1 storm运行模式

1、本地模式(Local Mode)： 即Topology（相当于一个任务，后续会详细讲解） 运行在本地机器的单一JVM上，这个模式主要用来开发、调试。

2、远程模式(Remote Mode):在这个模式，我们把我们的Topology提交到集群，在这个模式中，Storm的所有组件都是线程安全的，因为它们都会运行在不同的Jvm或物理机器上，这个模式就是正式的生产模式。

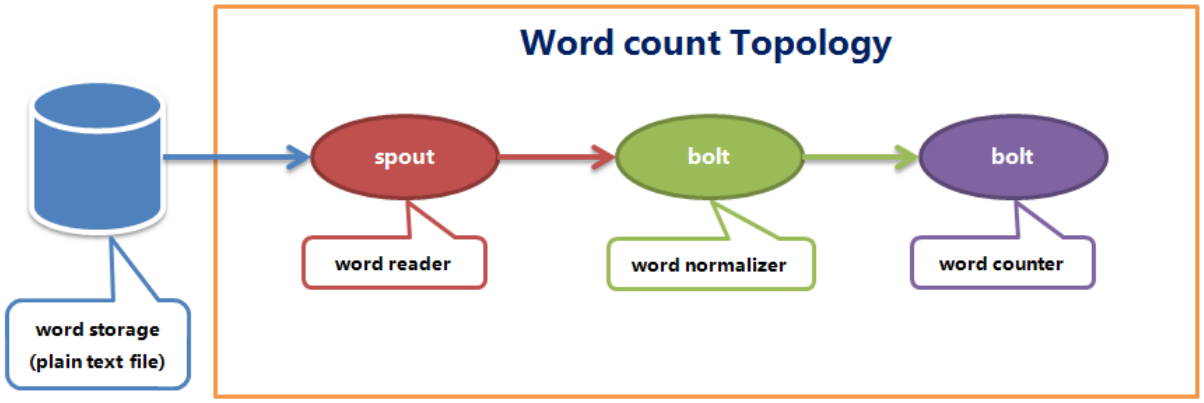
2 需求

对文本文件中的单词进行计数，以下是文本文件中的内容，分隔符为制表符（\t）：

```
storm	hadoop	hive zookeeper	spark
hadoop	hadoop	storm	spark	storm
storm	spark	storm	hadoop	hadoop
```

3 storm实现wordcount（本地模式）

我们需要创建一这样的Topology，用一个spout负责读取文本文件，用第一个bolt来解析成单词，用第二个bolt来对解析出的单词计数，整体结构如图所示：



3.1 搭建基础开发环境

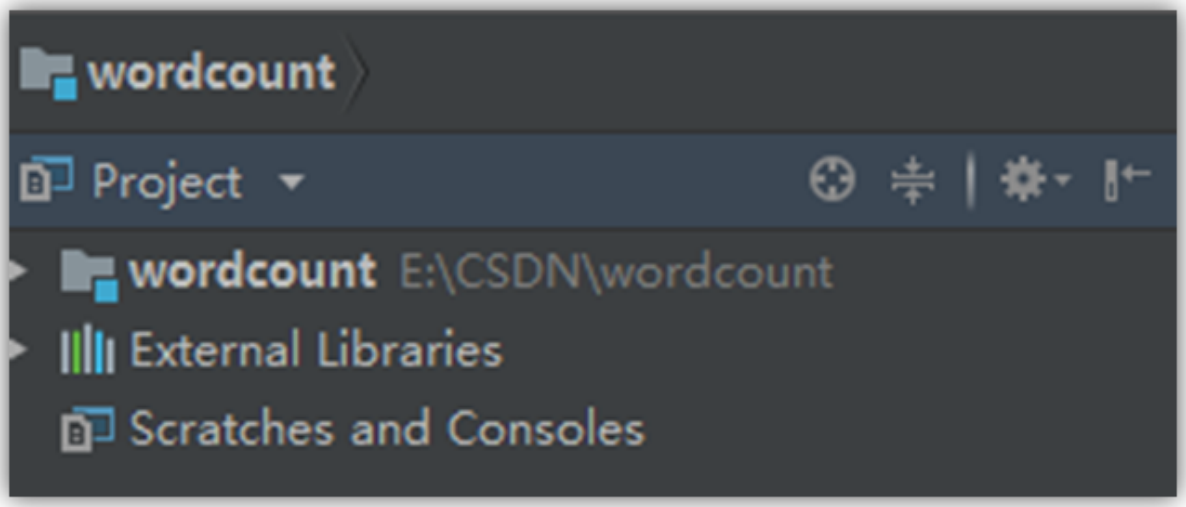
3.1.1 开发工具与JDK

idea

JDK 1.8

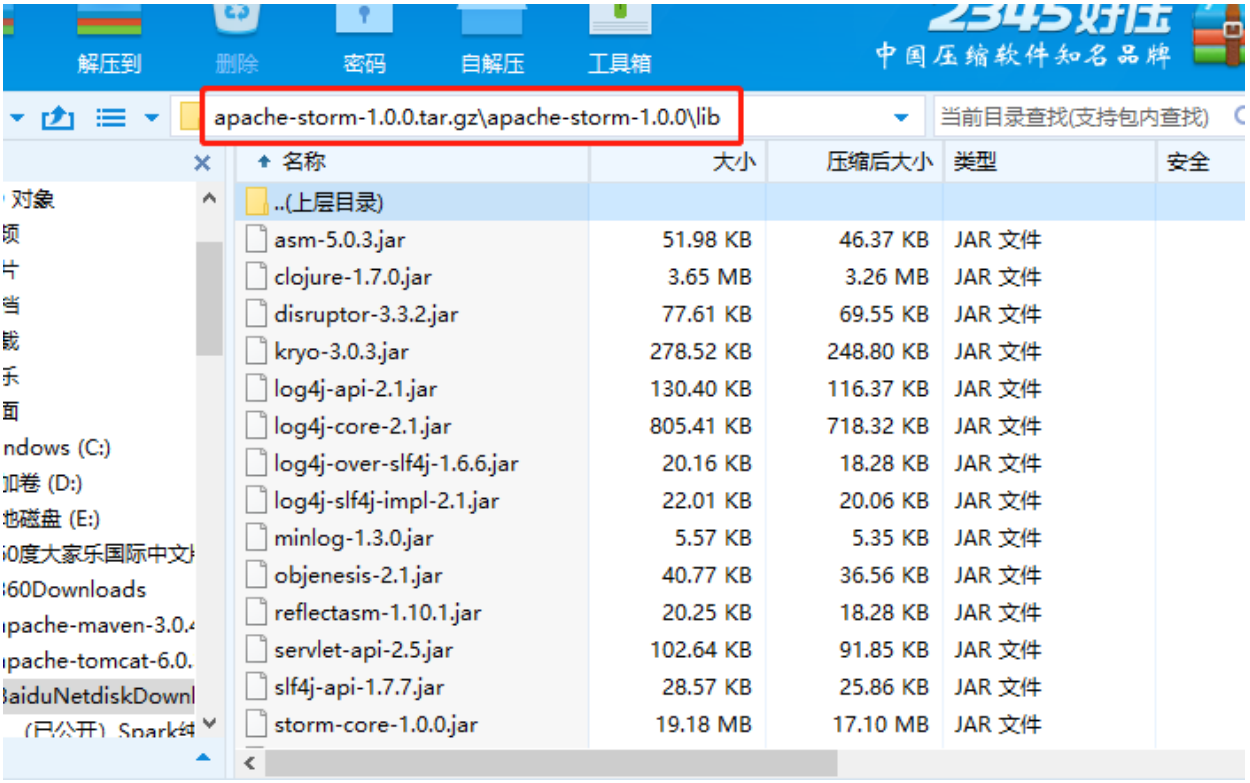
3.1.2 创建项目

创建一个java project，项目名为wordcount



3.1.3 导入相关jar

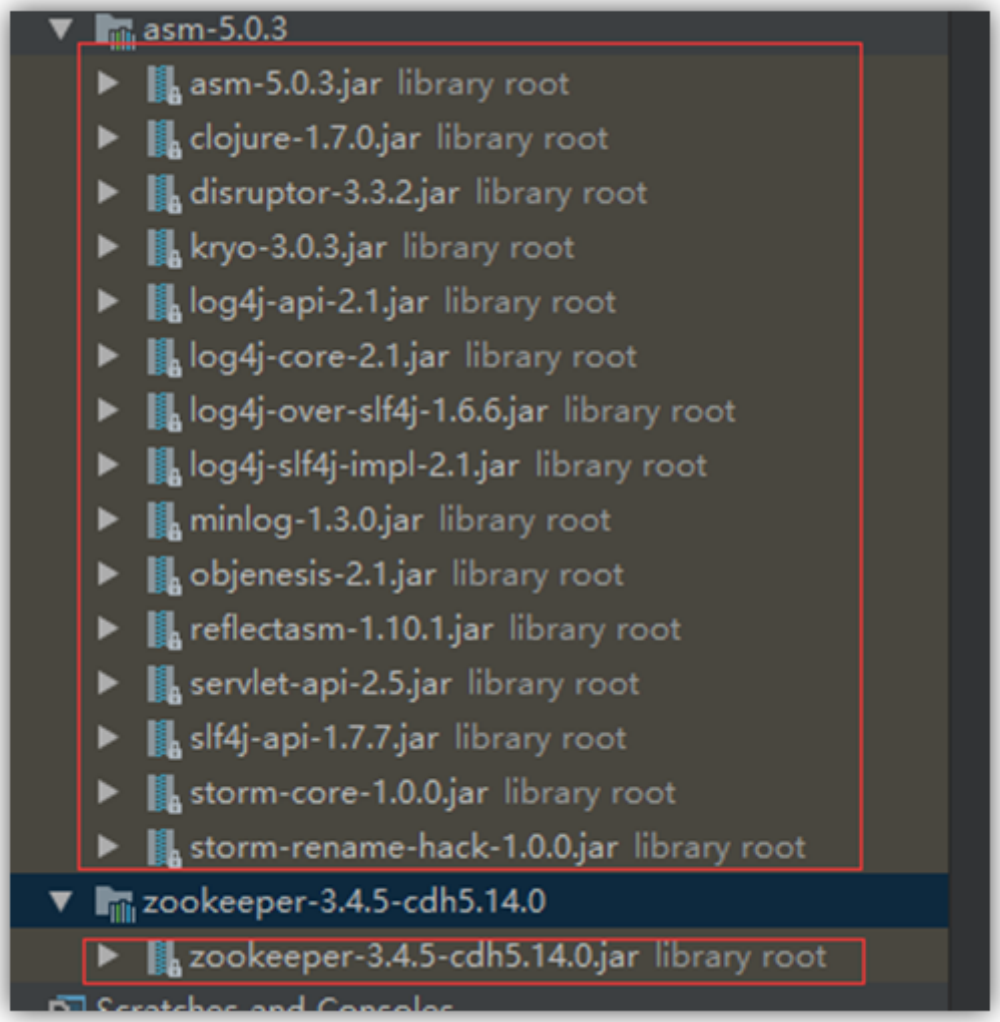
1）apache-storm-1.0.0.tar.gz压缩包中的lib文件夹下所有的jar



2）zookeeper的jar

zookeeper-3.4.5-cdh5.14.0.tar.gz压缩包中的zookeeper-3.4.5-cdh5.14.0.jar

3）项目中导入以上的jar



3.2 编写代码

3.2.1 编写spout读取文本文档

spout需要实现IRichSpout接口，实现以下的方法

public void open(Map map, TopologyContext context, SpoutOutputCollector collector)	在运行spout实例时会先运行open方法，是用来做初始化的，在这里要将SpoutOutputCollector spoutOutputCollector对象保存下来，供后面的 nextTuple()接口使用，还可以执行一些业务逻辑的初始化操作。
public void nextTuple()	具体的读取数据源的方法，当该方法被调用时，要求SpoutOutputCollector发射tuple。
public void declareOutputFields(OutputFieldsDeclarer declarer)	在这个接口里，我们要声明本Spout要发射（emit）的数据的结构，即一个backtype.storm.tuple.Fields对象。这个对象和public void nextTuple()接口中emit的backtype.storm.tuple.Values共同组成了一个元组对象（backtype.storm.tuple.Tuple），供后面接收该数据的Blot使用
public void close()	topology终止时，执行此方法
public void activate()	当thread运行完spout实例的open()方法之后，该spout实例处于deactivated（失效）模式，过段时间会变成activated（激活）模式，此时会调用Spout实例的activate()方法
public void deactivate()	在Topology运行过程中，通过客户端执行deactivate命令，禁用指定的Topology时，被禁用的Topology的spout实例会变成deactivate（失效），并且会调用spout实例deactivate()方法
public void ack(Object msgId)	从此spout发射的带有messageID的tuple处理成功时调用此方法。
public void fail(Object msgId)	从此spout发射的带有messageID的tuple处理失败时调用此方法。
Map<String, Object> getComponentConfiguration()	运行TopologyBuilder的createTopology()时调用此方法，用于输出特定于Spout和Bolt实例的配置参数值对，此方法用于声明针对当前组件的特殊的Configuration配置

实现代码如下：

```
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;

import backtype.storm.Config;
import backtype.storm.spout.SpoutOutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.IRichSpout;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Values;

/**
 * 读取txt文档，按行发射给bolt
 * @author Administrator
 *
```

```
 */
public class WordReader implements IRichSpout{

	/**
	 * 序列化
	 */
	private static final long serialVersionUID = 1L;


	private FileInputStream is;
	private InputStreamReader isr;
	private BufferedReader br;
	private String line = "";
	private SpoutOutputCollector collector;

	/**
	 * 此方法用于声明当前Spout的Tuple发送流的域名字，即一个backtype.storm.tuple.Fields对象。
	 * 这个对象和public void nextTuple()接口中emit的backtype.storm.tuple.Values
	 * 共同组成了一个元组对象（backtype.storm.tuple.Tuple）
	 * 供后面接收该数据的Blot使用
	 * 运行TopologyBuilder的createTopology()时调用此方法
	 */
	@Override
	public void declareOutputFields(OutputFieldsDeclarer declarer) {
		declarer.declare(new Fields("line"));
	}

	/**
	 * 运行TopologyBuilder的createTopology()时调用此方法
	 * 用于输出特定于Spout和Bolt实例的配置参数值对
	 * 此方法用于声明针对当前组件的特殊的Configuration配置，在需要的情况下会进行配置。
	 */
```

```
    @Override
    public Map<String, Object> getComponentConfiguration() {
        //设置Topology中当前组件的线程数量上限为3
//        Map<String, Object> ret = new HashMap<String, Object>();
//      ret.put(Config.TOPOLOGY_MAX_TASK_PARALLELISM, 3);
//      return ret;
        return null;
    }

    /**
     * 当一个Task被初始化的时候会调用此open方法。
     * 在这里要将SpoutOutputCollector spoutOutputCollector对象保存下来，
     * 供后面的public void nextTuple()接口使用，还可以执行一些其他的操作。
     * 例如这里将txt文档转换成流，也就是初始化操作。
     * 里面接收了三个参数，第一个是创建Topology时的配置，
     * 第二个是所有的Topology数据，第三个是用来把Spout的数据发射给bolt
     */
    @Override
    public void open(Map conf, TopologyContext context,
            SpoutOutputCollector collector) {
        try {
            this.collector = collector;
            this.is = new FileInputStream("E:\\words.txt");
            this.isr = new InputStreamReader(is,"utf-8");
            this.br = new BufferedReader(isr);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /**
     * 当thread运行完spout实例的open()方法之后，
     * 该spout实例处于deactivated（失效）模式，
     * 过段时间会变成activated（激活）模式，此时会调用Spout实例的activate()
方法
```

```
     */
    @Override
    public void activate() {

    }

    /**
     * 接口该接口实现具体的读取数据源的方法。
     * 当该方法被调用时，要求SpoutOutputCollector喷射tuple。
     * 在本例中，nextTuple()方法是负责对txt文档转换成的流进行逐行读取。
     * 将获取的数据emit（发射）出去。
     * emit的对象是通过public Values createValues(String line)方法
     * 生成的backtype.storm.tuple.Values对象。
     * 该方法从数据源的一行数据中，选取的若干个目标值组成一个
backtype.storm.tuple.Values对象。
     * 这个对象中可以存储不同类型的对象，例如你可以同时将String对象，
     * Long对象存取在一个backtype.storm.tuple.Values中emit出去。
     * 实际上只要实现了Storm要求的序列化接口的对象都可以存储在里面。
     * emit该值得时候需要注意，
     * 他的内容要和declareOutputFields中声明的backtype.storm.tuple.Fields
对象相匹配，必须一一对应。
     * 他们被共同组成一个backtype.storm.tuple.Tuple元组对象，被后面接收该
数据流的对象使用。
     */
    @Override
    public void nextTuple() {
        try {
            while ((this.line = this.br.readLine()) != null) {
                //数据过滤

                //发射数据
                this.collector.emit(new
Values(this.line),UUID.randomUUID());
            }
        } catch (Exception e) {
```

```
            e.printStackTrace();
        }
    }

    /**
     * 从此spout发射的带有messageID的tuple处理成功时调用此方法。
     * 该方法的一个典型实现是把消息从队列中移走，避免被再次处理。
     */
    @Override
    public void ack(Object msgId) {
        System.out.println("OK:" + msgId);
    }

    /**
     * 从此spout发射的带有messageID的tuple处理失败时调用此方法。
     */
    @Override
    public void fail(Object msgId) {
        System.out.println("FAIL:" + msgId);
    }

    /**
     * topology终止时，执行此方法
     * 在本例子中，可以在这个阶段释放资源
     */
    @Override
    public void close() {
        try {
            this.br.close();
            this.isr.close();
            this.is.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

```
        /**
         * 在Topology运行过程中，通过客户端执行deactivate命令，
         * 禁用指定的Topology时，被禁用的Topology的spout实例会变成
deactivate（失效），
         * 并且会调用spout实例deactivate()方法
         */
        @Override
        public void deactivate() {

        }
    }
```

3.2.2 编写bolt解析spout发射的每行数据

bolt需要实现IRichBolt接口，实现以下的方法

public void prepare(Map stormConf, TopologyContext context, OutputCollector collector)	当这个组件的task在集群中的一台worker内被初始化的时候，该函数被调用。它向bolt提供了该bolt执行的环境(里面接收了三个参数，第一个是创建Topology时的配置，第二个是所有的Topology数据，第三个是用来把Bolt的数据发射给下一个bolt)在这里要将OutputCollector collector对象保存下来
public void execute(Tuple input)	处理输入的一个单一tuple。每次接收到元组时都会被调用一次，还会再发布若干个元组
public void cleanup()	topology终止时，执行此方法。 注意： 由于cleanup方法并不可靠，它只在local mode下生效，Storm集群模式下cleanup不会被调用执行。很多资源得不到释放。所以，在kill topology之前，先deactivate相应的topology。在spout中实现deactivate()方法，deactivate()方法中给bolt emit特殊的数据(如：emit “shutDown”字符串给bolt)，bolt中判断接收的数据为”shutDown”就调用cleanup()方法。在cleanup()方法中释放需要释放的资源。
public void declareOutputFields(OutputFieldsDeclarer declarer)	此方法用于声明当前bolt的Tuple发送流的域名字，即一个backtype.storm.tuple.Fields对象。这个对象和public void execute(Tuple input)接口中emit的backtype.storm.tuple.Values共同组成了一个元组对象（backtype.storm.tuple.Tuple）供后面接收该数据的Blot使用
public Map<String, Object> getComponentConfiguration()	用于输出特定于Spout和Bolt实例的配置参数值对，此方法用于声明针对当前组件的特殊的Configuration配置，在需要的情况下会进行配置。

实现代码如下：

```
import java.util.Map;

import backtype.storm.task.OutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.IRichBolt;
import backtype.storm.topology.OutputFieldsDeclarer;
```

```
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Values;

/**
 * 解析spout发射的tuple，拆分成一个个单词发射给下一个bolt
 * @author Administrator
 *
 */
public class WordNormalizer implements IRichBolt{

	/**
	 * 序列化
	 */
	private static final long serialVersionUID = 1L;

	private OutputCollector collector;
	private String line;
	private String[] words;

	/**
	 * 当这个组件的task在集群中的一台worker内被初始化的时候，该函数被调用。
	 * 它向bolt提供了该bolt执行的环境
	 * (
	 *     里面接收了三个参数，
	 *     第一个是创建Topology时的配置，
	 *     第二个是所有的Topology数据，
	 *     第三个是用来把Bolt的数据发射给下一个bolt
	 * )
	 * 在这里要将OutputCollector collector对象保存下来
	 */
	@Override
	public void prepare(Map stormConf, TopologyContext context,
			OutputCollector collector) {
		this.collector = collector;
```

```
    }

    /**
     * 处理输入的一个单一tuple。
     * 每次接收到元组时都会被调用一次，还会再发布若干个元组
     */
    @Override
    public void execute(Tuple input) {
//          line = (String)input.getValue(0);
            line = (String)input.getValueByField("line");//与上面等价
            words = line.split("\t");
            for(String word : words){
                collector.emit(new Values(word));
            }
            //成功，提示从此spout喷出的带有messageID的tuple已被完全处理，把消息从队列中移走，避免被再次处理。
            this.collector.ack(input);
    }

    /**
     * Topology执行完毕的清理工作，比如关闭连接、释放资源等操作都会写在这里
     * topology终止时，执行此方法
     * 注意：
     *     由于cleanup方法并不可靠，它只在local mode下生效，Storm集群模式下cleanup不会被调用执行。很多资源得不到释放
     *     所以，在kill topology之前，先deactivate相应的topology。
     *     bolt中判断接收的数据为”shutDown”就调用cleanup()方法。在cleanup()方法中释放需要释放的资源。
     */
    @Override
    public void cleanup() {

    }

    /**
```

```
     * 此方法用于声明当前bolt的Tuple发送流的域名字，即一个
backtype.storm.tuple.Fields对象。
     * 这个对象和public void execute(Tuple input)接口中emit的
backtype.storm.tuple.Values
     * 共同组成了一个元组对象（backtype.storm.tuple.Tuple）
     * 供后面接收该数据的Blot使用
     */
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }

    /**
     * 用于输出特定于Spout和Bolt实例的配置参数值对
     * 此方法用于声明针对当前组件的特殊的Configuration配置，在需要的情况下会进
行配置。
     */
    @Override
    public Map<String, Object> getComponentConfiguration() {
        return null;
    }

}
```

3.2.3 编写bolt对上一个bolt发射的单词进行计数

bolt同样实现IRichBolt

具体代码如下：

```
import java.util.HashMap;
import java.util.Map;

import backtype.storm.task.OutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.IRichBolt;
```

```
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.tuple.Tuple;

/**
 * 接收WordNormalizer发射的单词，进行计数
 * @author Administrator
 *
 */
public class WordCounter implements IRichBolt{

	/**
	 * 序列化
	 */
	private static final long serialVersionUID = 1L;

	Integer id;
   String name;
   Map<String, Integer> counters;
   private OutputCollector collector;

	@Override
	public void prepare(Map stormConf, TopologyContext context,
			OutputCollector collector) {
		this.counters = new HashMap<String, Integer>();
	this.collector = collector;
	this.name = context.getThisComponentId();
	this.id = context.getThisTaskId();
	}

	@Override
	public void execute(Tuple input) {
		String word = input.getString(0);
		//计数
		if(!counters.containsKey(word)){
			counters.put(word, 1);
```

```
            }else{
                Integer c = counters.get(word) + 1;
              counters.put(word, c);
            }

            //成功，提示从此spout喷出的带有messageID的tuple已被完全处理，把消息从队列中移走，避免被再次处理。
            this.collector.ack(input);
        }

        /**
         * Topology执行完毕的清理工作，比如关闭连接、释放资源等操作都会写在这里
         * topology终止时，执行此方法
         * 注意:
         *    由于cleanup方法并不可靠，它只在local mode下生效，Storm集群模式下cleanup不会被调用执行。很多资源得不到释放
         *    所以，在kill topology之前，先deactivate相应的topology。
         *    bolt中判断接收的数据为”shutDown”就调用cleanup()方法。在cleanup()方法中释放需要释放的资源。
         *
         * 因为这只是个Demo，我们用它来打印我们的计数器
         */
        @Override
        public void cleanup() {
            System.out.println("-- Word Counter [" + name + "-" + id + "] --");
            for(Map.Entry<String, Integer> entry : counters.entrySet()){
                System.out.println(entry.getKey() + ": " + entry.getValue());
            }
            counters.clear();
        }

        @Override
        public void declareOutputFields(OutputFieldsDeclarer declarer) {
```

```
            //不再向外发射数据，此处不写代码
        }

        @Override
        public Map<String, Object> getComponentConfiguration() {
            return null;
        }

    }
```

3.2.4 编写storm执行主类（本地模式）

在执行主类中对Topology进行设置，包括spout、bolt、分组、配置、运行模式的设置

具体代码实现：

```
import com.storm.bolt.WordCounter;
import com.storm.bolt.WordNormalizer;
import com.storm.spout.WordReader;

import backtype.storm.Config;
import backtype.storm.LocalCluster;
import backtype.storm.topology.TopologyBuilder;
import backtype.storm.tuple.Fields;
import backtype.storm.utils.Utils;

/**
 * storm执行主类
 * @author Administrator
 *
 */
public class WordCountTopologyMain {
    public static void main(String[] args) {
        //定义一个Topology
        TopologyBuilder builder = new TopologyBuilder();
```

```
            //第3个参数设置并行度
//          builder.setSpout("spout", new WordReader(), 1);
            //2个参数，默认并行度为
            builder.setSpout("word-reader",new WordReader());

            builder.setBolt("word-normalizer", new
WordNormalizer()).shuffleGrouping("word-reader");

            builder.setBolt("word-counter", new
WordCounter(),2).fieldsGrouping("word-normalizer", new Fields("word"));

            //配置
            Config conf = new Config();
            conf.put("wordsFile", "E:/test.txt");
            conf.setDebug(false);

            //本地提交模式，例如eclipse执行main方法
            LocalCluster localCluster = new LocalCluster();
            localCluster.submitTopology("mytopology", conf,
builder.createTopology());

            Utils.sleep(15000);

            //停止本地运行
            localCluster.shutdown();
        }
}
```

3.3 本地模式运行

直接在开发工具上运行执行主类的main方法，我这里是使用eclipse运行WordCountTopologyMain的main方法，执行结果如下：

```
-- Word Counter [word-counter-2] --
storm: 5
spark: 3
hadoop: 5
16633 [main] INFO  backtype.storm.daemon.executor - Shut down executor word-counter:[2 2]
16633 [main] INFO  backtype.storm.daemon.executor - Shutting down executor word-counter:[3 3]
16634 [Thread-10-disruptor-executor[3 3]-send-queue] INFO  backtype.storm.util - Async loop interrupted
16634 [Thread-11-word-counter] INFO  backtype.storm.util - Async loop interrupted!
-- Word Counter [word-counter-3] --
hive: 1
zookeeper: 1
16635 [main] INFO  backtype.storm.daemon.executor - Shut down executor word-counter:[3 3]
```