Storm 是什么

Storm 是一个分布式实时大数据处理系统，可以帮助我们方便的处理海量数据，具有高可靠、高容错、高扩展的特点

Storm 是流式框架，有很高的数据吞吐能力，Strom 本身是无状态的，通过 ZooKeeper 管理分布式集群环境和集群状态

Strom 的安装和使用都很简单，但功能强大，可以并行的对实时数据流进行各种处理

应用场景

应用 Storm 的场景例如：

- 日志处理

监控系统中的事件日志，使用storm检查每条日志信息，把符合匹配规则的消息保存到数据库

- 电商商品推荐

后台需要维护每个用户的兴趣点，主要基于用户的历史行为、查询、点击、地理信息等信息获得，其中有很多实时数据，可以使用 Storm 进行处理，在此基础上进行精准的商品推荐和放置广告

Storm与Hadoop的关系

Storm 与 Hadoop 都用来处理大数据，那么他们的关系是怎样的呢?

Hadoop 是强大的大数据处理系统，但是在实时计算方面不够擅长

Storm 的核心功能就是提供强大的实时处理能力，但没有涉及存储

所以 Storm 与 Hadoop 即不同也互补

他俩的最主要的区别例如：

1. Storm 是实时流处理模式，Hadoop 是批处理模式

2. Storm 就像一条川流不息的河流，只要不是意外或者人为停止，他就会一直运行，Hadoop 是在需要时执行 MapReduce 任务，执行完成后停止
3. 在处理时间上，Storm 每秒可以处理数万条消息，HDFS+MapReduce 处理大量数据时通常需要几分钟到几小时

Storm 怎么用

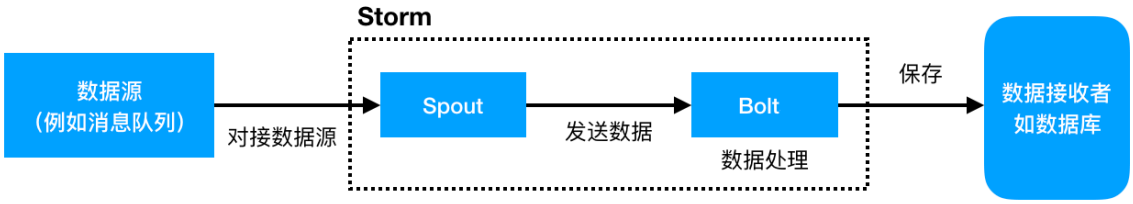
Storm 非常简洁，为了便于理解，先看2个最核心的概念：

1. 数据源头 Spout
2. 数据处理单元 Bolt

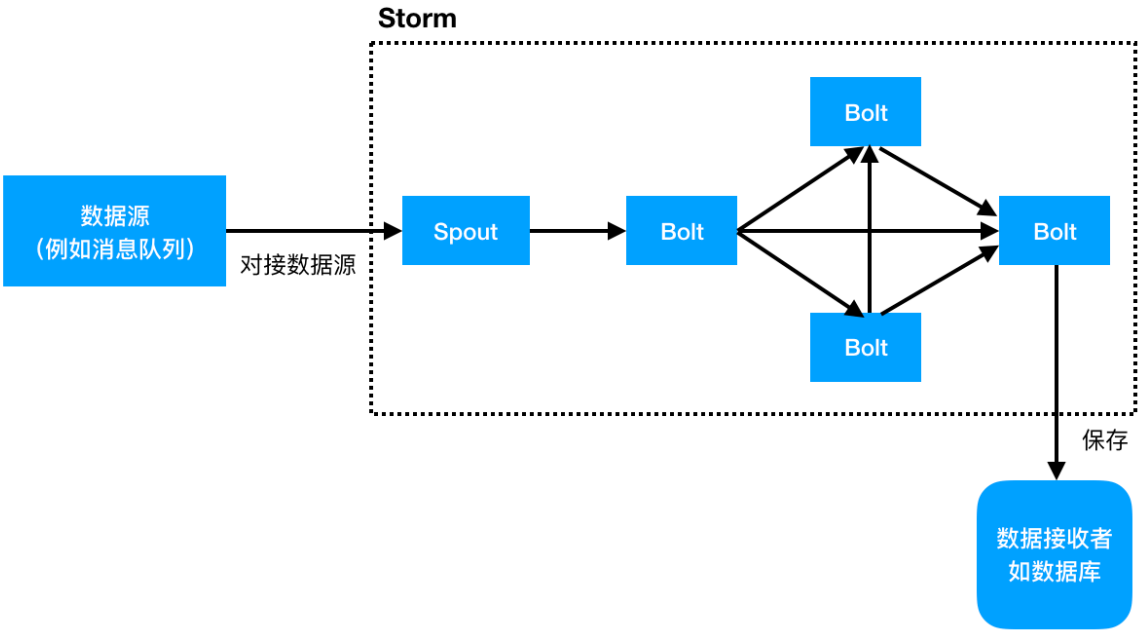
先把 Spout 与外部数据进行对接，这样就可以把外面的数据量引到 Storm 中了

Spout 接收到数据后就会发给 Bolt，这就需要告诉 Bolt 如何处理，处理完成后把数据放到哪儿，例如数据库

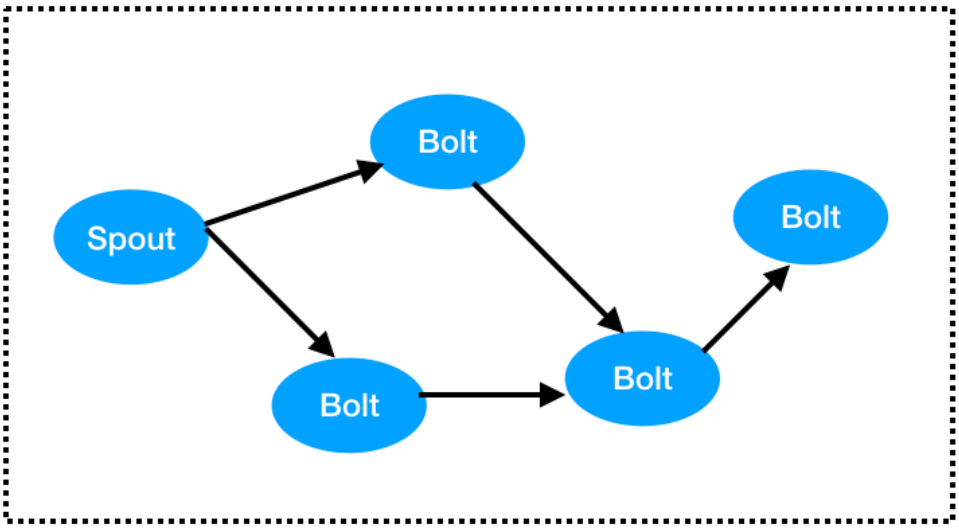
这就是一个最简单的模型



而其中 Bolt 可以有多个，这就使得 Storm 强大而灵活



可以看出 Storm 就是一个有拓扑图，点是 spout 或者 bolt，边是数据流向



所以，使用 storm 时需要做的就是把整个拓扑图构造出来：

1. 定义数据从哪儿来
2. 定义数据流向和处理单元的逻辑
3. 定义数据到哪儿去

示例1 统计单词出现的次数

以大数据中的helloworld “词频统计” 为例来学习 Storm 的开发方法

实现思路

构造拓扑结构：

1. spout 读取一行文本，向下游发送
2. 把句子分割为单词的 Bolt，从 spout 发出的数据中取得一行文本，然后分割为一个个的单词，把每个单词向下游发送
3. 单词计数的 Bolt，接收分词 Bolt 发出的单词，对每个单词出现的次数进行统计，然后把单词及其次数作为一个数据单元向下游发送
4. 结果输出的 Bolt，接收计数 Bolt 发出的数据单元，取出单词及其次数进行汇总，最后输出统计结果
5. 创建拓扑对象，把 spout 和各个 bolt 连接起来，然后提交到 storm 中执行

代码实现

创建项目目录 `storm-wordcount`

项目目录下新建maven工程文件 `pom.xml`

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="h
ttp://www.w3.org/2001/XMLSchema-instance"
    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.storm.test</groupId>
    <artifactId>wordcount</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>wordcount</name>
    <url>http://maven.apache.org</url>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sour
ceEncoding>
    </properties>

    <dependencies>
    <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.6.1</version>
```

```
        </dependency>
        <dependency>
            <groupId>org.apache.storm</groupId>
            <artifactId>storm-core</artifactId>
            <version>1.0.2</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

项目根目录下执行maven命令，安装依赖

```
mvn package
mvn dependency:tree
```

项目根目录下创建源码目录 `src/main/java`

然后在 java 目录下创建包目录 `com/storm/test`

现在项目的目录结构如下：

```
├── pom.xml
└── src
    └── main
        └── java
            └── com
                └── storm
                    └── test
```

具体代码如下：

SpoutWords.java

```
package com.storm.test;

import java.util.Map;

import org.apache.storm.spout.SpoutOutputCollector;
```

```java
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;

public class SpoutWords extends BaseRichSpout{

    private SpoutOutputCollector spoutOutputCollector;

    //为了简单,定义一个静态数据模拟不断的数据流产生
    private static final String[] sentences={
            "The core abstraction in Storm is the stream",
            "The basic primitives Storm provides for doing stre
am transformations ",
            "Spouts and bolts have interfaces that you implemen
t to run your application-specific logic"
    };

    private int index=0;

    //初始化操作
    public void open(Map map, TopologyContext topologyContext,
SpoutOutputCollector spoutOutputCollector) {
        this.spoutOutputCollector = spoutOutputCollector;
    }

    //核心逻辑
    public void nextTuple() {
        // 发射数据，Values 需要与 Fields 对应
        spoutOutputCollector.emit(new Values(sentences[index]))
;

        // 控制循环读取数组数据
        ++index;
        if(index>=sentences.length){
            index=0;
        }
    }

    //定义向下游输出数据的字段
    public void declareOutputFields(OutputFieldsDeclarer output
FieldsDeclarer) {
        // 定义发射数据都有哪些字段，下游 bolt 会根据字段名获取数据
        outputFieldsDeclarer.declare(new Fields("sentences"));
    }
}
```

BoltSplit.java

```
package com.storm.test;

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;

public class BoltSplit extends BaseRichBolt{

    private OutputCollector outputCollector;

    // 初始化
    public void prepare(Map map, TopologyContext topologyContex
t, OutputCollector outputCollector) {
        this.outputCollector = outputCollector;
    }

    // 每次收到数据都会执行此方法
    public void execute(Tuple tuple) {
        // 根据上游设置的字段名读取数据
        String sentence = tuple.getStringByField("sentences");

        // 分词
        String[] words = sentence.split(" ");
        for(String word : words){
            // 把每个单词发射出去
            outputCollector.emit(new Values(word));
        }
    }

    //定义向下游输出数据的字段
    public void declareOutputFields(OutputFieldsDeclarer output
FieldsDeclarer) {
        outputFieldsDeclarer.declare(new Fields("word"));
    }
}
```

BoltCount.java

```
package com.storm.test;

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.Fields;
import org.apache.storm.tuple.Tuple;
import org.apache.storm.tuple.Values;

public class BoltCount extends BaseRichBolt{

    //保存单词计数
    private Map<String,Long> wordCount = null;

    private OutputCollector outputCollector;

    // 初始化
    public void prepare(Map map, TopologyContext topologyContex
t, OutputCollector outputCollector) {
        this.outputCollector = outputCollector;
        wordCount = new HashMap<String, Long>();
    }

    // 每次收到数据都会执行此方法
    public void execute(Tuple tuple) {
        // 获取上游数据
        String word = tuple.getStringByField("word");

        // 计数
        Long count = wordCount.get(word);
        if(count == null){
            count = 0L;
        }
        ++count;
        wordCount.put(word,count);

        // 向下游发射此单词目前的统计结果
        outputCollector.emit(new Values(word,count));
    }
```

```
    //定义向下游输出数据的字段
    public void declareOutputFields(OutputFieldsDeclarer output
FieldsDeclarer) {
        outputFieldsDeclarer.declare(new Fields("word","count")
);
    }
}
```

BoltReport.java

```
package com.storm.test;

import java.util.HashMap;
import java.util.Map;
import java.util.Set;

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;

public class BoltReport extends BaseRichBolt {

    private Map<String, Long> counts = null;

    // 初始化
    public void prepare(Map map, TopologyContext topologyContex
t, OutputCollector outputCollector) {
        counts = new HashMap<String, Long>();
    }

    // 每次收到数据都会执行此方法
    public void execute(Tuple tuple) {
        // 获取上游数据
        String word = tuple.getStringByField("word");
        Long count = tuple.getLongByField("count");

        // 保存最新计数
        counts.put(word, count);

        //打印更新后的结果
        printReport();
    }

    public void declareOutputFields(OutputFieldsDeclarer output
```

```
FieldsDeclarer) {
        //无下游输出,不需要代码
    }

    // 输出整体技术信息
    private void printReport(){
        System.out.println("---------begin------");
        Set<String> words = counts.keySet();
        for(String word : words){
            System.out.println(word + " --> " + counts.get(word
));
        }
        System.out.println("---------end---------");
    }
}
```

WordCountTopology.java

```
package com.storm.test;

import org.apache.storm.Config;
import org.apache.storm.LocalCluster;
import org.apache.storm.topology.TopologyBuilder;
import org.apache.storm.tuple.Fields;

public class WordCountTopology {

    public static void main(String[] args) {

        //创建 spout 和 blot 的实例
        SpoutWords sentenceSpout = new SpoutWords();
        BoltSplit splitSentenceBolt = new BoltSplit();
        BoltCount wordCountBolt = new BoltCount();
        BoltReport reportBolt = new BoltReport();

        //拓扑Builder
        TopologyBuilder topologyBuilder = new TopologyBuilder()
;

        //配置 spout
        topologyBuilder.setSpout("spout", sentenceSpout, 2);

        //配置 split bolt,上游为 spout
        topologyBuilder.setBolt("bolt-split", splitSentenceBolt
).shuffleGrouping("spout");
```

```
        //配置 count bolt,上游为 bolt-split
        topologyBuilder.setBolt("bolt-count", wordCountBolt).fi
eldsGrouping("bolt-split", new Fields("word"));

        //配置 report bolt,上游为 bolt-count
        topologyBuilder.setBolt("bolt-report", reportBolt).glob
alGrouping("bolt-count");

        Config config = new Config();

        //建立本地集群,利用LocalCluster,storm在程序启动时会在本地自动
建立一个集群,不需要用户自己再搭建,方便本地开发和debug
        LocalCluster cluster = new LocalCluster();

        //创建拓扑实例,并提交到本地集群进行运行
        cluster.submitTopology("word-count", config, topologyBu
ilder.createTopology());
    }
}
```

编译运行

```
mvn compile

mvn exec:java -Dexec.mainClass="com.storm.test.WordCountTopolog
y" -Dexec.cleanupDaemonThreads=false
```

执行后会输出大量日志信息，之后会循环输出单词统计信息，例如：

```
Storm --> 188
doing --> 94
for --> 94
run --> 93
your --> 93
The --> 188
that --> 93
primitives --> 94
...
```

核心概念

1. Spouts - 数据流的源头

Storm 从外部接收数据，例如 Twitter Streaming API、Kafka ……，通过 Spouts 从这些数据源读取数据

2. Bolts - 逻辑处理单元

Spouts 向 Bolts 传递数据，Bolts 接收数据进行处理操作，然后把结果再发射出去

Bolts 的常见操作例如：过滤、聚合、连接、与数据源数据库交互

3. Tuple - 数据单元

Storm 数据流中的数据单元，例如水流中是一滴滴的水滴，Storm 数据流中流淌的就是一个个的 tuple，里面包含着数据

4. Stream - 数据流

是一个无序的 Tuple 序列

5. Topology - 拓扑

Spouts 和 Bolts 连接起来之后形成了 Topology，其中定义了整个应用的实时处理逻辑

Topology 是一个有向图，定点是计算单元，边是数据流

Topology 始于 Spout，Spout 向一个或者多个 Bolt 发射数据，Bolt 拥有处理逻辑，Bolt 的输出可以发射给其他 Bolt 作为他们的输入

Storm 会保持 Topology 一直运行，除非杀掉他

6. Tasks - 任务

一个 task 就是一个 Spout 的执行，或者一个 Bolt 的执行

7. Workers - 工作进程

worker 负责实际运行 task

topology 运行在一个分布式环境中的多个工作节点上，Storm 会把 tasks 均匀的分布在所有worker上

每个 worker 都是一个物理JVM，执行着 Topology 中所有 task 的一个子集

8. Stream Grouping - 流分组

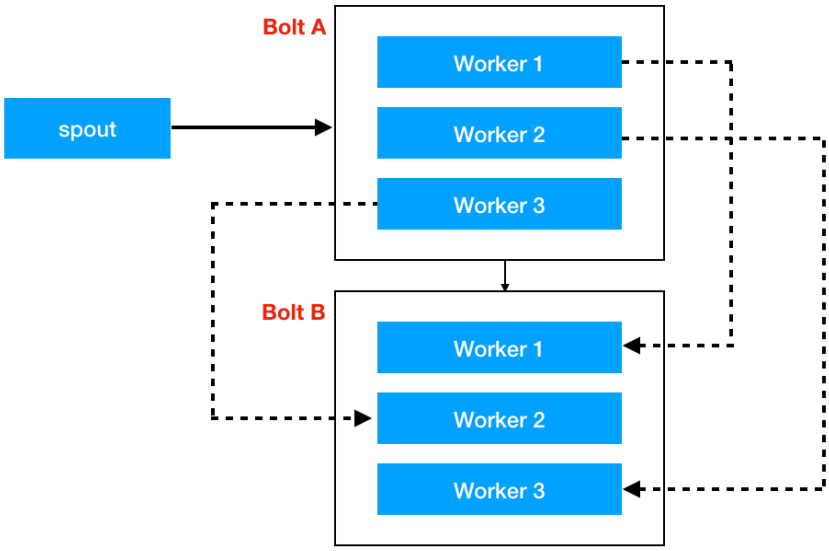
简单理解就是控制 Tuple 的路由，定义 Tuple 在 Topology 中如何流动

每个 Spout 或 Bolt 都会在集群中执行多个任务，每个任务都对应为一个线程的执行，Stream Grouping 定义的就是如何从一个 task集合 向其他 task集合 发送 tuple

现在 Storm 中已经有 8 种分组策略，下面看下其中 4 种常用的：

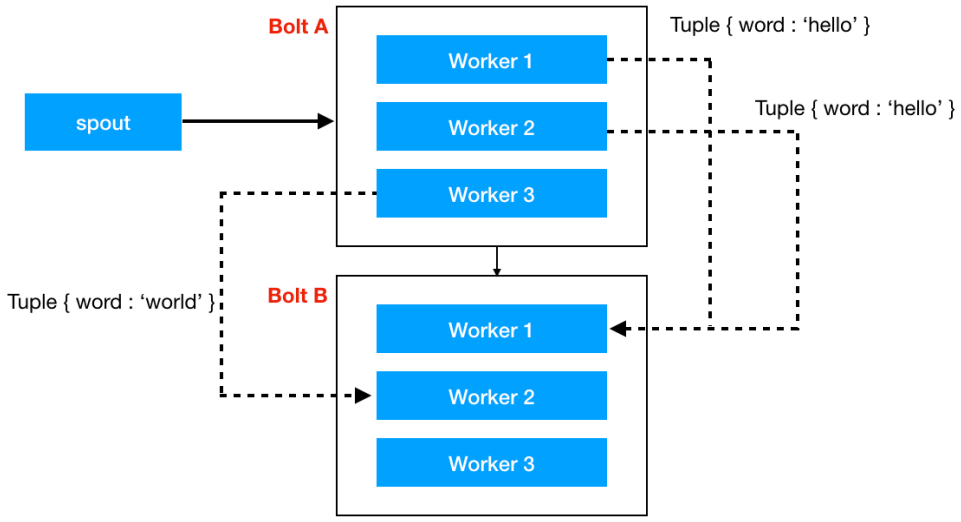
（1）Shuffle Grouping

随机分配，可以让每个 bolt 获得数量均衡的 tuple



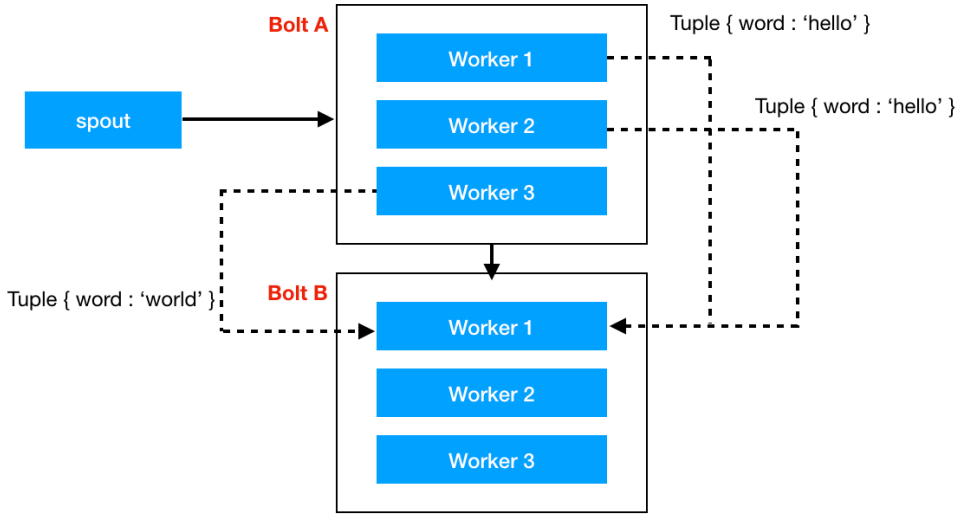
（2）Field Grouping

field 名字相同的 tuples 会被组织在一起，例如，如果根据 "user-id" 这个 field 分组，相同 "user-id" 的 tuples 将总是去向同一个 task，其他的 tuples 则去向不同的 task



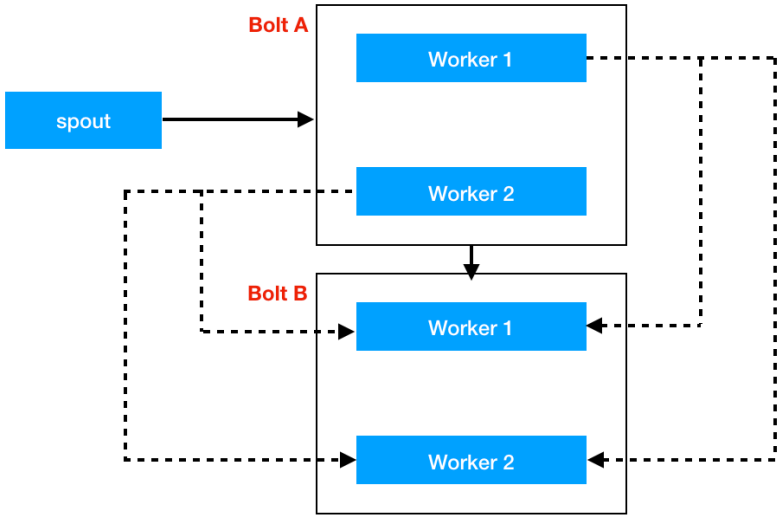
（3）Global Grouping

全局统一分组，所有数据流都流向同一个 Bolt



（4）All Grouping

向每个实例都发送一次，主要用于发送信号



回顾一下 wordcount 示例中使用的 grouping 方法

```
......

//配置 spout
topologyBuilder.setSpout("spout", sentenceSpout, 2);

//配置 split bolt,上游为 spout
topologyBuilder.setBolt("bolt-split", splitSentenceBolt).shuffl
eGrouping("spout");

//配置 count bolt,上游为 bolt-split
topologyBuilder.setBolt("bolt-count", wordCountBolt).fieldsGrou
ping("bolt-split", new Fields("word"));

//配置 report bolt,上游为 bolt-count
topologyBuilder.setBolt("bolt-report", reportBolt).globalGroupi
ng("bolt-count");

......
```

设置第一个 Bolt `bolt-split` 时使用的是 `shuffleGrouping`，因为 spout 发送一行文字，给谁都行，不关心分组，所以使用 shuffleGrouping 随机即可

设置第二个 Bolt `bolt-count` 时使用的是 `fieldsGrouping`，因为 `bolt-split` 是按单词发射的，所以需要让同一个单词被同一个task处理，就要使用按字段分组方式

设置第三个 Bolt `bolt-report` 时使用的是 `globalGrouping` 统一分组，因为到这儿就要汇总了，需要接收所有的统计结果

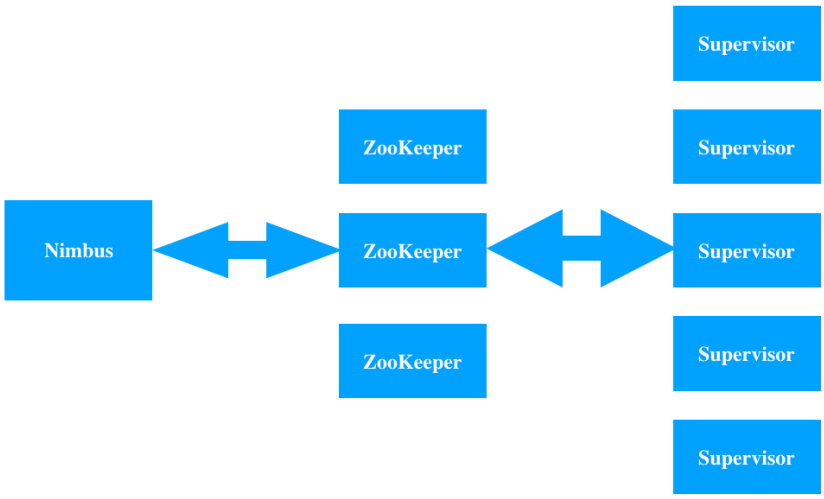
集群架构

Storm cluster 中有2种类型节点：master node、worker nodes

master node 中运行着一个守护进程，名为 **Nimbus**，负责向集群中分布代码、分配任务、监控失败状况

每个 worker node 中运行着一个守护进程，名为 **Supervisor**，负责接收工作任务，根据 Nimbus 的指令来启动或者停止工作进程

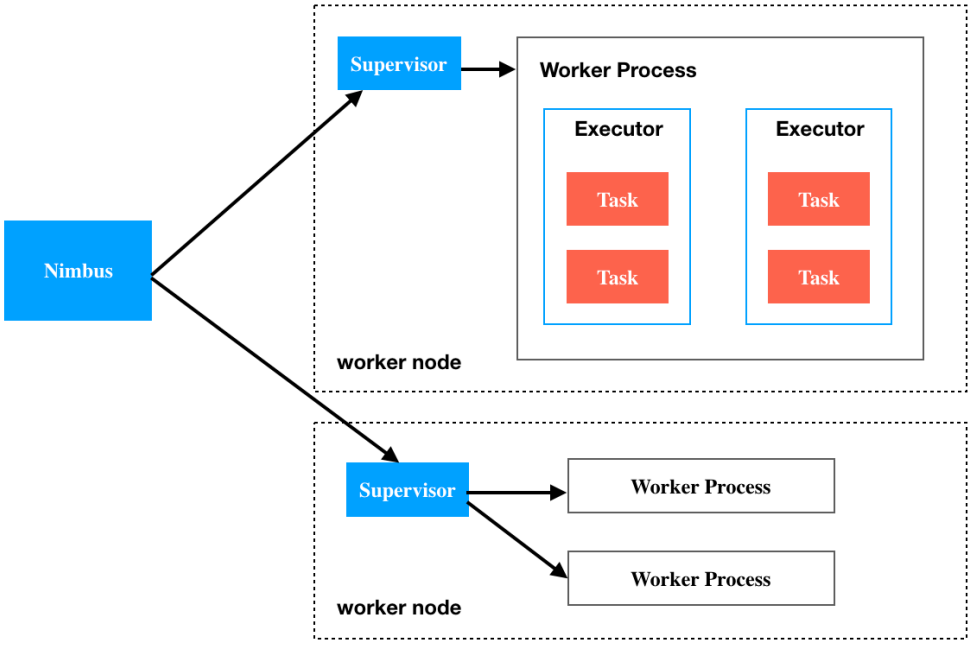
每个工作进程执行一个拓扑的子集，一个拓扑包含多个工作进程，这些工作进程散布在集群里的多台机器中



Nimbus 与 Supervisors 的协作是通过 zookeeper集群 完成的，Nimbus 与 Supervisors 都是无状态的，状态信息保存在 zookeeper 或者本地磁盘中

Nimbus 与 Supervisors 具有高可靠性，即使通过 `kill -9` 杀掉他们，也会快速重新启动起来，这使得 Storm cluster 极其稳定

上图是 Storm cluster 的全景图，下面我们看一下细节



概念总结：

- Nimbus - Storm cluster 的主节点，其他节点都是工作节点 worker node，主节点负责任务分配、监控失败等管理工作
- Supervisor - 负责接受 Nimbus 分配的任务，一个 Supervisor 会有多个工作进程 worker process，并管理这些工作进程来完成接收到的任务
- Worker process - 工作进程，一个工作进程会执行某个特定 topology 的相关任务，它并不直接自己执行任务，而是创建 `executors` 来执行，一个工作进程可以有多个 executor
- Executor - 执行器，就是工作进程创建的一个线程，一个执行器会执行一个或多个任务
- Task - 执行实际的数据处理，也就是一个 spout 或 bolt

简单总结一下：

1. 一个 Storm cluster 中有一个 `Nimbus` 和多个 `Supervisor`
2. 一个 `Supervisor` 下有多个 `Worker process`
3. 一个 `Worker process` 有多个 `Executor`
4. 一个 `Executor` 下有多个 `Task`

示例2 统计通话记录

需求描述

处理通话记录，统计相同呼叫人和被呼叫人的通话次数、通话总时长

例如通话记录：

```
# 呼叫者号码,接收者号码,持续时长
1234123402, 1234123401,20
...
```

统计结果例如：

```
1234123402-1234123401 : 87,2523
1234123402-1234123404 : 95,2919
...
```

实现思路

构造拓扑图：

1. spout 读取通话日志，发送给 bolt，数据包括 `from, to, duration`
2. bolt1 接收 spout 的数据，重新组织数据格式为 `from-to, duration`
3. bolt2 接收 bolt1 的数据，对 `from-to` 相同的数据进行汇总，累计次数、总时长，在最后输出统计结果
4. topology 中连接 spout 与 bolt1、bolt2，并提交 storm 执行

具体实现

（1）创建项目目录

在合适的位置创建项目目录 `storm-mobile`

（2）新建 pom.xml

项目根目录下新建maven工程文件 pom.xml，内容：

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="h
ttp://www.w3.org/2001/XMLSchema-instance"
    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.storm.test</groupId>
    <artifactId>mobile</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>mobile</name>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sour
ceEncoding>
    </properties>

    <dependencies>
    <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.6.1</version>
        </dependency>
        <dependency>
            <groupId>org.apache.storm</groupId>
            <artifactId>storm-core</artifactId>
            <version>1.0.2</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

项目根目录下创建源码目录 `src/main/java`

然后在 java 目录下创建包目录 `com/storm/test`

现在项目的目录结构如下：

```
├── pom.xml
├── src
│   └── main
│       └── java
│           └── com
│               └── storm
│                   └── test
```

（3）安装依赖

项目根目录下执行maven命令

```
mvn package
mvn dependency:tree
```

（4）代码

下面是具体代码：

FakeCallLogReaderSpout.java

```
package com.storm.test;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.Random;

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;

public class FakeCallLogReaderSpout extends BaseRichSpout {
  private SpoutOutputCollector collector;
  private boolean completed = false;

  private TopologyContext context;

  private Random randomGenerator = new Random();
  private Integer idx = 0;

  public void open(Map map, TopologyContext topologyContext, Sp
outOutputCollector spoutOutputCollector) {
    this.context = topologyContext;
    this.collector = spoutOutputCollector;
  }

  public void nextTuple() {
    if (this.idx <= 1000) {
      List<String> mobileNumbers = new ArrayList<String>();
```

```
      mobileNumbers.add("1234123401");
      mobileNumbers.add("1234123402");
      mobileNumbers.add("1234123403");
      mobileNumbers.add("1234123404");

      Integer localIdx = 0;
      while (localIdx++ < 100 && this.idx++ < 1000) {
        // 随机生成通话记录
        String fromMobileNumber = mobileNumbers.get(randomGener
ator.nextInt(4));
        String toMobileNumber = mobileNumbers.get(randomGenerat
or.nextInt(4));

        while (fromMobileNumber == toMobileNumber) {
          toMobileNumber = mobileNumbers.get(randomGenerator.ne
xtInt(4));
        }

        Integer duration = randomGenerator.nextInt(60);
        this.collector.emit(new Values(fromMobileNumber, toMobi
leNumber, duration));
      }
    }
  }

  public void declareOutputFields(OutputFieldsDeclarer declarer
) {
    declarer.declare(new Fields("from", "to", "duration"));
  }

}
```

CallLogCreatorBolt.java

```
package com.storm.test;

import java.util.Map;

import org.apache.storm.task.OutputCollector;
import org.apache.storm.task.TopologyContext;
import org.apache.storm.topology.IRichBolt;
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;
```

```
public class CallLogCreatorBolt implements IRichBolt{
    private OutputCollector collector;

       public void prepare(Map conf, TopologyContext context, O
utputCollector collector) {
          this.collector = collector;
       }

       public void execute(Tuple tuple) {
          String from = tuple.getString(0);
          String to = tuple.getString(1);
          Integer duration = tuple.getInteger(2);
          collector.emit(new Values(from + "-" + to, duration))
;
       }

       public void cleanup() {}

       public void declareOutputFields(OutputFieldsDeclarer dec
larer) {
          declarer.declare(new Fields("call", "duration"));
       }

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

CallLogCounterBolt.java

```
package com.storm.test;

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.IRichBolt;
import org.apache.storm.topology.OutputFieldsDeclarer;
import org.apache.storm.tuple.Fields;
import org.apache.storm.tuple.Tuple;

public class CallLogCounterBolt implements IRichBolt {
    Map<String, Map<String, Integer>> counterMap;
    private OutputCollector collector;
```

```
    public void prepare(Map conf, TopologyContext context, Outp
utCollector collector) {
        this.counterMap = new HashMap<String, Map<String, Integ
er>>();
        this.collector = collector;
    }

    public void execute(Tuple tuple) {
        String call = tuple.getString(0);
        Integer duration = tuple.getInteger(1);

        if (!counterMap.containsKey(call)) {
            Map<String, Integer> record = new HashMap<String, I
nteger>();
            record.put("count", 1);
            record.put("duration", duration);
            counterMap.put(call, record);
        } else {
            Map<String, Integer> record = counterMap.get(call);
            Integer count = record.get("count") + 1;
            Integer dur = record.get("duration") + duration;

            record.put("count", count);
            record.put("duration", dur);
            counterMap.put(call, record);
        }

        collector.ack(tuple);
    }

    // bolt 停止时调用此方法
    public void cleanup() {
        for(String mobile : counterMap.keySet()){
            Map<String, Integer> record = counterMap.get(mobile
);
            System.out.println(mobile + " : " + record.get("cou
nt") + "," + record.get("duration"));
        }
    }

    public void declareOutputFields(OutputFieldsDeclarer declar
er) {
        declarer.declare(new Fields("call"));
    }

    public Map<String, Object> getComponentConfiguration() {
```

```
            return null;
        }
    }
```

LogAnalyserStorm.java

```
package com.storm.test;

import org.apache.storm.Config;
import org.apache.storm.LocalCluster;
import org.apache.storm.topology.TopologyBuilder;
import org.apache.storm.tuple.Fields;

public class LogAnalyserStorm {
    public static void main(String[] args) throws Exception{
          Config config = new Config();
          config.setDebug(true);

          TopologyBuilder builder = new TopologyBuilder();
          builder.setSpout("call-log-reader-spout", new FakeCal
lLogReaderSpout());

          builder.setBolt("call-log-creator-bolt", new CallLogC
reatorBolt())
             .shuffleGrouping("call-log-reader-spout");

          builder.setBolt("call-log-counter-bolt", new CallLogC
ounterBolt())
             .fieldsGrouping("call-log-creator-bolt", new Field
s("call"));

          LocalCluster cluster = new LocalCluster();
          cluster.submitTopology("LogAnalyserStorm", config, bu
ilder.createTopology());
          Thread.sleep(10000);

          cluster.shutdown();
       }
}
```

（5）编译运行

```
mvn compile
```

```
mvn exec:java -Dexec.mainClass="com.storm.test.LogAnalyserStorm
" -Dexec.cleanupDaemonThreads=false
```

会输出大量的log信息，执行完成后，会在底部附近输出程序的执行结果，类似如下信息：

```
1234123401-1234123404 : 91,2909
1234123403-1234123402 : 94,2971
1234123404-1234123403 : 90,2576
1234123403-1234123404 : 72,2208
1234123404-1234123401 : 108,3192
1234123402-1234123404 : 93,2354
1234123402-1234123403 : 84,2400
```

小结

storm 开发的基本思路：

- spout 中与外部数据源对接，然后发送给内部的 bolt

spout 中的主要方法 `nextTuple()` 被循环调用，在这里面处理数据的接收、发射

- bolt 中定义数据处理逻辑，对接收到的数据进行处理

其主要方法 `execute()` 每次收到数据时被调用，在这里定义处理逻辑

- topology 中把 spout 和 bolt 串起来，定义好上下游关系，然后提交到 storm 执行

先把这个最简单的思路理解好，然后在此基础上进行扩充，学习更多的用法就简单了，希望本文可以帮助您快速认识 Storm

公众号 性能与架构