# 项目1-Storm安装与编程实践

## 实验目的

掌握Storm的安装配置，了解Storm相关API，掌握storm的初级编程方法，掌握storm应用程序的编译打包、部署运行的方法。

## 实验准备

Linux环境

zookeeper环境

kafka环境

Java 环境

## 注意事项

1.用户的权限分配；

2.环境变量配置与生效；

## 知识准备

Twitter Storm是一个免费、开源的分布式实时计算系统，Storm对于实时计算的意义类似于Hadoop对于批处理的意义，Storm可以简单、高效、可靠地处理流数据，并支持多种编程语言；Storm框架可以方便地与数据库系统进行整合，从而开发出强大的实时计算系统。

### 1. Storm的特点

Storm可用于许多领域中，如实时分析、在线机器学习、持续计算、远程RPC、数据提取加载转换等。

Storm具有以下主要特点：

* 整合性：Storm可方便地与队列系统和数据库系统进行整合；
* 简易的API：Storm的API在使用上即简单又方便；
* 可扩展性：Storm的并行特性使其可以运行在分布式集群中；
* 容错性：Storm可自动进行故障节点的重启、任务的重新分配；
* 可靠的消息处理：Storm保证每个消息都能完整处理；
* 支持各种编程语言：Storm支持使用各种编程语言来定义任务；
* 快速部署：Storm可以快速进行部署和使用；
* 免费、开源：Storm是一款开源框架，可以免费使用。

### 2. Storm设计思想

Storm主要术语包括Nimbus、Supervisor、Worker、Task、Streams、Spouts、Bolts、Topology、Stream Groupings和Reliability；

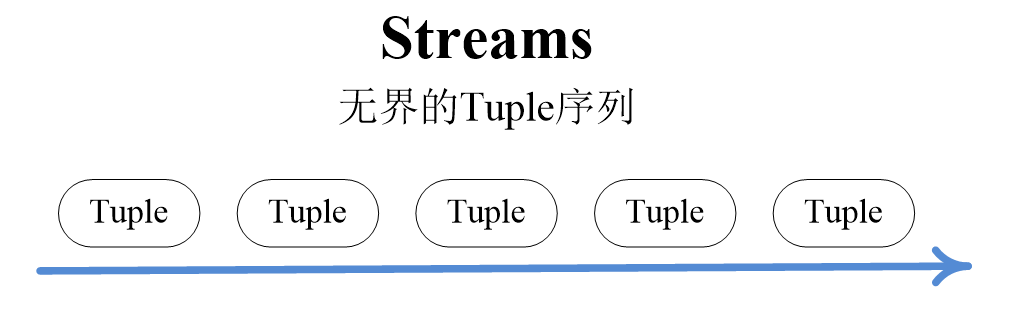
**Nimbus：**即Storm的Master，负责资源分配和任务调度。一个Storm集群只有一个Nimbus。

**Supervisor：**即Storm的Slave，负责接收Nimbus分配的任务，管理所有Worker，一个Supervisor节点中包含多个Worker进程。

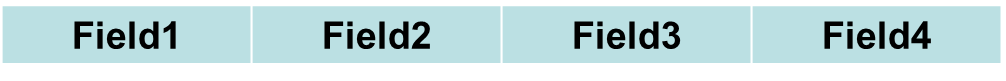
**Worker：**工作进程，每个工作进程中都有多个Task。

**Task：**任务，在 Storm 集群中每个 Spout 和 Bolt 都由若干个任务（tasks）来执行。每个任务都与一个执行线程相对应。

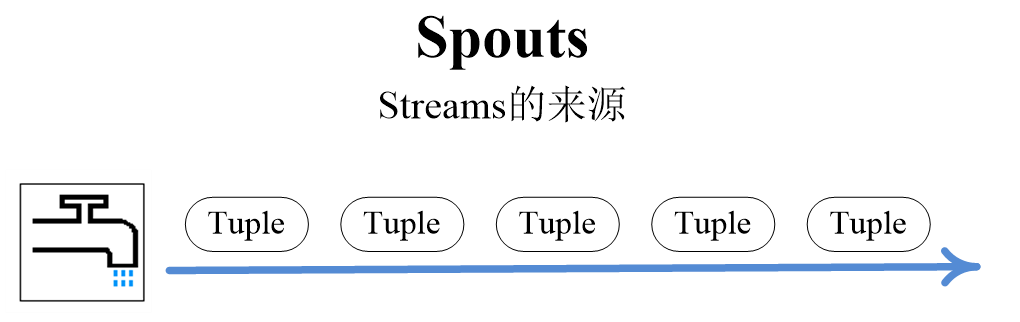
**Streams：**Storm将流数据Stream描述成一个无限的Tuple序列，这些Tuple序列会以分布式的方式并行地创建和处理；



每个tuple是一堆值，每个值有一个名字，并且每个值可以是任何类型；Tuple本来应该是一个Key-Value的Map，由于各个组件间传递的tuple的字段名称已经事先定义好了，所以Tuple只需要按序填入各个Value，所以就是一个Value List（值列表）。



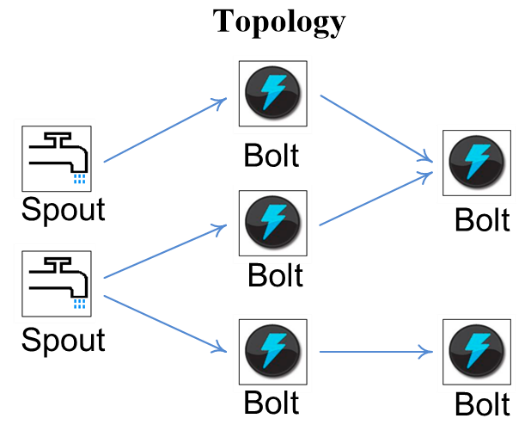
**Spout：**Storm认为每个Stream都有一个源头，并把这个源头抽象为Spout；通常Spout会从外部数据源（队列、数据库等）读取数据，然后封装成Tuple形式，发送到Stream中。Spout是一个主动的角色，在接口内部有个nextTuple函数，Storm框架会不停的调用该函数。



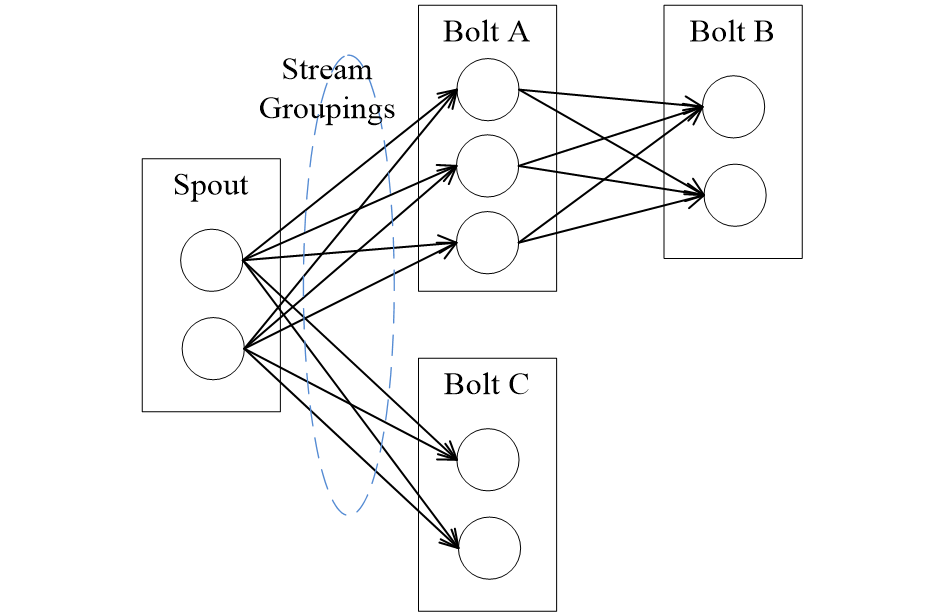
Bolt：Storm将Streams的状态转换过程抽象为Bolt。Bolt即可以处理Tuple，也可以将处理后的Tuple作为新的Streams发送给其他Bolt；Bolt可以执行过滤、函数操作、Join、操作数据库等任何操作；Bolt是一个被动的角色，其接口中有一个execute(Tuple input)方法，在接收到消息之后会调用此函数，用户可以在此方法中执行自己的处理逻辑；



**Topology：**Storm将Spouts和Bolts组成的网络抽象成Topology，它可以被提交到Storm集群执行。Topology可视为流转换图，图中节点是一个Spout或Bolt，边则表示Bolt订阅了哪个Stream。当Spout或者Bolt发送元组时，它会把元组发送到每个订阅了该Stream的Bolt上进行处理；Topology里面的每个处理组件（Spout或Bolt）都包含处理逻辑， 而组件之间的连接则表示数据流动的方向；Topology里面的每一个组件都是并行运行的；在Topology里面可以指定每个组件的并行度， Storm会在集群里面按照并行度分配线程来同时计算；在Topology的具体实现上，Storm中的Topology定义仅仅是一些Thrift结构体（二进制高性能的通信中间件），支持各种编程语言进行定义；



**Stream Groupings：**Storm中的Stream Groupings用于告知Topology如何在两个组件间（如Spout和Bolt之间，或者不同的Bolt之间）进行Tuple的传送。每一个Spout和Bolt都可以有多个分布式任务，一个任务在什么时候、以什么方式发送Tuple就是由Stream Groupings来决定的；



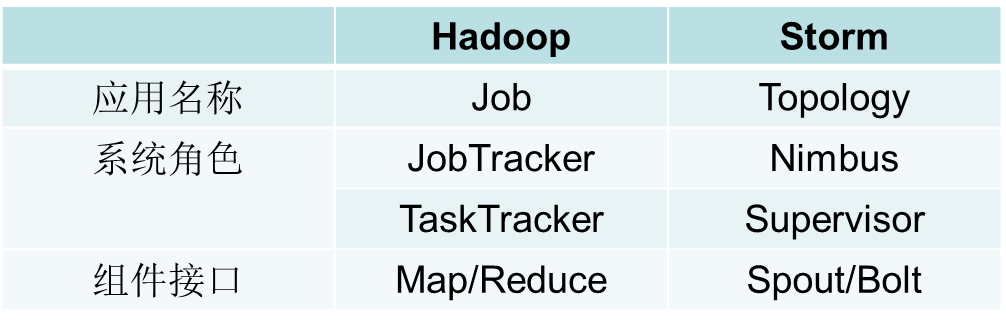
目前，Storm中的Stream Groupings有如下几种方式：

* ShuffleGrouping：随机分组，随机分发Stream中的Tuple，保证每个Bolt的Task接收Tuple数量大致一致；
* FieldsGrouping：按照字段分组，保证相同字段的Tuple分配到同一个Task中；
* AllGrouping：广播发送，每一个Task都会收到所有的Tuple；
* GlobalGrouping：全局分组，所有的Tuple都发送到同一个Task中；
* NonGrouping：不分组，和ShuffleGrouping类似，当前Task的执行会和它的被订阅者在同一个线程中执行；
* DirectGrouping：直接分组，直接指定由某个Task来执行Tuple的处理；

**Reliability：**可靠性。Storm 可以通过拓扑来确保每个发送的元组都能得到正确处理。通过跟踪由 Spout 发出的每个元组构成的元组树可以确定元组是否已经完成处理。每个拓扑都有一个“消息延时”参数，如果 Storm 在延时时间内没有检测到元组是否处理完成，就会将该元组标记为处理失败，并会在稍后重新发送该元组。

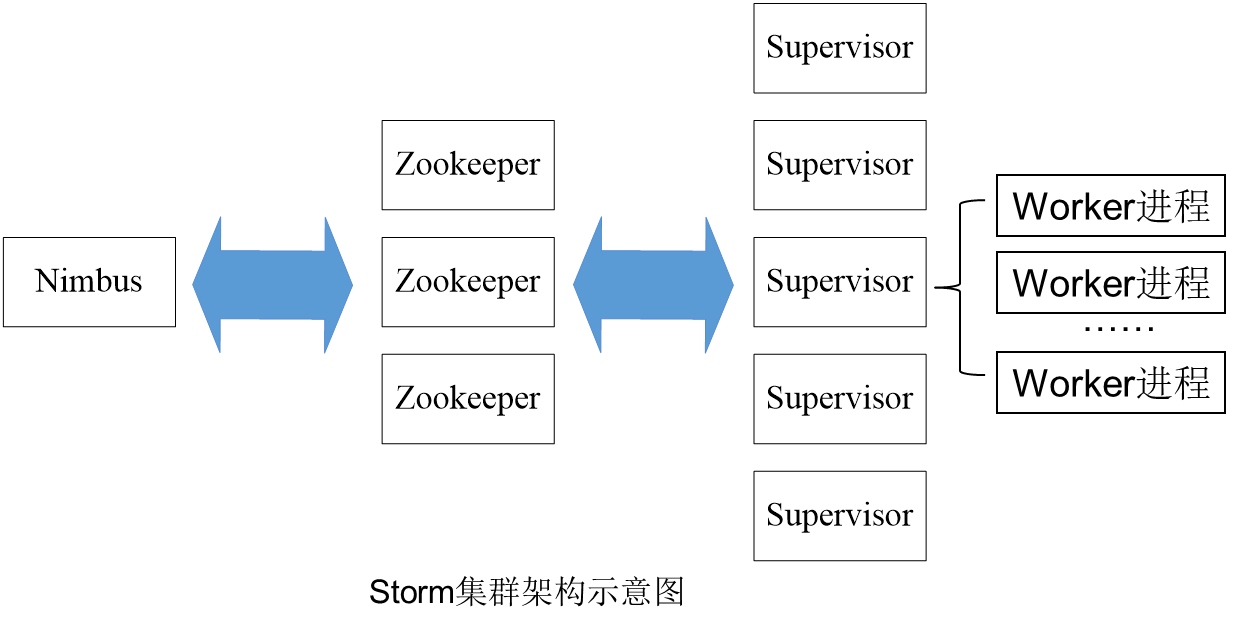
### 3. Storm框架设计

Storm运行任务的方式与Hadoop类似：Hadoop运行的是MapReduce作业，而Storm运行的是“Topology”，但两者的任务大不相同，主要的不同是：MapReduce作业最终会完成计算并结束运行，而Topology将持续处理消息（直到人为终止）。

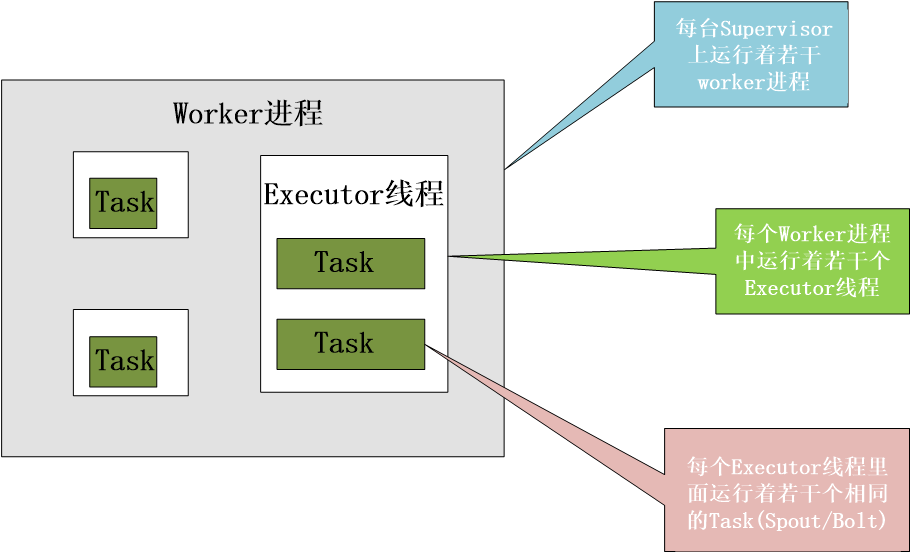


Storm集群采用“Master—Worker”的节点方式：Master节点运行名为“Nimbus”的后台程序（类似Hadoop中的“JobTracker”），负责在集群范围内分发代码、为Worker分配任务和监测故障；Worker节点运行名为“Supervisor”的后台程序，负责监听分配给它所在机器的工作，即根据Nimbus分配的任务来决定启动或停止Worker进程，一个Worker节点上同时运行若干个Worker进程。

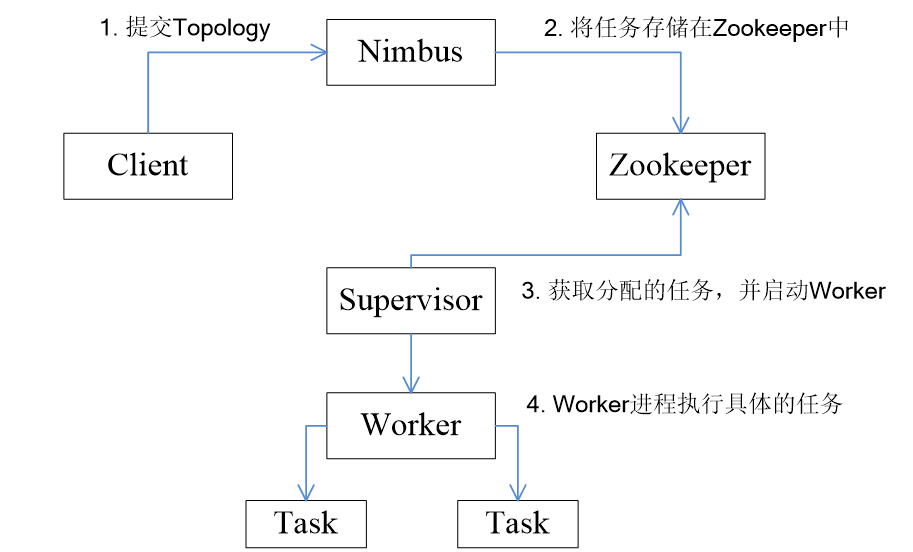
Storm使用Zookeeper作为分布式协调组件，负责Nimbus和多个Supervisor之间的所有协调工作。借助于Zookeeper，若Nimbus进程或Supervisor进程意外终止，重启时也能读取、恢复之前的状态并继续工作，使得Storm极其稳定；



* worker：每个worker进程都属于一个特定的Topology，每个Supervisor节点的worker可以有多个，每个worker对Topology中的每个组件（Spout或 Bolt）运行一个或者多个executor线程来提供task的运行服务。
* executor：executor是产生于worker进程内部的线程，会执行同一个组件的一个或者多个task。
* task：实际的数据处理由task完成，在Topology的生命周期中，每个组件的task数目是不会发生变化的，而executor的数目却不一定。executor数目小于等于task的数目，默认情况下，二者是相等的。



基于这样的架构设计，Storm的工作流程如下图所示：



所有Topology任务的提交必须在Storm客户端节点上进行，提交后由Nimbus节点分配给其他Supervisor节点进行处理；Nimbus节点首先将提交的Topology进行分片，分成一个个Task，分配给相应的Supervisor，并将Task和Supervisor相关的信息提交到Zookeeper集群上；Supervisor会去Zookeeper集群上认领自己的Task，通知自己的Worker进程进行Task的处理。

说明：在提交了一个Topology之后，Storm就会创建Spout/Bolt实例并进行序列化。之后，将序列化的组件发送给所有的任务所在的机器(即Supervisor节点)，在每一个任务上反序列化组件。

### 4. Storm与Spark Streaming比较

Storm和Spark Streaming都是分布式的流处理开源框架，但是它们之间还是有一些区别的，这里将进行比较并指出它们的重要的区别。

1. **处理模型以及延迟**

虽然这两个框架都提供可扩展性(Scalability)和可容错性(Fault Tolerance)，但是它们的处理模型从根本上说是不一样的。Storm处理的是每次传入的一个事件，而Spark Streaming是处理某个时间段窗口内的事件流。因此，Storm处理一个事件可以达到亚秒级的延迟，而Spark Streaming则有秒级的延迟。

1. **容错和数据保证**

在容错数据保证方面的权衡方面，Spark Streaming提供了更好的支持容错状态计算。在Storm中，当每条单独的记录通过系统时必须被跟踪，所以Storm能够至少保证每条记录将被处理一次，但是在从错误中恢复过来时候允许出现重复记录，这意味着可变状态可能不正确地被更新两次。而Spark Streaming只需要在批处理级别对记录进行跟踪处理，因此可以有效地保证每条记录将完全被处理一次，即便一个节点发生故障。虽然Storm的 Trident library库也提供了完全一次处理的功能，但是它依赖于事务更新状态，而这个过程是很慢的，并且通常必须由用户实现。

简而言之，如果你需要亚秒级的延迟，Storm是一个不错的选择，而且没有数据丢失。如果你需要有状态的计算，而且要完全保证每个事件只被处理一次，Spark Streaming则更好。Spark Streaming编程逻辑也可能更容易，因为它类似于批处理程序，特别是在你使用批次(尽管是很小的)时。

1. **实现和编程API**

Storm主要是由Clojure语言实现，Spark Streaming是由Scala实现。如果你想看看这两个框架是如何实现的或者你想自定义一些东西你就得记住这一点。Storm是由BackType和 Twitter开发，而Spark Streaming是在UC Berkeley开发的。

Storm提供了Java API，同时也支持其他语言的API。 Spark Streaming支持Scala和Java语言(其实也支持Python)。另外Spark Streaming的一个很棒的特性就是它是在Spark框架上运行的。这样你就可以想使用其他批处理代码一样来写Spark Streaming程序，或者是在Spark中交互查询。这就减少了单独编写流批量处理程序和历史数据处理程序。

1. **生产支持**

Storm已经出现好多年了，而且自从2011年开始就在Twitter内部生产环境中使用，另外还有其他一些公司在用，而Spark Streaming是一个新的项目。

Storm是Hortonworks Hadoop数据平台中流处理的解决方案，而Spark Streaming出现在MapR的分布式平台和Cloudera的企业数据平台中。除此之外，Databricks是为Spark提供技术支持的公司，包括了Spark Streaming。

1. **集群管理集成**

尽管两个系统都运行在它们自己的集群上，Storm也能运行在Mesos，而Spark Streaming能运行在YARN和Mesos上。

### 5. Storm应用运行

运行Storm计算任务，就是提交Topology。运行一个Topology是很简单的，首先把所有的代码以及所依赖的jar打进一个jar包中，然后运行类似下面的这个命令：

strom jar all-your-code.jar backtype.storm.MyTopology arg1 arg2

storm jar负责连接到nimbus并且上传jar文件。

## 任务1：Storm安装配置

### 任务实施

#### 1.1. 准备工作

分布式实时流计算框架 Storm广泛应用于实时日志分析、个性化推荐、实时监控等应用场景中。下面介绍如何在单机上安装、运行Storm。

具体运行环境如下：

CentOS 7.x

Storm 1.2.3

Java JDK 1.8

ZooKeeper 3.4.6

Python 2.6

#### 1.2 安装Java环境

CentOS中已默认安装了Python 2.6，我们还需要安装 JDK 环境以及分布式应用程序协调服务Zookeeper。

Storm运行需要Java 环境，可选择Oracle的JDK，或是OpenJDK，现在一般Linux系统默认安装的基本是OpenJDK。需要注意的是，CentOS 6.4 中默认安装的只是Java JRE，而不是JDK，为了开发方便，我们还是需要通过yum进行安装JDK，安装过程中会让输入[y/N]，输入y即可：

sudo yum install java-1.8.0-openjdk java-1.8.0-openjdk-devel

通过上述命令安装 OpenJDK，默认安装位置为 /usr/lib/jvm/java-1.8.0-openjdk（该路径可以通过执行 rpm -ql java-1.8.0-openjdk-devel | grep '/bin/javac' 命令确定，执行后会输出一个路径，除去路径末尾的 “/bin/javac”，剩下的就是正确的路径了）。OpenJDK 安装后就可以直接使用 java、javac 等命令了。

接着需要配置一下 JAVA\_HOME 环境变量，为方便我们在 ~/.bashrc 中进行设置（扩展阅读: [设置Linux环境变量的方法和区别](http://dblab.xmu.edu.cn/blog/linux-environment-variable/)）：

vim ~/.bashrc

在文件最后面添加如下单独一行（指向JDK的安装位置），并保存：

export JAVA\_HOME=/usr/lib/jvm/java-1.8.0-openjdk

接着还需要让该环境变量生效，执行如下代码：

source ~/.bashrc *# 使变量设置生效*

设置好后我们来检验一下是否设置正确：

echo $JAVA\_HOME *# 检验变量值*

java -version

$JAVA\_HOME/bin/java -version *# 与直接执行 java -version 一样*

如果设置正确的话，$JAVA\_HOME/bin/java -version 会输出 java 的版本信息，且和 java -version 的输出结果一样.

这样，Storm所需的Java运行环境就安装好了。

#### 1.3 安装 Zookeeper

本教程选择安装 zookeeper 稳定版（3.4.6），下载地址：[http://mirrors.cnnic.cn/apache/zookeeper/stable/](http://mirrors.cnnic.cn/apache/zookeeper/stable/" \t "_blank) 或 <http://mirror.bit.edu.cn/apache/zookeeper/stable/> （打开网页，点击 Projects 下的 “zookeeper-3.4.6.tar.gz” 进行下载）。

下载后执行如下命令进行安装 zookeeper（将命令中 3.4.6 改为你下载的版本）：

sudo tar -zxf ~/Downloads/zookeeper-3.4.6.tar.gz -C /usr/local

cd /usr/local

sudo mv zookeeper-\* zookeeper

sudo chown -R hadoop ./zookeeper *# 此处的hadoop为你的用户名*

接着执行如下命令进行zookeeper配置：

cd /usr/local/zookeeper

mkdir tmp

cp ./conf/zoo\_sample.cfg ./conf/zoo.cfg

vim ./conf/zoo.cfg

将当中的 dataDir=/tmp/zookeeper 更改为 dataDir=/usr/local/zookeeper/tmp 。接着执行：

./bin/zkServer.sh start

若显示如下图则表示启动成功（显示 “Starting zookeeper … STARTED”）：



#### 1.4 安装Storm（单机）

本教程所使用的版本为 Storm 1.2.3 ，下载地址：<https://storm.apache.org/downloads.html>；下载后执行如下命令进行安装Storm：

sudo tar -zxf ~/Downloads/apache-storm-1.2.3.tar.gz -C /usr/local

cd /usr/local

sudo mv apache-storm-1.2.3 storm

sudo chown -R hadoop ./storm *# 此处的hadoop为你的用户名*

接着执行如下命令进行Storm配置:

cd /usr/local/storm

vim ./conf/storm.yaml

修改其中的 storm.zookeeper.servers 和 nimbus.host 两个配置项，即取消掉注释且都修改值为 mnode1（我们只需要在单机上运行），如下所示。

storm.local.dir: /usr/local/storm

storm.zookeeper.servers:

- "mnode1"

storm.zookeeper.port: 2181

nimbus.seeds: ["mnode1"]

ui.host: 0.0.0.0

ui.port: 8090

supervisor.slots.ports:

- 6700

- 6701

- 6702

- 6703

简单配置后就可以启动 Storm 了。执行如下命令启动 nimbus 后台进程：

./bin/storm nimbus &

若启动成功则显示如下图内容：



启动 nimbus 后，终端被该进程占用了，不能再继续执行其他命令了。因此我们需要另外开启一个终端，然后执行如下命令启动 supervisor 后台进程：

*# 需要另外开启一个终端*

/usr/local/storm/bin/storm supervisor

同样的，启动 supervisor 后，我们还需要开启另外的终端才能执行其他命令。另外，我们可以使用 jps 命令 检查是否成功启动，若成功启动会显示 nimbus、supervisor、QuorumPeeMain （QuorumPeeMain 是 zookeeper 的后台进程，若显示 config\_value 表明 nimbus 或 supervisor 还在启动中），如下图所示。



#### 1.5 启动Storm可能会遇到的问题

如果启动 nimbus 时有显示 ERROR java.net.UnknownHostException 未知的名称或服务，或者启动supervisor后一会进程就中断了，则需要添加主机名的ip映射。执行 sudo vim /etc/hosts，增加一行 127.0.0.1 dblab，如下图所示。其中 dblab 为主机名，也就是终端的标题 hadoop@dblab:~，@与冒号中间的内容。



#### 1.6 关闭Storm

之前启动的 nimbus 和 supervisor 占用了两个终端窗口，切换到这两个终端窗口，按键盘的 Ctrl+C 可以终止进程，终止后，也就相当于关闭了 Storm。

## 任务2：Storm单词统计编程

Storm进行单词统计的流程说明：

从Spout中发送Stream（每个英文句子为一个Tuple）；用于分割单词的Bolt将接收的句子分解为独立的单词，将单词作为Tuple的字段名发送出去；用于计数的Bolt接收表示单词的Tuple，并对其进行统计；输出每个单词以及单词出现过的次数；

### 任务实施

#### 2.1 代码编写

下面开始进行程序编写，请新打开一个终端，然后，执行命令创建代码目录：

cd /usr/local/storm/

mkdir mycode

cd mycode

mkdir wordcount

cd wordcount

mkdir -p src/main/java

vim src/main/java/TestSpout.java

定义数据源编写Spout类，用来随机生成单词，java类文件名为：TestSpout.java，代码如下：

package com.nhbd.storm.example.wordcount;

import java.util.Map;

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;

/\*\*

\*

\* Title: TestSpout

\* Description:

\* 发送信息

\* Version:1.0.0

\* @author peiyy

\* @date 2021年3月6日

\*/

public class TestSpout extends BaseRichSpout{

private static final long serialVersionUID = 225243592780939490L;

private SpoutOutputCollector collector;

private static final String field="word";

private int count=1;

private String[] message = {

"My nickname is xuwujing",

"My blog address is http://www.panchengming.com/",

"My interest is playing games"

};

/\*\*

\* open()方法中是在ISpout接口中定义，在Spout组件初始化时被调用。

\* 有三个参数:

\* 1.Storm配置的Map;

\* 2.topology中组件的信息;

\* 3.发射tuple的方法;

\*/

@Override

public void open(Map map, TopologyContext arg1, SpoutOutputCollector collector) {

System.out.println("open:"+map.get("test"));

this.collector = collector;

}

/\*\*

\* nextTuple()方法是Spout实现的核心。

\* 也就是主要执行方法，用于输出信息,通过collector.emit方法发射。

\*/

@Override

public void nextTuple() {

if(count<=message.length){

System.out.println("第"+count +"次开始发送数据...");

this.collector.emit(new Values(message[count-1]));

}

count++;

}

/\*\*

\* declareOutputFields是在IComponent接口中定义，用于声明数据格式。

\* 即输出的一个Tuple中，包含几个字段。

\*/

@Override

public void declareOutputFields(OutputFieldsDeclarer declarer) {

System.out.println("定义Tuple格式...");

declarer.declare(new Fields(field));

}

/\*\*

\* 当一个Tuple处理成功时，会调用这个方法

\*/

@Override

public void ack(Object obj) {

System.out.println("TestSpout ack:"+obj);

}

/\*\*

\* 当Topology停止时，会调用这个方法

\*/

@Override

public void close() {

System.out.println("TestSpout关闭...");

}

/\*\*

\* 当一个Tuple处理失败时，会调用这个方法

\*/

@Override

public void fail(Object obj) {

System.out.println("TestSpout失败:"+obj);

}

}

编写单词统计的单词拆分处理类，即通过空格来分割单词，分割后的单词通过emit()方法以Tuple的形式发送给订阅了该Stream的Bolt进行接收和处理，采用命令：

vim src/main/java/Test1Bolt.java 创建java类文件。代码如下：

package com.nhbd.storm.example.wordcount;

import java.util.Map;

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;

/\*\*

\*

\* Title: TestBolt

\* Description:

\* 对单词进行分割

\* Version:1.0.0

\* @author peiyy

\* @date 2021年3月6日

\*/

public class Test1Bolt extends BaseRichBolt{

/\*\*

\*

\*/

private static final long serialVersionUID = 4743224635827696343L;

private OutputCollector collector;

/\*\*

\* 在Bolt启动前执行，提供Bolt启动环境配置的入口

\* 一般对于不可序列化的对象进行实例化。

\* 注:如果是可以序列化的对象，那么最好是使用构造函数。

\*/

@Override

public void prepare(Map map, TopologyContext arg1, OutputCollector collector) {

System.out.println("prepare:"+map.get("test"));

this.collector=collector;

}

/\*\*

\* execute()方法是Bolt实现的核心。

\* 也就是执行方法，每次Bolt从流接收一个订阅的tuple，都会调用这个方法。

\*/

@Override

public void execute(Tuple tuple) {

String msg=tuple.getStringByField("word");

System.out.println("开始分割单词:"+msg);

String[] words = msg.toLowerCase().split(" ");

for (String word : words) {

this.collector.emit(new Values(word));//向下一个bolt发射数据

}

}

/\*\*

\* 声明数据格式

\*/

@Override

public void declareOutputFields(OutputFieldsDeclarer declarer) {

declarer.declare(new Fields("count"));

}

/\*\*

\* cleanup是IBolt接口中定义,用于释放bolt占用的资源。

\* Storm在终止一个bolt之前会调用这个方法。

\*/

@Override

public void cleanup() {

System.out.println("Test1Bolt的资源释放");

}

}

编写单词统计的词频统计处理类，采用命令：vim src/main/java/Test2Bolt.java 创建java类文件名为：Test2Bolt.java。代码如下：

package com.nhbd.storm.example.wordcount;

import java.util.HashMap;

import java.util.Map;

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;

/\*\*

\*

\* Title: Test2Bolt

\* Description:

\* 统计单词出现的次数

\* Version:1.0.0

\* @author peiyy

\* @date 2021年3月6日

\*/

public class Test2Bolt extends BaseRichBolt{

/\*\*

\*

\*/

private static final long serialVersionUID = 4743224635827696343L;

/\*\*

\* 保存单词和对应的计数

\*/

private HashMap<String, Integer> counts = null;

private long count=1;

/\*\*

\* 在Bolt启动前执行，提供Bolt启动环境配置的入口

\* 一般对于不可序列化的对象进行实例化。

\* 注:如果是可以序列化的对象，那么最好是使用构造函数。

\*/

@Override

public void prepare(Map map, TopologyContext arg1, OutputCollector collector) {

System.out.println("prepare:"+map.get("test"));

this.counts=new HashMap<String, Integer>();

}

/\*\*

\* execute()方法是Bolt实现的核心。

\* 也就是执行方法，每次Bolt从流接收一个订阅的tuple，都会调用这个方法。

\*

\*/

@Override

public void execute(Tuple tuple) {

String msg=tuple.getStringByField("count");

System.out.println("第"+count+"次统计");

/\*\*

\* 如果不包含该单词，说明在该map是第一次出现

\* 否则进行加1

\*/

if (!counts.containsKey(msg)) {

counts.put(msg, 1);

} else {

counts.put(msg, counts.get(msg)+1);

}

count++;

}

/\*\*

\* cleanup是IBolt接口中定义,用于释放bolt占用的资源。

\* Storm在终止一个bolt之前会调用这个方法。

\*/

@Override

public void cleanup() {

System.out.println("===========开始显示单词数量============");

for (Map.Entry<String, Integer> entry : counts.entrySet()) {

System.out.println("word "+entry.getKey() + ": " + entry.getValue());

}

System.out.println("===========结束============");

System.out.println("Test2Bolt的资源释放");

}

/\*\*

\* 声明数据格式

\*/

@Override

public void declareOutputFields(OutputFieldsDeclarer arg0) {

}

}

编写单词统计的storm拓扑类，采用命令：vim src/main/java/ TopologyApp.java 创建java类文件名为：TopologyApp.java。代码如下：

package com.nhbd.storm.example.wordcount;

import org.apache.storm.Config;

import org.apache.storm.LocalCluster;

import org.apache.storm.StormSubmitter;

import org.apache.storm.topology.TopologyBuilder;

import org.apache.storm.tuple.Fields;

/\*\*

\*

\* Title: TopologyApp

\* Description:

\* storm wordcount 测试

\* Version:1.0.0

\* @author peiyy

\* @date 2021年3月6日

\*/

public class TopologyApp {

private static final String test\_spout="test\_spout";

private static final String test1\_bolt="test1\_bolt";

private static final String test2\_bolt="test2\_bolt";

public static void main(String[] args) {

//定义一个拓扑

TopologyBuilder builder=new TopologyBuilder();

//设置两个Executeor(线程)，默认一个

builder.setSpout(test\_spout, new TestSpout(),1);

//shuffleGrouping:表示是随机分组

//设置两个Executeor(线程)，和两个task

builder.setBolt(test1\_bolt, new Test1Bolt(),2).setNumTasks(2).shuffleGrouping(test\_spout);

//fieldsGrouping:表示是按字段分组

//设置两个Executeor(线程)，和两个task

builder.setBolt(test2\_bolt, new Test2Bolt(),2).setNumTasks(2).fieldsGrouping(test1\_bolt, new Fields("count"));

Config conf = new Config();

conf.put("test", "test");

try{

//运行拓扑

if(args !=null&&args.length>0){ //有参数时，表示向集群提交作业，并把第一个参数当做topology名称

System.out.println("运行远程模式");

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

} else{//没有参数时，本地提交

//启动本地模式

System.out.println("运行本地模式");

LocalCluster cluster = new LocalCluster();

cluster.submitTopology("Word-counts" ,conf, builder.createTopology() );

Thread.sleep(20000);

//关闭本地集群

cluster.shutdown();

}

}catch (Exception e){

e.printStackTrace();

}

}

}

下面开始pom文件编写，请新打开一个终端，然后，执行命令创建pom.xml：

cd /usr/local/storm/mycode/wordcount

vim pom.xml

pom.xml代码如下：

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns="http://maven.apache.org/POM/4.0.0"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.nhbd.storm</groupId>

<artifactId>storm-example</artifactId>

<version>1.0</version>

<properties>

<storm.version>1.2.2</storm.version>

<kafka.version>2.2.0</kafka.version>

</properties>

<dependencies>

<dependency>

<groupId>org.apache.storm</groupId>

<artifactId>storm-core</artifactId>

<version>${storm.version}</version>

</dependency>

</dependencies>

</project>

#### 2.2 打包编译

在此之前请先开启 Storm （即开启 zookeeper，nimbus 和 supervisor），并确定系统处于联网状态，因为在编译过程中需要下载不少依赖包。

接着我们开始使用maven编译、运行wordcount项目。我们只需要进入wordcount所在目录，再执行相应的 mvn 命令即可：

cd /usr/local/storm/mycode/wordcount

mvn package

mvn 命令需要在项目的根目录中运行，即在wordcount目录中运行，该命令会查找当前目录下的pom.xml文件，下载所需要的依赖包。首次运行需要下载较多东西，请耐心等待。最后依赖包完成下载并完成打包后直接提交到 Storm 中运行。

#### 2.3 实例运行

打包完成后，会在 target 文件中生成storm-example-1.0.jar，就可以把 jar 包提交到 Storm 中运行了。

/usr/local/storm/bin/storm jar ./target/storm-example-1.0.jar com.nhbd.storm.example.wordcount.TopologyApp

后续我们编写自己的代码时，建议使用 maven 来方便的编译、运行代码。

## 任务3：Storm整合kafka

Storm 官方对 Kafka 的整合分为两个版本，官方说明文档分别如下：

**Storm Kafka Integration:** 主要是针对 0.8.x 版本的 Kafka 提供整合支持；

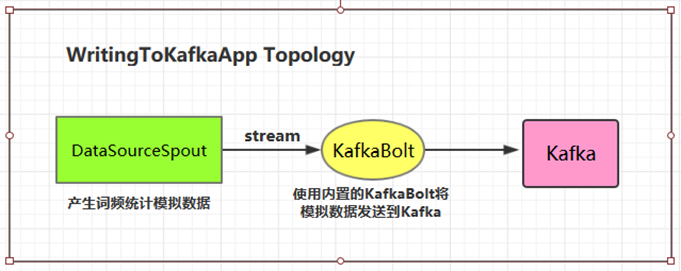
**Storm Kafka Integration(0.10.x+):** 包含 Kafka 新版本的 consumer API，主要对 Kafka 0.10.x + 提供整合支持。

这里我们安装的版本是2.x.版本，按照官方0.10.x+的整合文档进行整合。

### 任务实施

#### 3.1写入数据到Kafka

##### 3.1.1 项目结构



##### 3.1.2 项目主要依赖

pom.xml代码如下：

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns="http://maven.apache.org/POM/4.0.0"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.nhbd.storm</groupId>

<artifactId>storm-example</artifactId>

<version>1.0</version>

<properties>

<storm.version>1.2.2</storm.version>

<kafka.version>2.2.0</kafka.version>

</properties>

<dependencies>

<dependency>

<groupId>org.apache.storm</groupId>

<artifactId>storm-core</artifactId>

<version>${storm.version}</version>

</dependency>

<dependency>

<groupId>org.apache.storm</groupId>

<artifactId>storm-kafka-client</artifactId>

<version>${storm.version}</version>

</dependency>

<dependency>

<groupId>org.apache.kafka</groupId>

<artifactId>kafka-clients</artifactId>

<version>${kafka.version}</version>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>8</source>

<target>8</target>

</configuration>

</plugin>

<!--使用shade进行打包-->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-shade-plugin</artifactId>

<configuration>

<createDependencyReducedPom>true</createDependencyReducedPom>

<filters>

<filter>

<artifact>\*:\*</artifact>

<excludes>

<exclude>META-INF/\*.SF</exclude>

<exclude>META-INF/\*.sf</exclude>

<exclude>META-INF/\*.DSA</exclude>

<exclude>META-INF/\*.dsa</exclude>

<exclude>META-INF/\*.RSA</exclude>

<exclude>META-INF/\*.rsa</exclude>

<exclude>META-INF/\*.EC</exclude>

<exclude>META-INF/\*.ec</exclude>

<exclude>META-INF/MSFTSIG.SF</exclude>

<exclude>META-INF/MSFTSIG.RSA</exclude>

</excludes>

</filter>

</filters>

<artifactSet>

<excludes>

<exclude>org.apache.storm:storm-core</exclude>

</excludes>

</artifactSet>

</configuration>

<executions>

<execution>

<phase>package</phase>

<goals>

<goal>shade</goal>

</goals>

<configuration>

<transformers>

<transformer

implementation="org.apache.maven.plugins.shade.resource.ServicesResourceTransformer"/>

<transformer

implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">

</transformer>

</transformers>

</configuration>

</execution>

</executions>

</plugin>

</plugins>

</build>

</project>

##### 3.1.3 定义Spout

我们定义一个类，模拟产生如下所示格式的数据：

Spark HBase

Hive Flink Storm Hadoop HBase Spark

Flink

HBase Storm

HBase Hadoop Hive Flink

HBase Flink Hive Storm

Hive Flink Hadoop

HBase Hive

Hadoop Spark HBase Storm

DataSourceSpout代码如下：

package com.nhbd.storm.example.kafka.write;

import org.apache.storm.shade.org.apache.commons.lang.StringUtils;

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 org.apache.storm.utils.Utils;

import java.util.\*;

/\*\*

\* 产生词频样本的数据源

\*/

public class DataSourceSpout extends BaseRichSpout {

private List<String> list = Arrays.asList("Spark", "Hadoop", "HBase", "Storm", "Flink", "Hive");

private SpoutOutputCollector spoutOutputCollector;

/\*\*

\* open()方法中是在ISpout接口中定义，在Spout组件初始化时被调用。

\* 有三个参数:

\* 1.Storm配置的Map;

\* 2.topology中组件的信息;

\* 3.发射tuple的方法;

\*/

@Override

public void open(Map map, TopologyContext topologyContext, SpoutOutputCollector spoutOutputCollector) {

this.spoutOutputCollector = spoutOutputCollector;

}

/\*\*

\* nextTuple()方法是Spout实现的核心。

\* 也就是主要执行方法，用于输出信息,通过collector.emit方法发射。

\*/

@Override

public void nextTuple() {

// 模拟产生数据

String lineData = productData();

spoutOutputCollector.emit(new Values("key",lineData));

Utils.sleep(1000);

}

@Override

public void declareOutputFields(OutputFieldsDeclarer outputFieldsDeclarer) {

outputFieldsDeclarer.declare( new Fields("key", "message"));

}

/\*\*

\* 模拟数据

\*/

private String productData() {

Collections.shuffle(list);

Random random = new Random();

int endIndex = random.nextInt(list.size()) % (list.size()) + 1;

return StringUtils.join(list.toArray(), "\t", 0, endIndex);

}

}

##### 3.1.4 定义Topology

创建WritingToKafkaApp类，用来定义写入Kafka队列信息，WritingToKafkaApp代码如下：

package com.nhbd.storm.example.kafka.write;

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.bolt.KafkaBolt;

import org.apache.storm.kafka.bolt.mapper.FieldNameBasedTupleToKafkaMapper;

import org.apache.storm.kafka.bolt.selector.DefaultTopicSelector;

import org.apache.storm.topology.TopologyBuilder;

import java.util.Properties;

/\*\*

\* 写入数据到Kafka中

\*/

public class WritingToKafkaApp {

private static final String BOOTSTRAP\_SERVERS = "mnode1:9092";

private static final String TOPIC\_NAME = "storm-topic";

public static void main(String[] args) {

TopologyBuilder builder = new TopologyBuilder();

// 定义Kafka生产者属性

Properties props = new Properties();

/\*

\* 指定broker的地址清单，清单里不需要包含所有的broker地址，生产者会从给定的broker里查找其他broker的信息。

\* 不过建议至少要提供两个broker的信息作为容错。

\*/

props.put("bootstrap.servers", BOOTSTRAP\_SERVERS);

/\*

\* acks 参数指定了必须要有多少个分区副本收到消息，生产者才会认为消息写入是成功的。

\* acks=0 : 生产者在成功写入消息之前不会等待任何来自服务器的响应。

\* acks=1 : 只要集群的首领节点收到消息，生产者就会收到一个来自服务器成功响应。

\* acks=all : 只有当所有参与复制的节点全部收到消息时，生产者才会收到一个来自服务器的成功响应。

\*/

props.put("acks", "1");

props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");

props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

KafkaBolt bolt = new KafkaBolt<String, String>()

.withProducerProperties(props)

.withTopicSelector(new DefaultTopicSelector(TOPIC\_NAME))

.withTupleToKafkaMapper(new FieldNameBasedTupleToKafkaMapper<>());

builder.setSpout("sourceSpout", new DataSourceSpout(), 1);

builder.setBolt("kafkaBolt", bolt, 1).shuffleGrouping("sourceSpout");

if (args.length > 0 && args[0].equals("cluster")) {

try {

StormSubmitter.submitTopology("ClusterWritingToKafkaApp", new Config(), builder.createTopology());

} catch (AlreadyAliveException | InvalidTopologyException | AuthorizationException e) {

e.printStackTrace();

}

} else {

LocalCluster cluster = new LocalCluster();

cluster.submitTopology("LocalWritingToKafkaApp",

new Config(), builder.createTopology());

}

}

}

##### 3.1.5测试准备

进行测试前需要启动 Kakfa：

1. 启动Kakfa

Kafka的运行依赖zookeeper，需要预先启动。可以启动Kafka内置的 zookeeper，也可以启动自己安装的zookeeper，这里我们启动独立安装的zookeeper，命令如下：

cd /usr/local/zookeeper

# zookeeper启动命令

bin/zkServer.sh start

启动单节点kafka用于测试，命令如下：

cd /usr/local/kafka

# 启动kafka服务命令

bin/kafka-server-start.sh config/server.properties

2. 创建topic

# 创建用于测试主题

bin/kafka-topics.sh --create --bootstrap-server mnode1:9092 --replication-factor 1 --partitions 1 --topic storm-topic

# 查看所有主题

bin/kafka-topics.sh --list --bootstrap-server mnode1:9092

3. 启动消费者

启动一个消费者用于观察写入情况，启动命令如下：

# 启动消费者

bin/kafka-console-consumer.sh --bootstrap-server mnode1:9092 --topic storm-topic --from-beginning

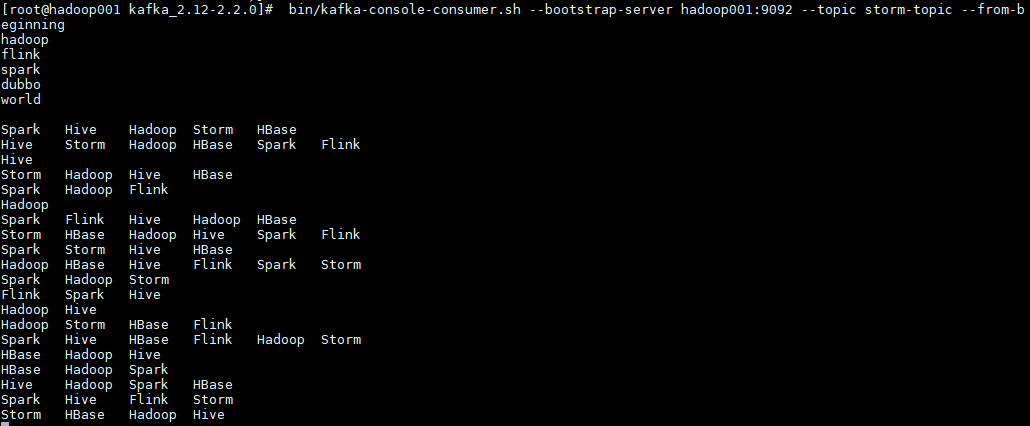
##### 2.6 测试

可以用直接使用本地模式运行，也可以打包后提交到服务器集群运行。本仓库提供的源码默认采用 maven-shade-plugin 进行打包，打包命令如下：

# 进度maven打包

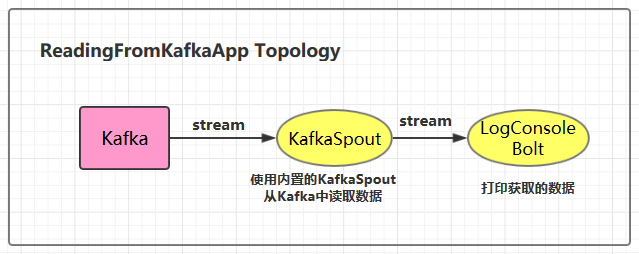
mvn clean package -D maven.test.skip=true

启动后，消费者监听情况如下：



#### 3.2从Kafka中读取数据

##### 3.2.1 项目结构



##### 3.1.2 项目主要依赖

pom.xml代码如下：

<?xml version="1.0" encoding="UTF-8"?>

<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns="http://maven.apache.org/POM/4.0.0"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.nhbd.storm</groupId>

<artifactId>storm-example</artifactId>

<version>1.0</version>

<properties>

<storm.version>1.2.2</storm.version>

<kafka.version>2.2.0</kafka.version>

</properties>

<dependencies>

<dependency>

<groupId>org.apache.storm</groupId>

<artifactId>storm-core</artifactId>

<version>${storm.version}</version>

</dependency>

<dependency>

<groupId>org.apache.storm</groupId>

<artifactId>storm-kafka-client</artifactId>

<version>${storm.version}</version>

</dependency>

<dependency>

<groupId>org.apache.kafka</groupId>

<artifactId>kafka-clients</artifactId>

<version>${kafka.version}</version>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>8</source>

<target>8</target>

</configuration>

</plugin>

<!--使用shade进行打包-->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-shade-plugin</artifactId>

<configuration>

<createDependencyReducedPom>true</createDependencyReducedPom>

<filters>

<filter>

<artifact>\*:\*</artifact>

<excludes>

<exclude>META-INF/\*.SF</exclude>

<exclude>META-INF/\*.sf</exclude>

<exclude>META-INF/\*.DSA</exclude>

<exclude>META-INF/\*.dsa</exclude>

<exclude>META-INF/\*.RSA</exclude>

<exclude>META-INF/\*.rsa</exclude>

<exclude>META-INF/\*.EC</exclude>

<exclude>META-INF/\*.ec</exclude>

<exclude>META-INF/MSFTSIG.SF</exclude>

<exclude>META-INF/MSFTSIG.RSA</exclude>

</excludes>

</filter>

</filters>

<artifactSet>

<excludes>

<exclude>org.apache.storm:storm-core</exclude>

</excludes>

</artifactSet>

</configuration>

<executions>

<execution>

<phase>package</phase>

<goals>

<goal>shade</goal>

</goals>

<configuration>

<transformers>

<transformer

implementation="org.apache.maven.plugins.shade.resource.ServicesResourceTransformer"/>

<transformer

implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">

</transformer>

</transformers>

</configuration>

</execution>

</executions>

</plugin>

</plugins>

</build>

</project>

##### 3.2.3 定义Bolt

我们定义一个类LogConsoleBolt，打印从Kafka中获取的数据，LogConsoleBolt代码如下：

package com.nhbd.storm.example.kafka.read;

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 java.util.Map;

/\*\*

\* 打印从Kafka中获取的数据

\*/

public class LogConsoleBolt extends BaseRichBolt {

private OutputCollector collector;

public void prepare(Map stormConf, TopologyContext context, OutputCollector collector) {

this.collector=collector;

}

public void execute(Tuple input) {

try {

String value = input.getStringByField("value");

System.out.println("received from kafka : "+ value);

// 必须ack,否则会重复消费kafka中的消息

collector.ack(input);

}catch (Exception e){

e.printStackTrace();

collector.fail(input);

}

}

public void declareOutputFields(OutputFieldsDeclarer declarer) {

}

}

这里从value字段中获取kafka输出的值数据。在开发中，我们可以通过继承RecordTranslator 接口定义了Kafka中Record与输出流之间的映射关系，可以在构建KafkaSpoutConfig的时候通过构造器或者setRecordTranslator() 方法传入，并最后传递给具体的KafkaSpout。

##### 3.1.4 定义Topology

创建ReadingFromKafkaApp类，用来定义读取Kafka队列信息，ReadingFromKafkaApp代码如下：

package com.nhbd.storm.example.kafka.read;

import org.apache.kafka.clients.consumer.ConsumerConfig;

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.kafka.spout.KafkaSpoutRetryExponentialBackoff;

import org.apache.storm.kafka.spout.KafkaSpoutRetryExponentialBackoff.TimeInterval;

import org.apache.storm.kafka.spout.KafkaSpoutRetryService;

import org.apache.storm.topology.TopologyBuilder;

/\*\*

\* 从Kafka中读取数据

\*/

public class ReadingFromKafkaApp {

private static final String BOOTSTRAP\_SERVERS = "mnode1:9092";

private static final String TOPIC\_NAME = "storm-topic";

public static void main(String[] args) {

final TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("kafka\_spout", new KafkaSpout<>(getKafkaSpoutConfig(BOOTSTRAP\_SERVERS, TOPIC\_NAME)), 1);

builder.setBolt("bolt", new LogConsoleBolt()).shuffleGrouping("kafka\_spout");

// 如果外部传参cluster则代表线上环境启动,否则代表本地启动

if (args.length > 0 && args[0].equals("cluster")) {

try {

StormSubmitter.submitTopology("ClusterReadingFromKafkaApp", new Config(), builder.createTopology());

} catch (AlreadyAliveException | InvalidTopologyException | AuthorizationException e) {

e.printStackTrace();

}

} else {

LocalCluster cluster = new LocalCluster();

cluster.submitTopology("LocalReadingFromKafkaApp",

new Config(), builder.createTopology());

}

}

private static KafkaSpoutConfig<String, String> getKafkaSpoutConfig(String bootstrapServers, String topic) {

return KafkaSpoutConfig.builder(bootstrapServers, topic)

// 除了分组ID,以下配置都是可选的。分组ID必须指定,否则会抛出InvalidGroupIdException异常

.setProp(ConsumerConfig.GROUP\_ID\_CONFIG, "kafkaSpoutTestGroup")

// 定义重试策略

.setRetry(getRetryService())

// 定时提交偏移量的时间间隔,默认是15s

.setOffsetCommitPeriodMs(10\_000)

.build();

}

// 定义重试策略

private static KafkaSpoutRetryService getRetryService() {

return new KafkaSpoutRetryExponentialBackoff(TimeInterval.microSeconds(500),

TimeInterval.milliSeconds(2), Integer.MAX\_VALUE, TimeInterval.seconds(10));

}

}

##### 3.1.5测试准备

进行测试前需要启动 Kakfa：

1. 启动Kakfa

Kafka的运行依赖zookeeper，需要预先启动。可以启动Kafka内置的 zookeeper，也可以启动自己安装的zookeeper，这里我们启动独立安装的zookeeper，命令如下：

cd /usr/local/zookeeper

# zookeeper启动命令

bin/zkServer.sh start

启动单节点kafka用于测试，命令如下：

cd /usr/local/kafka

# 启动kafka服务命令

bin/kafka-server-start.sh config/server.properties

2. 创建topic

# 创建用于测试主题

bin/kafka-topics.sh --create --bootstrap-server mnode1:9092 --replication-factor 1 --partitions 1 --topic storm-topic

# 查看所有主题

bin/kafka-topics.sh --list --bootstrap-server mnode1:9092

3. 启动生产者

启动一个生产者用于创建消息数据，启动命令如下：

# 启动生产者

bin/kafka-console-producer.sh --broker-list mnode1:9092 --topic storm-topic

##### 2.6 测试

可以用直接使用本地模式运行，也可以打包后提交到服务器集群运行。本仓库提供的源码默认采用 maven-shade-plugin 进行打包，打包命令如下：

# 进度maven打包

mvn clean package -D maven.test.skip=true

启动后，代码监听执行情况如下：

