第十讲　流计算系统Storm

徐 辰
cxu@dase.ecnu.edu.cn
華東師範大學

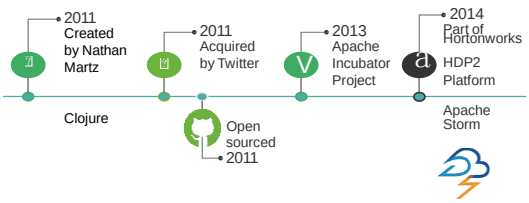
流数据

2

- 近年来，在Web应用、网络监控、传感监测等领域，兴起了一种新的数据密集型应用——流数据，即数据以大量、快速、时变的流形式持续到达
- 实例：PM2.5检测、电子商务网站用户点击流
- 流数据具有如下特征：
  - 数据快速持续到达，潜在大小也许是无穷无尽的
  - 数据来源众多，格式复杂
  - 数据量大，但是不十分关注存储，一旦经过处理，要么被丢弃，要么被归档存储
  - 注重数据的整体价值，不过分关注个别数据
  - 数据顺序颠倒，或者不完整，系统无法控制将要处理的新到达的数据元素的顺序

Storm历史

3



大纲

4

- 设计思想
  - 数据模型
  - 计算模型
- 体系架构
- 工作原理
- 容错机制
- 编程实例

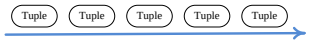
数据模型：Tuple

5

- 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

Streams

无界的Tuple序列



- 从逻辑上看，Storm的tuple类似于关系数据库中的tuple，是一个值列表

Field1	Field2	Field3	…
value1	value2	value3	…

大纲

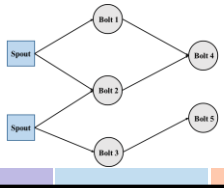
6

- 设计思想
  - 数据模型
  - 计算模型
- 体系架构
- 工作原理
- 容错机制
- 编程实例

逻辑计算描述：Topology

7

- 流转换图：描述Spouts和Bolts组成的网络
  - 顶点：Spout或Bolt（处理逻辑）
  - 边：Bolt订阅哪个Stream（数据流动的方向）
    - 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

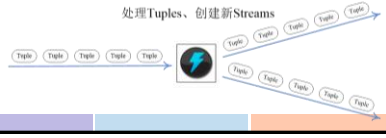


逻辑计算描述：Topology

8

- Spout：Stream的源头，从外部数据源（Kafka、数据库等）读取数据，然后封装成Tuple，发送到Stream中

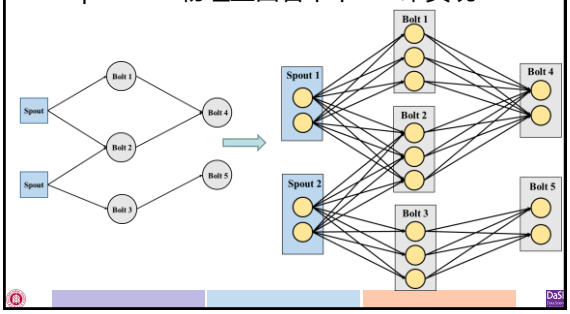
- Bolt：描述Streams的转换过程，将处理后的Tuple作为新的Streams发送给其他Bolt



物理计算描述

9

- Spout/Bolt物理上由若干个task来实现



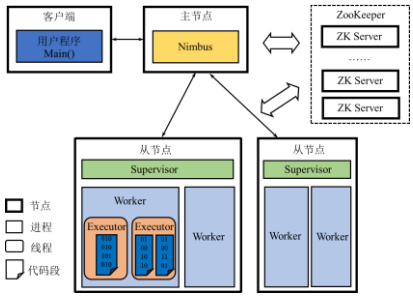
大纲

10

- 设计思想
- 体系架构
  - 架构图
  - 应用程序执行流程
- 工作原理
- 容错机制
- 编程实例

Storm架构

11



Storm角色

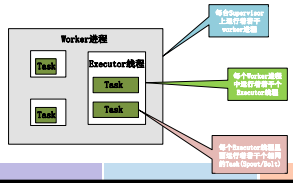
12

- Nimbus：主节点运行的后台程序，负责分发代码、分配任务和监测故障
- Supervisor：从节点运行的后台程序
  - 负责监听所在机器的工作，根据Nimbus分配的任务来决定启动或停止Worker进程
  - 一个从节点上同时运行若干个Worker进程
- Zookeeper：负责Nimbus和Supervisor之间的所有协调工作
  - 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

Storm角色

13

- Worker：运行一个或多个executor线程来提供task的运行服务
  - executor：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: 执行数据处理的代码实例（spout/bolt）



Storm与MapReduce/Spark比较

14

	MapReduce	Storm	Spark
系统进程	JobTracker	Nimbus	Master
	TaskTracker	Supervisor	Worker
	Child	Worker	CoarseGrainedExecutorBackend
工作线程	/	Executor	Task
任务代码	Task	Task	
基础接口	Map/Reduce	Spout/Bolt	RDD

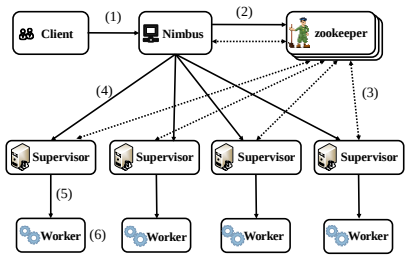
大纲

15

- 设计思想
- 体系架构
  - 架构图
  - 应用程序执行流程
- 工作原理
- 容错机制
- 编程实例

执行流程

16



执行流程

17

1. 用户编写的Topology程序，经过序列化、打包并提交给主节点Nimbus。
2. Nimbus创建一个组件与物理节点的对应关系文件，将该文件原子地写入Zookeeper中某Znode
3. 所有Supervisor因监听Znode来得到通知从而获取所在节点所需执行的组件任务。
4. Supervisor需要从Nimbus处拉取可执行的代码。
5. Supervisor启动若干Worker进程执行具体的任务
6. Worker进程根据ZooKeeper中获取的文件信息，启动若干个Executor线程，该线程负责执行组件（Spout或Bolt）所描述的任务Task。

大纲

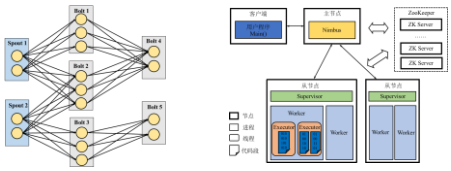
18

- 设计思想
- 体系架构
- 工作原理
  - 流分组策略
  - 消息传递
- 容错机制
- 编程实例

连续处理(continuous processing)

19

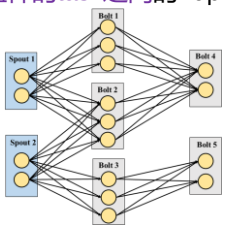
- 系统运行Topology的过程中，一个Spout或Bolt通常由多个task同时执行
- 系统中的所有task按照Topology进行部署后一直长期驻留并执行流计算任务



Task之间的Tuple传输

20

- Spout和Bolt之间，或者不同的Bolt之间的Tuple传输表现为属于上游组件的task和属于下游组件的task之间的Tuple传输。



Tuple传输中的问题

21

- 对于一组Tuple来说，上游组件的task发送哪些Tuple给下游组件的task?
  - 如何对这组Tuple进行划分?
- 对于一个Tuple来说，上游组件的task如何向下游组件的task传递Tuple?
  - 立即传输 vs. 成批传输?

大纲

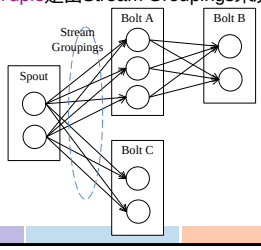
22

- 设计思想
- 体系架构
- 工作原理
  - 流分组策略
  - 消息传递
- 容错机制
- 编程实例

Stream Groupings

23

- 定义如何在两个组件间（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都可以由多个分布式task执行，task如何分发一组Tuple是由Stream Groupings来决定的



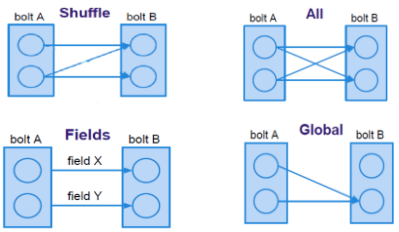
Stream Groupings

24

- 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
- DirectGrouping：直接分组，直接指定由某个Task来执行Tuple的处理

Stream Groupings

25



大纲

26

- 设计思想
- 体系架构
- 工作原理
  - 流分组策略
  - 消息传递
- 容错机制
- 编程实例

Message Passing

27

- 消息传递机制：Task之间如何传递一条tuple?
  - Tuple处理完即可发送：无阻塞、符合实时计算要求
  - 一次一记录：一次发送一个tuple
- 与批处理方式相比
  - MapReduce和Spark的shuffle阶段存在阻塞
  - 数据传输是成块进行的

大纲

28

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程实例

故障类型

29

- ZooKeeper故障：极端情况
- Nimbus故障：怎么办?
- Supervisor故障
  - 重启新的Supervisor
  - 原来监控的所有Worker重新调度、启动
- Worker故障
  - 重新启动

大纲

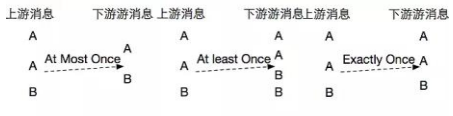
30

- 设计思想
- 体系架构
- 工作原理
- 容错机制
  - 容错语义
  - 元组树
  - ACK机制
  - 消息重放
- 编程实例

容错语义

31

- 流计算系统容错语义
  - At Most Once：消息可能会丢失
  - At Least Once：消息不会丢失，可能会重复
  - Exactly Once：消息不丢失，不重复



大纲

32

- 设计思想
- 体系架构
- 工作原理
- 容错机制
  - 容错语义
  - 元组树
  - ACK机制
  - 消息重放
- 编程实例

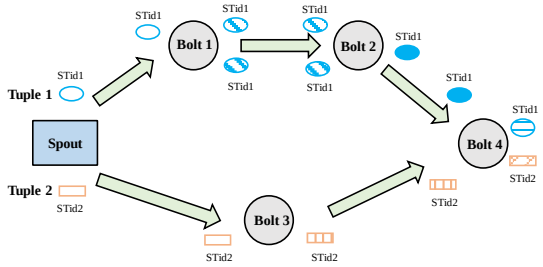
元组树

33

- Topology中的Spout会源源不断地发射出Tuple，每一条Tuple都会由后续的Bolt处理并演化为新的Tuple
- 元组树：Spout发出的Tuple及其衍生出来的Tuple抽象为一棵树
  - Spout中每一条元组都对应一棵元组树
- Spout-Tuple-id(STid)：Spout中发射Tuple时用户可以为其指定标识
  - 该STid伴随着元组在处理过程中的演化

元组树

34



大纲

35

- 设计思想
- 体系架构
- 工作原理
- 容错机制
  - 容错语义
  - 元组树
  - ACK机制
  - 消息重放
- 编程实例

Mid vs. STid

36

- 元组树中的Tuple传输物理上表现为组件的Task之间的消息传输，该消息有一个64位标识，称为Mid
  - Mid是系统定义的消息传递标识，每条消息的Mid都不同
  - STid是用户定义的标识，同一元组树的STid都相同

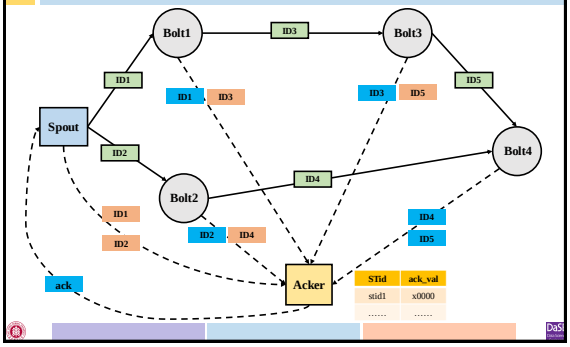
Ack机制

37

- Storm里面有一类特殊的Task作为Acker，负责跟踪Spout发出的元组及其元组树
- Ack机制
  - 上游组件的Task发射消息的同时，会向Acker报告Mid及STid
  - 当下游组件的Task接收到消息时向Acker报告Mid及STid

Ack机制

38



Acker数据结构

39

- 元组树中的元组非常多，并且往往并行地向Acker报告Mid和STid
- Acker端需要维护类似于<STid, list<Mid>>的映射表
  - 维护list对于Acker来说内存开销太大

Acker数据结构

40

- <STid, ack_val>映射表
  - 收到Spout发来消息时将相应STid的ack_val初始化为0
  - 无论Acker接收到上游组件还是下游组件报告的消息，均将其中的Mid与映射表中相应STid的ack_val进行异或（XOR）操作
  - 如果Acker在设定的时间范围内收到处于拓扑最末端的Bolt报告并且ack_val为0，那么Acker会告诉相应的Spout：STid对应的元组树已经成功地处理完。

STid	ack_val
stid1	x0000
……	……

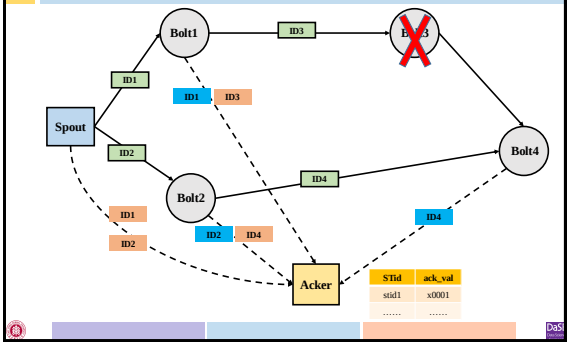
大纲

41

- 设计思想
- 体系架构
- 工作原理
- 容错机制
  - 容错语义
  - 元组树
  - ACK机制
  - 消息重放
- 编程实例

故障发生

42



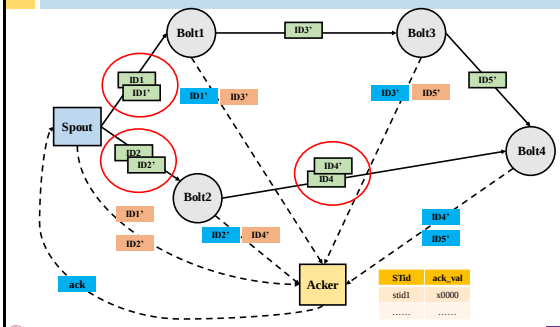
故障处理

43

- Storm认为消息的传输发生了故障
  - Acker在设定的时间范围内
    - STid对应的ack_val不为0
    - 或者无法收到拓扑最末端的Bolt发来的关于STid的确认消息
- Spout重新发送以STid为标识的元组。但这种消息重放机制可能会导致消息的重复计算，达到的是至少一次的容错语义级别

消息重放

44



大纲

45

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程实例
  - WordCount
  - 开启Ack机制的WordCount
  - 热门Twitter话题

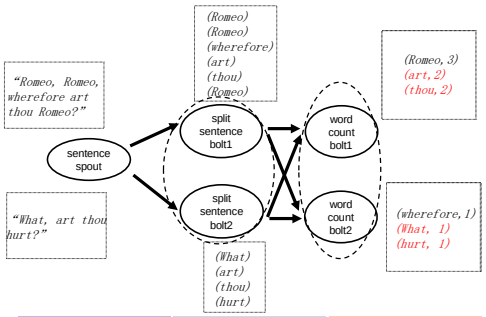
WordCount

46

- 基于Storm的单词统计在形式上与基于MapReduce的单词统计是类似的，MapReduce使用的是Map和Reduce的抽象，而Storm使用的是Spout和Bolt的抽象
- Storm进行单词统计的整个流程：
  - 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

WordCount运行实例

47



WordCount Topology

48

```
import org.apache.storm.Config;
Import ……
public class WordCountTopology {
public  static class RandomSentenceSpout extends BaseRichSpout {
……
}
public static class SplitSentence extends ShellBolt implements IRichBolt {
……
}
 public static class WordCount extends BaseBasicBolt {
……
}
 public static void main(String[] args) throws Exception {
   TopologyBuilder builder = new TopologyBuilder();
   builder.setSpout("sentences", new RandomSentenceSpout(), 5);
   builder.setBolt("split", new SplitSentence(), 8).shuffleGrouping("sentences");
   builder.setBolt("count", new WordCount(), 12).fieldsGrouping("split", new Fields("word"));
   ……
}
}
```

SentenceSpout

49

```
public class RandomSentenceSpout extends BaseRichSpout {
  SpoutOutputCollector _collector;
  Random _rand;
 @Override
  public void nextTuple() {
    String[] sentences = new String[]{ "the cow jumped over the moon", "an apple a day keeps the
        doctor away", "four score and seven years ago", "snow white and the seven dwarfs" };
    String sentence = sentences[_rand.nextInt(sentences.length)];
    _collector.emit(new Values(sentence));
  }

  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("sentences"));
  }
}
```

SplitSentenceBolt

50

- 如SplitSentence()方法虽然是通过Java语言定义的，但具体的操作可通过Python脚本来完成
- Topology里面的每个组件必须定义它要发射的Tuple的每个字段

```
public static class SplitSentence extends ShellBolt implements IRichBolt {
    public SplitSentence() {
        // 单词分割的具体实现由Python脚本实现
        super("python", "splitsentence.py");
    }
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}
```

SplitSentenceBolt

51

- Python脚本splitsentence.py定义了一个简单的单词分割方法，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

```
# 简单的单词分割实现，以空格作为分割点
import storm
class SplitSentenceBolt(storm.BasicBolt):
    def process(self, tup):
        words = tup.values[0].split(" ")
        for word in words:
            storm.emit([word])
SplitSentenceBolt().run()
```

WordCountBolt

52

- 单词统计的具体逻辑：首先判断单词是否统计过，若未统计过，需先将count值置为0。若单词已统计过，则每出现一次该单词，count值就加1

```
// 对单词进行计数
public static class WordCount extends BaseBasicBolt {
    Map<String, Integer> counts = new HashMap<String, Integer>();
    @Override
    public void execute(Tuple tuple, BasicOutputCollector collector) {
        String word = tuple.getString(0);
        Integer count = counts.get(word);
        if (count == null)
        count = 0;
        count++;
        counts.put(word, count);
        collector.emit(new Values(word, count));
    }
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word", "count"));
    }
}
```

大纲

53

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程实例
  - WordCount
  - 开启Ack机制的WordCount
  - 热门Twitter话题

SentenceSpout

54

```
public class RandomSentenceSpout extends BaseRichSpout {
  SpoutOutputCollector _collector;
  Random _rand;
  private HashMap<String, String> waitAck = new HashMap<String, String>();
 @Override
  public void nextTuple() {
    String[] sentences = new String[]{ "the cow jumped over the moon", "an apple a day keeps the
        doctor away", "four score and seven years ago", "snow white and the seven dwarfs"};
    String sentence = sentences[_rand.nextInt(sentences.length)];
    String spout_tuple_id = UUID.randomUUID().toString().replaceAll("-", "");
    waitAck.put(spout_tuple_id, sentence);
    _collector.emit(new Values(sentence) , spout_tuple_id);
  }
@Override
 public void ack(Object id) {  waitAck.remove(id);   }
@Override
public void fail(Object id) { _collector.emit(new Values(waitAck.get(id)), id);   }
  @Override
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("sentences"));}
  }
```

大纲

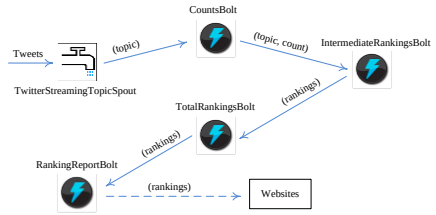
55

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程实例
  - WordCount
  - 开启Ack机制的WordCount
  - 热门Twitter话题

Storm@Twitter

56

- 上述虽然是一个简单的单词统计，但对其进行扩展，便可应用到许多场景中，如微博中的实时热门话题。Twitter也正是使用了Storm框架实现了实时热门话题



Twitter实时热门话题处理流程示意图

课后阅读

57

- 论文
  - Toshniwal, A., Donham, J., Bhagat, N., Mittal, S., Ryaboy, D., Taneja, S., … Fu, M. (2014). Storm@twitter. In SIGMOD Conference (pp. 147–156).

本讲小结

58

- 设计思想
- 体系架构
- 工作原理
- 容错机制
- 编程实例

谢谢！Q&A