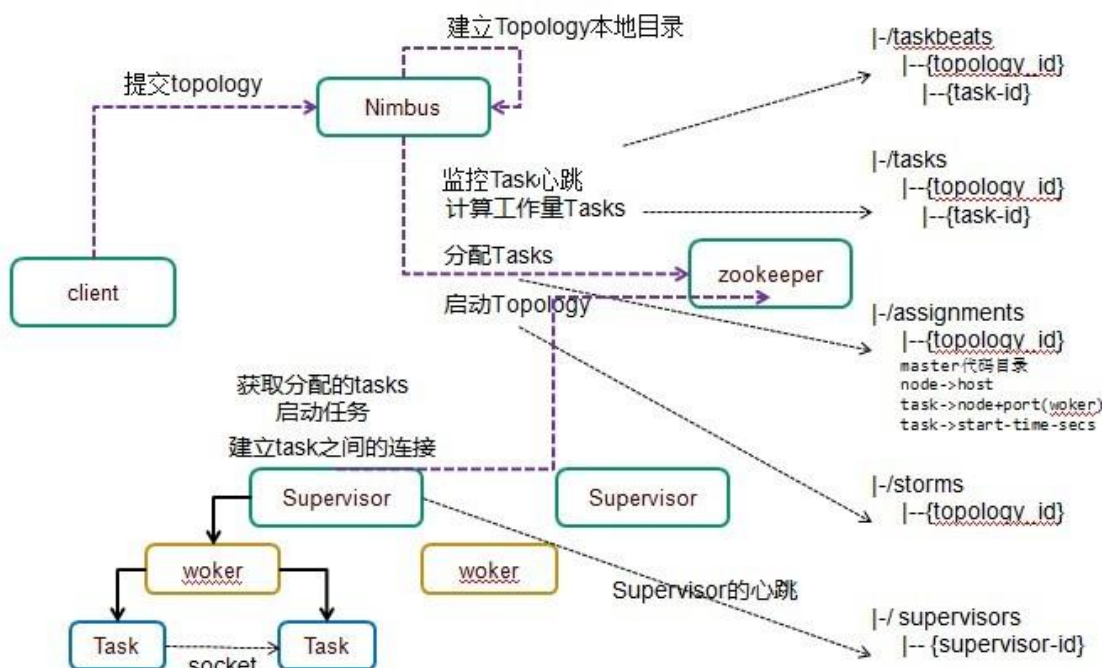
 flux-examples	2017/12/19 23:15	文件夹
 storm-elasticsearch-examples	2017/12/19 23:15	文件夹
 storm-hbase-examples	2017/12/19 23:15	文件夹
 storm-hdfs-examples	2017/12/19 23:15	文件夹
 storm-hive-examples	2017/12/19 23:15	文件夹
 storm-jdbc-examples	2017/12/19 23:15	文件夹
 storm-jms-examples	2017/12/19 23:15	文件夹
 storm-kafka-client-examples	2017/12/19 23:15	文件夹
 storm-kafka-examples	2017/12/19 23:15	文件夹
 storm-mongodb-examples	2017/12/19 23:15	文件夹
 storm-mqtt-examples	2017/12/19 23:15	文件夹
 storm-opentsdb-examples	2017/12/19 23:15	文件夹
 storm-perf	2017/12/19 23:15	文件夹
 storm-pmml-examples	2017/12/19 23:15	文件夹
 storm-redis-examples	2017/12/19 23:15	文件夹
 storm-solr-examples	2017/12/19 23:15	文件夹
 storm-starter	2017/12/19 23:15	文件夹

6.6 Storm 源码编译



第7章 Storm原理

7.1 Storm 任务提交的过程



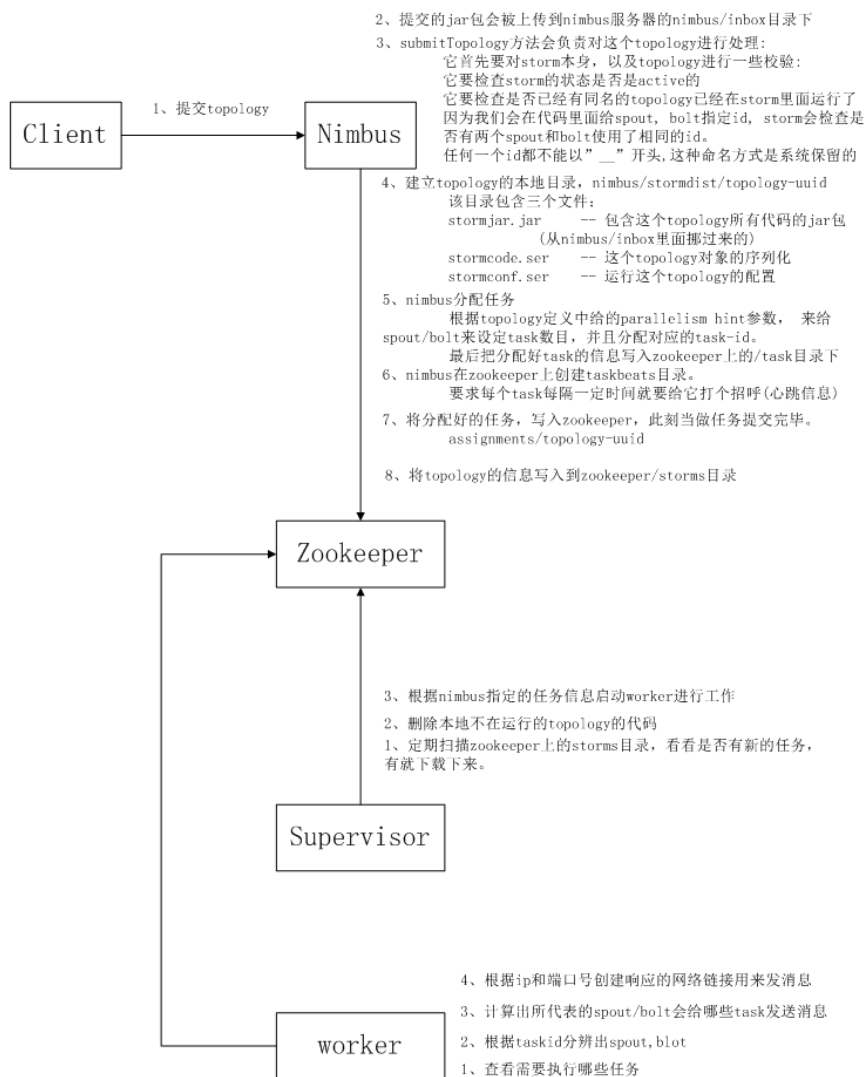
```
TopologyMetricsRunnable.TaskStartEvent[oldAssignment=<null>,newAssignment=Assignment[
masterCodeDir=C:\Users\MAOXIA~1\AppData\Local\Temp\e73862a8-f7e7-41f3-883d-af49461
8bc9f\nimbus\stormdist\double11-1-1458909887,nodeHost={61ce10a7-1e78-4c47-9fb3-c21f43a3
31ba=192.168.1.106},taskStartTimeSecs={ 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,task
s=[1, 2, 3, 4, 5, 6, 7, 8],jvm=<null>,nodeId=61ce10a7-1e78-4c47-9fb3-c21f43a331ba,port=6900]],timeStamp=145890
9910633,type=Assign],task2Component=<null>,clusterName=<null>,topologyId=double11-1-145
8909887,timestamp=0]
```

```

taskEvent = {TopologyMetricsRunnable$TaskStartEvent@2395} "TopologyMetricsRunnable.TaskStartEvent[oldAssignment=null,newAssignment=Assignment[masterCodeDir=C:\Users\MAOXIA~1\AppData\Local\Temp\... View
  oldAssignment = null
  newAssignment = {Assignment@2324} "Assignment[masterCodeDir=C:\Users\MAOXIA~1\AppData\Local\Temp\... View
    masterCodeDir = {String@2335} "C:\Users\MAOXIA~1\AppData\Local\Temp\... View
    nodeHost = {HashMap@2336} size = 1
      0 = {HashMap$Node@2431} "61ce10a7-1e78-4c47-9fb3-c21f43a331ba" -> "192.168.1.106"
    taskStartTimeSecs = {TreeMap@2337} size = 8
    workers = {HashSet@2338} size = 1
    timestamp = 1458909910633
    type = {Assignment$AssignmentType@2339} "Assign"
    task2Component = null
    clusterName = {String@2460} "default"
    topologyId = {String@2426} "double11-1-1458909887"
    timestamp = 1458910163663
taskInfoMap = {TreeMap@3946} size = 8
  0 = {TreeMap$Entry@3969} "1" -> "TaskInfo[componentId=saveResult2Redis,componentType=bolt]"
  1 = {TreeMap$Entry@3970} "2" -> "TaskInfo[componentId=saveResult2Redis,componentType=bolt]"
  2 = {TreeMap$Entry@3971} "3" -> "TaskInfo[componentId=processIndex,componentType=bolt]"
  3 = {TreeMap$Entry@3972} "4" -> "TaskInfo[componentId=processIndex,componentType=bolt]"
  4 = {TreeMap$Entry@3973} "5" -> "TaskInfo[componentId=__acker,componentType=bolt]"
  5 = {TreeMap$Entry@3974} "6" -> "TaskInfo[componentId=readPaymentInfo,componentType=spout]"
  6 = {TreeMap$Entry@3975} "7" -> "TaskInfo[componentId=readPaymentInfo,componentType=spout]"
  7 = {TreeMap$Entry@3976} "8" -> "TaskInfo[componentId=readPaymentInfo,componentType=spout]"

```

Storm任务提交过程



7.2 Storm 组件本地目录树

```
/{storm-local-dir}
|
|-- /nimbus
|   |
|   |-- /inbox                                -- 从nimbus客户端上传的jar包会在这个目录里面
|   |   |
|   |   |-- /stormjar-{uuid}.jar            -- 上传的jar包其中{uuid}表示生成的一个uuid
|   |
|   |-- /stormdist
|       |
|       |-- /{topology-id}
|           |
|           |-- /stormjar.jar                -- 包含这个topology所有代码的jar包
|           |                                (从nimbus/inbox里面挪过来的)
|           |-- /stormcode.ser              -- 这个topology对象的序列化
|           |-- /stormconf.ser              -- 运行这个topology的配置
|
|-- /supervisor
|   |
|   |-- /stormdist
|       |
|       |-- /{topology-id}
|           |
|           |-- /resources                  -- 这里保存的是topology的jar包里面的resources目录下面的所有文件
|           |-- /stormjar.jar              -- 从nimbus机器上下载来的topology的jar包
|           |-- /stormcode.ser             -- 从nimbus机器上下载来的这个topology对象的序列化形式
|           |-- /stormconf.ser             -- 从nimbus机器上下载来的运行这个topology的配置
|
|   |-- /localstate                          -- supervisor的localstate
|
|   |-- /tmp                                -- 临时目录，从Nimbus上下载的文件会先存在这个目录里面，
|       |                                然后做一些简单处理再copy到stormdist/{topology-id}
|       |
|       |-- /{uuid}
|           |
|           |-- /stormjar.jar              -- 从Nimbus上面download下来的工作jar包
|
|-- /workers
|   |
|   |-- /{worker-id}
|       |
|       |-- /pids                           -- 一个worker可能会起多个子进程所以可能会有多个pid
|           |
|           |-- /{pid}                     -- 运行这个worker的JVM的pid
|
|       |-- /heartbeats                     -- 这个supervisor机器上的worker的心跳信息
|           |
|           |-- /{worker-id}              -- 这里面存的是一个worker的心跳：
|                                           主要包括心跳时间和worker的id
```

7.3 Storm zookeeper 目录树

```
/{storm-zk-root}          -- storm在zookeeper上的根目录
|
|-/assignments            -- topology的任务分配信息
| |
| |-//{topology-id}        -- 这个下面保存的是每个topology的assignments信息包括：
| |                        -- 对应的nimbus上的代码目录,所有task的启动时间,每个task与机器、端口的映射
|
|-/tasks                  -- 所有的task
| |
| |-//{topology-id}        -- 这个目录下面id为{topology-id}的topology所对应的所有的task-id
| | |
| | |-//{task-id}          -- 这个文件里面保存的是这个task对应的component-id:
| | |                        -- 可能是spout-id或者bolt-id
|
|-/storms                 -- 这个目录保存所有正在运行的topology的id
| |
| |-//{topology-id}        -- 这个文件保存这个topology的一些信息，
| |                        -- 包括topology的名字，topology开始运行的时间以及这个topology的状态
| |                        -- (具体看StormBase类)
|
|-/supervisors            -- 这个目录保存所有的supervisor
| |                        -- 的心跳信息
| |
| |-//{supervisor-id}      -- 这个文件保存的是supervisor的心跳信息包括：
| |                        -- 心跳时间，主机名，这个supervisor上worker的端口号运行时间
| |                        -- (具体看SupervisorInfo类)
|
|-/taskbeats              -- 所有task的心跳
| |
| |-//{topology-id}        -- 这个目录保存这个topology的所有的task的心跳信息
| | |
| | |-//{task-id}          -- task的心跳信息：
| | |                        -- 包括心跳的时间，task运行时间以及一些统计信息
|
|-/taskerrors             -- 所有task所产生的error信息
| |
| |-//{topology-id}        -- 这个目录保存这个topology下面每个task的出错信息
| | |
| | |-//{task-id}          -- 这个task的出错信息
```

7.4 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--->ack---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

7.5 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);
    }
}
```

第8章 Storm通信机制（了解）

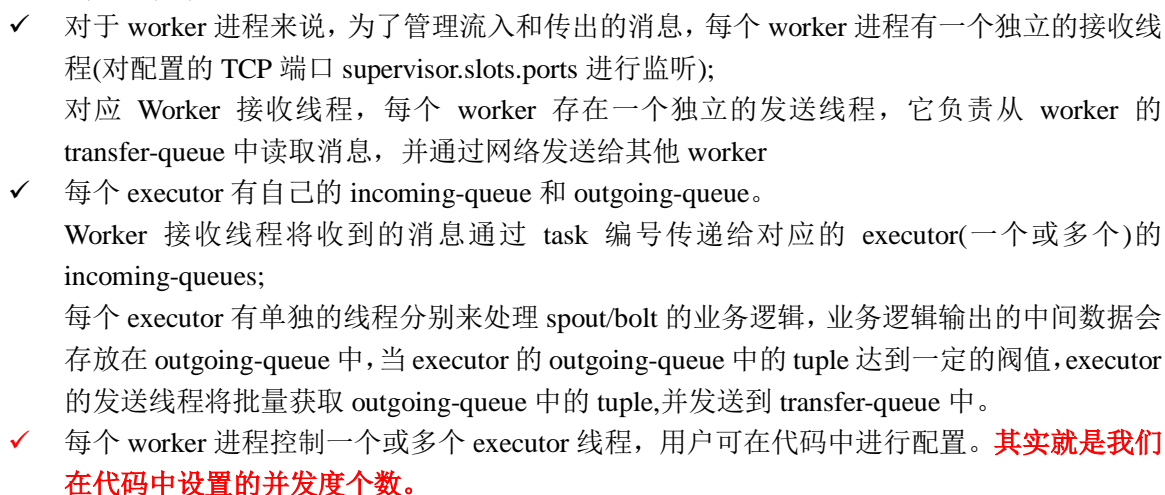
Worker 间的通信经常需要通过网络跨节点进行，Storm 使用 ZeroMQ 或 Netty(0.9 以后默认使用) 作为进程间通信的消息框架。

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

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

8.1 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



```

graph LR
    SC((Socket Connection)) --> RT[Receive Thread]
    RT -- Transfer Queue --> RQM[Receive-Queue Map]
    RQM --> E1[Executor1]
    RQM --> E2[Executor2]
    RQM --> E3[Executor3]
    E1 --> TQ((Transfer Queue))
    E2 --> TQ
    E3 --> TQ
    TQ --> TT[Transfer Thread]
    TT --> SCM[Send-Connection Map]
    SCM --> NP1((node+port1))
    SCM --> NP2a((node+port2))
    SCM --> NP2b((node+port2))
  
```

- 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(yyyy)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 达到一定的阈值，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。

8.2.1 Worker 进程间技术(Netty、ZeroMQ)

8.2.1.1 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

8.2.1.2 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”。

8.2.2 Worker 内部通信技术(Disruptor)

8.2.2.1 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 百万分之一秒内）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。

8.2.2.2 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）。

8.2.2.3 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。

8.2.2.4 Disruptor 核心技术点

Disruptor 可以看成是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。

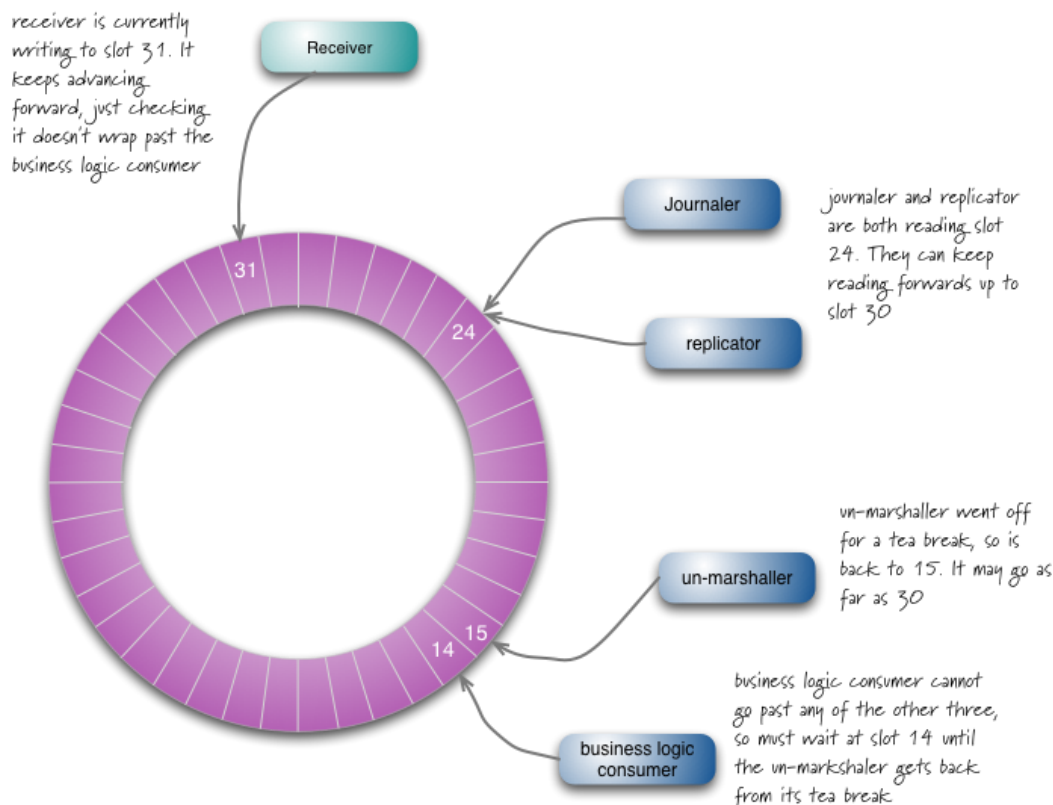
底层是单个数据结构：一个 ring buffer。

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数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、Journaler 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。

第9章 消息不丢失机制

9.1 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

9.2 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)`

9.3 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

9.4 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

