Streaming应用开发

www.huawei.com

HUAWEI

目标

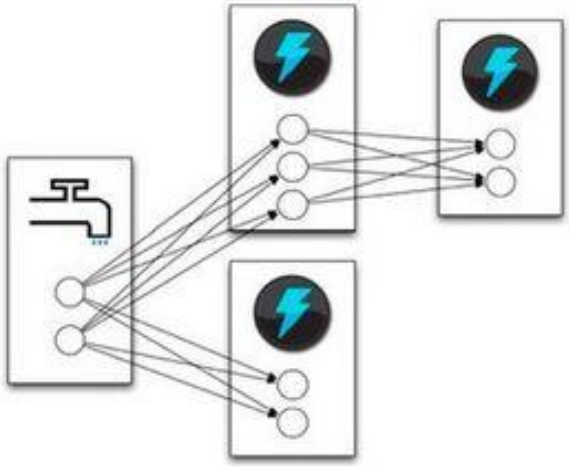
- 学完本课程后，您将能够:
  - 掌握**Streaming**基本业务开发流程
  - 熟悉**Streaming**常用**API**接口使用
  - 掌握**Streaming**业务设计基本原则
  - 了解**Streaming**应用开发环境
  - 了解**CQL**开发流程及使用

目录

Streaming的定义

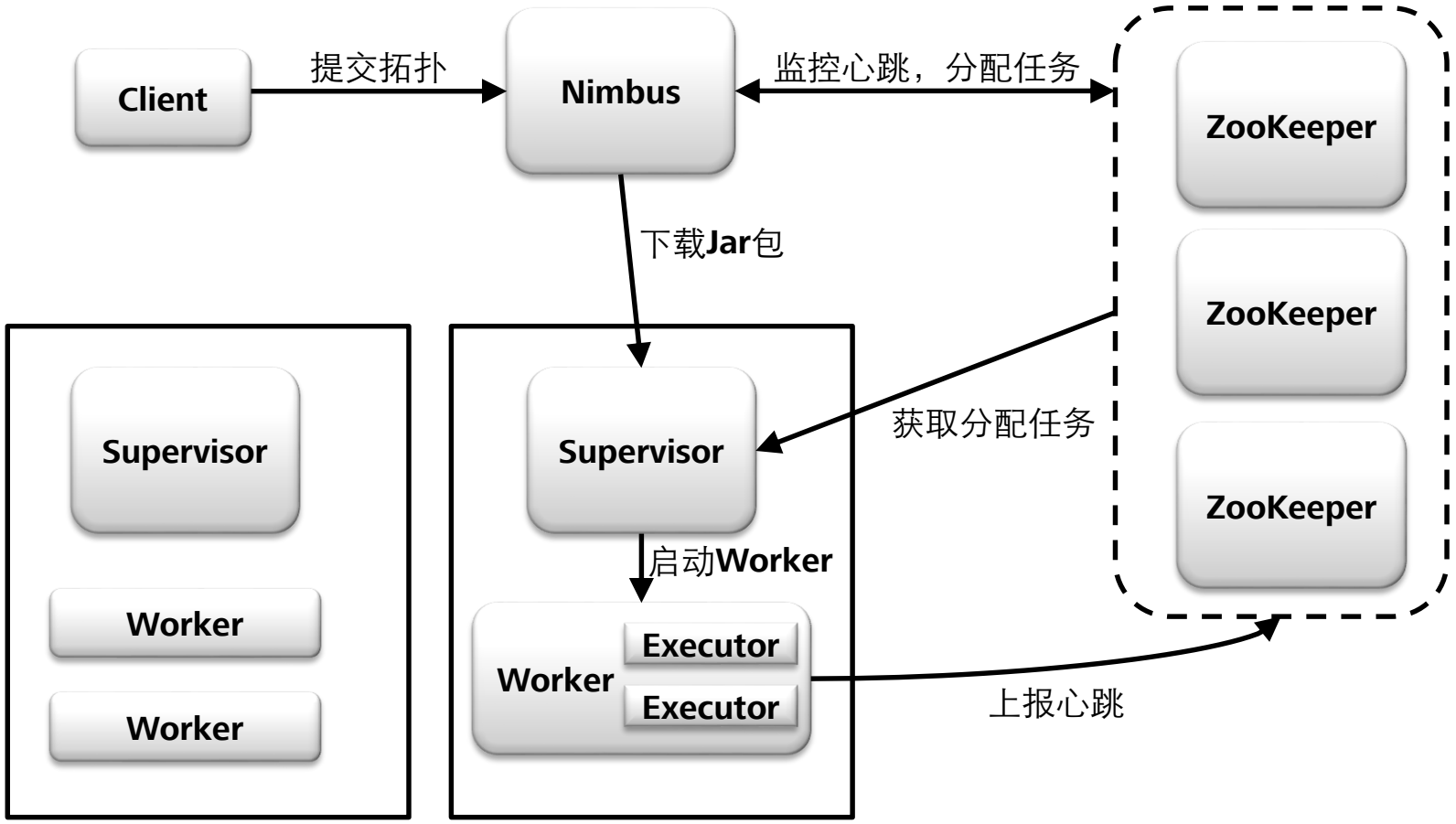
Streaming基于开源的**Storm**，是一个分布式、实时计算框架。**Streaming**在开源**Storm**的基础上增加了持续查询语言**CQL**、增强了安全性和可靠性。

- ✓ 事件驱动
- ✓ 连续查询
- ✓ 数据不存储，先计算
- ✓ 实时响应，低延迟



CQL(Continuous Query Language)，持续查询语言，是一种用于实时数据流上的查询语言，它是一种**SQL-like**的语言，目前主要适配**Storm** 。相对于**SQL**，**CQL**中增加了（时序）窗口的概念，将待处理的数据保存在内存中，进行快速的内存计算，**CQL**的输出结果为数据流在某一时刻的计算结果。使用**CQL**，可以快速进行业务开发，并方便地将业务提交到**Storm**平台开启实时数据的接收、处理及结果输出；并可以在合适的时候中止业务。

Streaming架构回顾

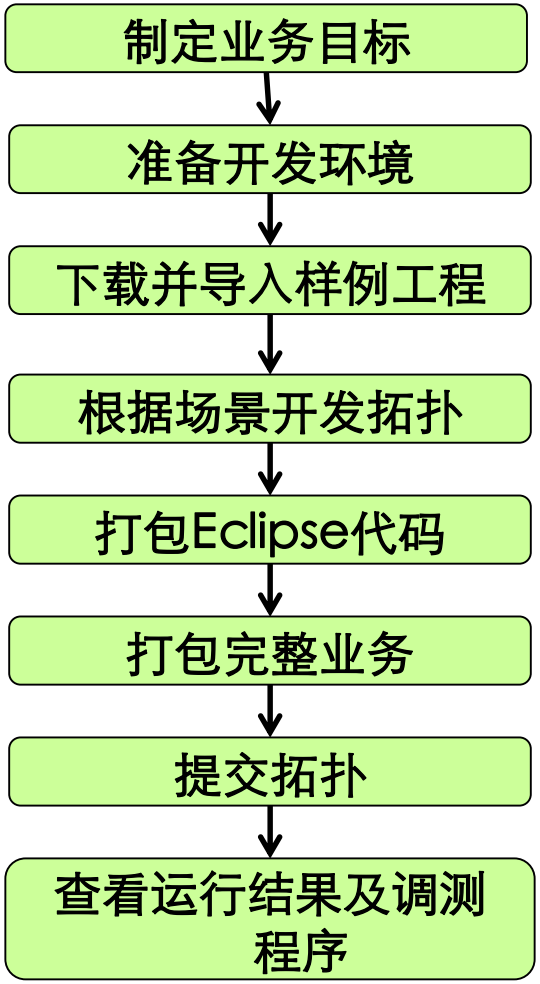


Streaming的适用场景

- **Streaming**主要应用于以下几种对响应时延有严格要求的场景：
  - ✓ 实时分析：如实时日志处理、交通流量分析等
  - ✓ 实时统计：如网站的实时访问统计、排序等
  - ✓ 实时推荐：如实时广告定位、事件营销等

目录

Streaming应用开发流程



指定业务目标

- 数据源？**Spout**从哪里获取数据，例如**Kafka**、**TCP**或者**MQ**等。
- 结果输出到哪里？写入**Kafka**、**HDFS**或者**Redis**等。
- 拓扑结构设计。**Spout**、**Bolt**如何组织。
- 可靠性要求？是否带**Acker**?

准备开发环境

准备项	说明
操作系统	**Windows**系统，推荐**Windows 7**以上版本。
安装**JDK**	开发环境的基本配置。版本要求：**1.7**或者**1.8**。
安装和配置**Eclipse**	用于开发**Streaming**拓扑的工具。
网络	确保客户端与**Streaming**服务主机在网络上互通。

下载并导入Streaming样例工程

1、下载并解压**Streaming**客户端压缩包。

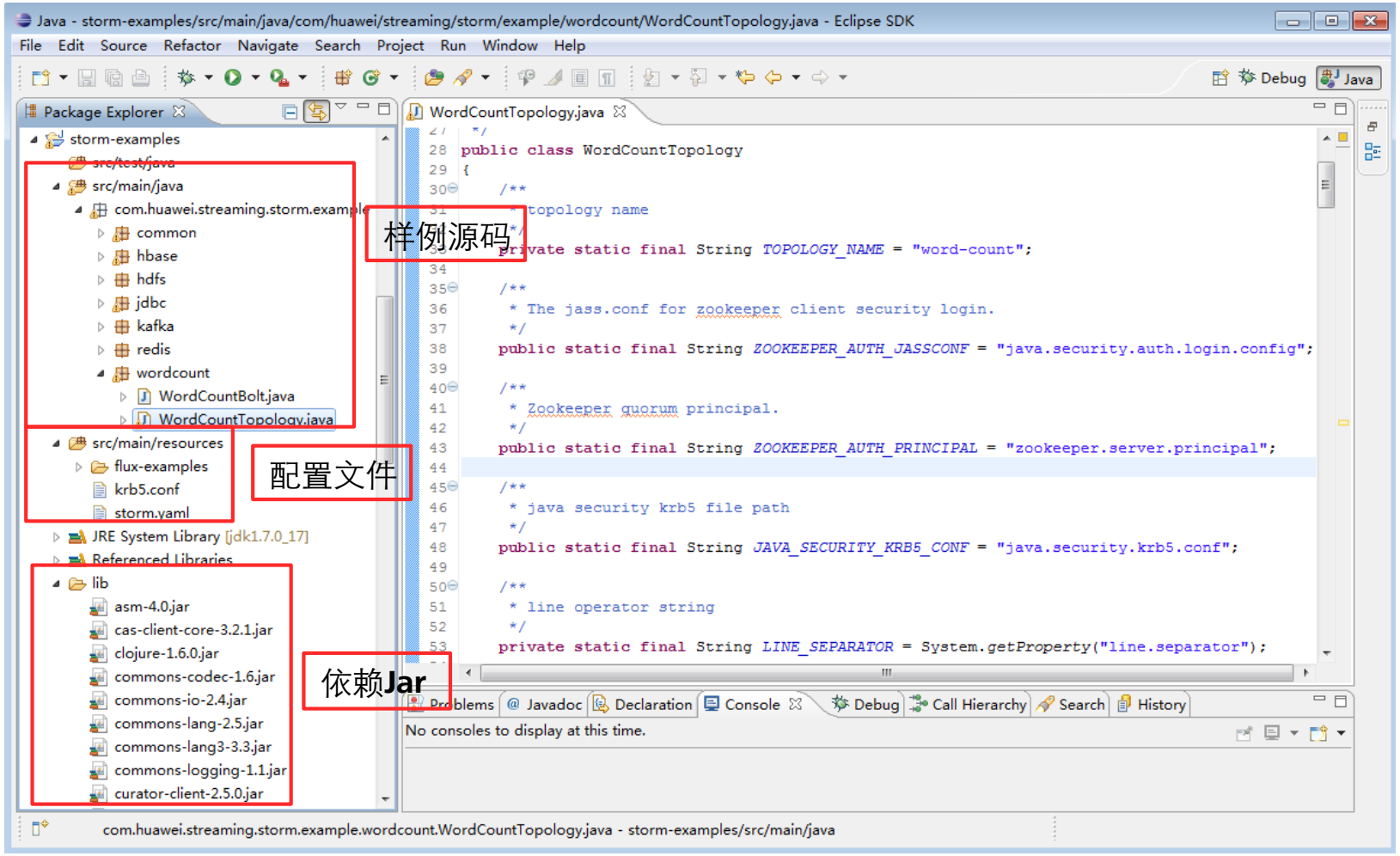
2、在**FusionInsight Manager**页面新建人机或机机账户，用于登录与操作。

3、下载用户的认证凭据文件。

4、导入客户端中“**storm-examples**”样例工程到**Eclipse**开发环境。

5、配置认证凭据文件到样例工程的“**src/main/resources**”下。

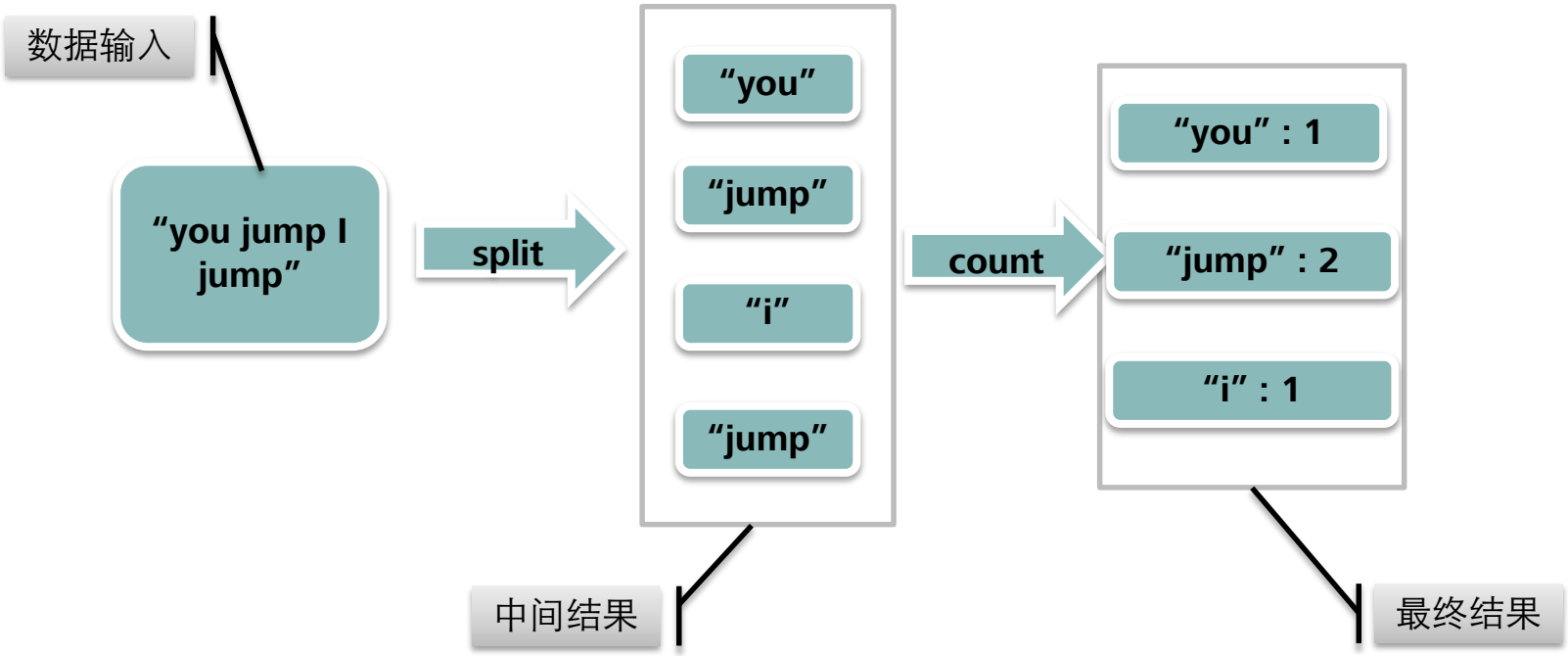
客户端工程示例



根据场景开发拓扑

- 根据功能实现**Spout/Bolt**。
- 熟悉**Storm**的**API**，调用相应的**API**构造拓扑。
- 业务代码开发完成后，参考样例代码**WordCountTopology.java**选择提交方式，如**Local**、**Remote**和**CMD**等。

一个例子-WordCount



一个例子-WordCount

Spout的初始化，该方法会在业务进程启动时调用一次

Spout的数据发送方法，使用随机数挑选语句字符串发送，该方法会被重复调用

一条消息发送失败后会回调**fail**方法，这里不作处理，一般在这里实现消息重发逻辑

定义所发送消息的格式，这里发送给下一跳的字段为**"word"**

```
public class RandomSentenceSpout extends BaseRichSpout {
  SpoutOutputCollector _collector;
  Random _rand;
  public void open(Map conf, TopologyContext context,
    SpoutOutputCollector collector) {
     _collector = collector;
     _rand = new Random();
  }
  public void nextTuple() {
     String[] sentences = new String[]{ "the cow jumped
    over the moon", "an apple a day keeps the doctor
    away"};
     String sentence =
    sentences[_rand.nextInt(sentences.length)];
     _collector.emit(new Values(sentence));
  }
  public void ack(Object id) {}
  public void fail(Object id) {}
  public void declareOutputFields(OutputFieldsDeclarer
    declarer) {
     declarer.declare(new Fields("word"));  }}
```

一个例子-WordCount

Bolt的执行逻辑，在收到消息后将字符串拆分，并循环发送给下一跳

定义发送给下一跳的消息格式，这里发送给下一跳的字段为**"word"**

```
public class SplitSentenceBolt extends BaseBasicBolt {

    private static final long serialVersionUID = -1L;
    public void execute(Tuple input,
    BasicOutputCollector collector) {
          String sentence = input.getString(0);
          String[] words = sentence.split(" ");
          for (String word : words) {
                    word = word.trim();
                    if (!word.isEmpty()) {
                              word = word.toLowerCase();
                              collector.emit(new
    Values(word));
                    } } }
    public void
    declareOutputFields(OutputFieldsDeclarer declarer)
    {
          declarer.declare(new Fields("word"));
    }
}
```

一个例子-WordCount

Bolt的执行逻辑，在收到消息后从缓存**map**中取出缓存的该字段的**count**值，自增后再存入**map**中，该**Bolt**作为最后一跳，没有将处理结果发送出去，用户可以自行选择消息的输出位置

定义发送给下一跳的消息格式，这里发送给下一跳的字段为**[“word”,”count”]**，作为最后一跳可以不用实现

```
public class WordCountBolt extends BaseBasicBolt {
    private static final long serialVersionUID = -1L;
    private 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);
    }
    @Override
    public void declareOutputFields(OutputFieldsDeclarer
    declarer) {
          declarer.declare(new Fields("word", "count"));
    } }
```

一个例子-WordCount

定义拓扑构造器

给拓扑设置**Spout**并指定其并发数

给拓扑设置**Bolt**，指定其并发数，并指定和上一跳消息分组方式

初始化**Storm**的**Config**对象，客户端参数可以设置在这里

设置客户端参数，这里设置**worker**个数为**3**个和**Acker**个数为**1**

使用提交器提交定义的拓扑，这里**args[0]**作为拓扑的名字

```
public static void main(String[] args) throws Exception
    {

     TopologyBuilder builder = new TopologyBuilder();

     builder.setSpout("spout", new
    RandomSentenceSpout(), 5);

     builder.setBolt("split", new SplitSentence(),
    8).shuffleGrouping("spout");
    builder.setBolt("count", new WordCount(),
    12).fieldsGrouping("split", new Fields("word"));

     Config conf = new Config();
     conf.setNumWorkers(3);
     conf.setNumAckers(1);

    StormSubmitter.submitTopologyWithProgressBar(args[0]
    , conf, builder.createTopology());
}
```

打包Eclipse代码

业务开发完成后，在开发环境**Eclipse**工程上，右击选择“**Export**”，在“**Export**”面板中选择“**Jar File**”，将工程中的“**src/main/java**”源码打包到指定路径，如“**D:\tmp\example.jar**”。

打包完整业务

- 将**Eclipse**打出的**example.jar**和业务中引入的外部**jar**包和相关配置文件整合到同一目录下，并打出最终的业务**jar**包。
- 在**Eclipse**工程的**tools**目录下找到打包工具：“**streaming-jartool.bat**”来完成打包。

提交拓扑

- 当前**Streaming**支持三种方式提交拓扑
  - **Linux**命令行提交—**CMD**模式：将打出的业务**jar**包上传至已安装**streaming**客户端的**linux**环境上，使用**storm jar <jarFile> <className> <topoName>**命令提交拓扑。
  - **Eclipse**远程提交——**Remote**模式：在提交前需要在本地进行安全准备，适配代码中的业务**jar**包路径和安全参数，然后右键->单击“**Run as > Java Application**”提交拓扑。
  - 本地模式提交——**Local**模式：该模式下需要拷贝业务所需的外部**jar**包及配置问价到本地工程并且加入到**classpath**中，然后右键->单击“**Run as > Java Application**”提交拓扑。本地模式一般用来测试。

查看运行结果及调测程序

- 查看运行结果
  - 登录**FusionInsight Manager**系统，选择“服务管理 **> Streaming**”，单击进入**Streaming WebUI**，在**Storm UI**中点击应用名称，查看应用程序运行情况。
- 调测程序
  - 日志分析，通过查看提交后的**Worker**日志进行分析业务逻辑的运行情况。
  - 单步调试
    - 登录**FusionInsight Manager**系统，选择“服务管理 **> Streaming**”，选择“服务配置”选项卡，在搜索框中搜索并修改**nimbus.task.timeout.secs**和**supervisor.worker.start.timeout.secs**的值为最大值，在**WORKER_GC_OPTS**的现有值后追加**-Xdebug -Xrunjdwp:transport=dt_socket,address=8011,suspend=n,server=y**，保存配置后重启相关实例。
    - 提交拓扑后，在**Storm UI**上查看**worker**进程运行的主机，使用**IDE**的远程调试功能连接**worker**进程启动的主机**ip**和**8011**端口，开始单步调试。

目录

Shell提交

- 下载并安装**Streaming**客户端到**Linux**客户端主机。
- 在**FusionInsight Manager**页面新建用户，用于登录与操作。
- 使用新建的用户执行**kinit**进行安全登录。
- 进入安装好的**Streaming**客户端“**streaming-cql-1.0/bin**”目录，执行“**cql**”进入**CQL Shell**。
- 在**Shell**中使用**CQL**语句定义并提交拓扑。

CQL Shell示例

```
[root@160-165-0-82 client]#
[root@160-165-0-82 client]# cql
 End CQL with ';' and end client with 'exit;'
Streaming>--添加用户自定义接口实现的jar包
>ADD JAR "/opt/streaming/example/example.jar";
Streaming>--使用用户自定义接口实现创建输入流
>CREATE INPUT STREAM S1(C1 STRING, C2 STRING)
>    SOURCE  "com.huawei.streaming.example.userdefined.operator.input.FileInput"
>    PROPERTIES ("fileinput.path" =  "/opt/streaming/example/input.txt");
Streaming>--创建输出流
>CREATE OUTPUT STREAM rs(C1 STRING, C2 STRING)
>    SINK kafkaOutput
>        PROPERTIES(groupid = "test", topic="ttopic");
Streaming>--应用程序逻辑部分
>INSERT INTO STREAM rs SELECT * FROM S1;
Streaming>--提交应用程序
>SUBMIT APPLICATION example;
Streaming>
```

Spouts (All time)

Id	Executors	Tasks	Emitted	Transferred	Complete latency (ms)	Acked	Failed
Input_001	1	1	129580	129580	0.000	0	0

Bolts (All time)

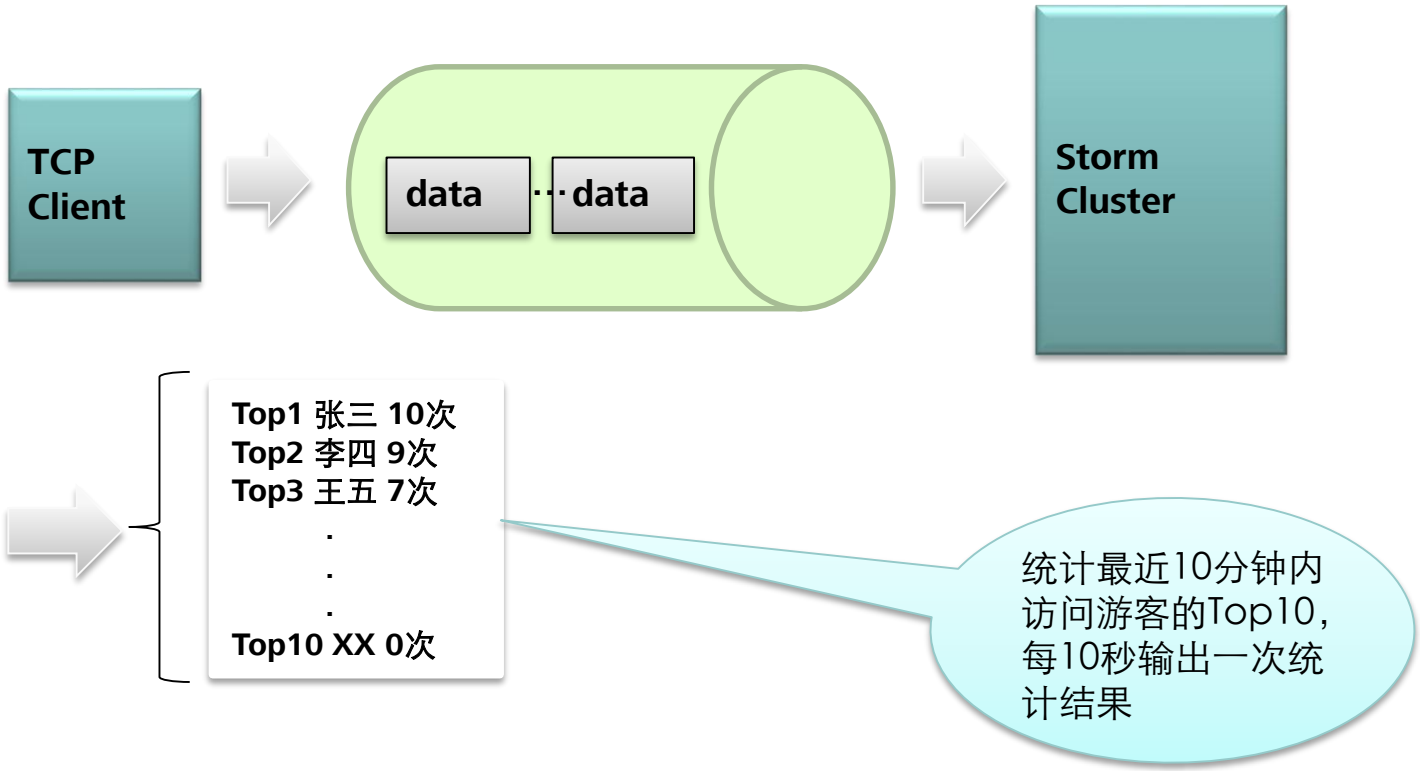
Id	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed
Functor_004	1	1	127000	127000	0.083	0.012	127000
KafkaOutput_002	1	1	0	0	1.104	0.173	121480

Eclipse远程提交

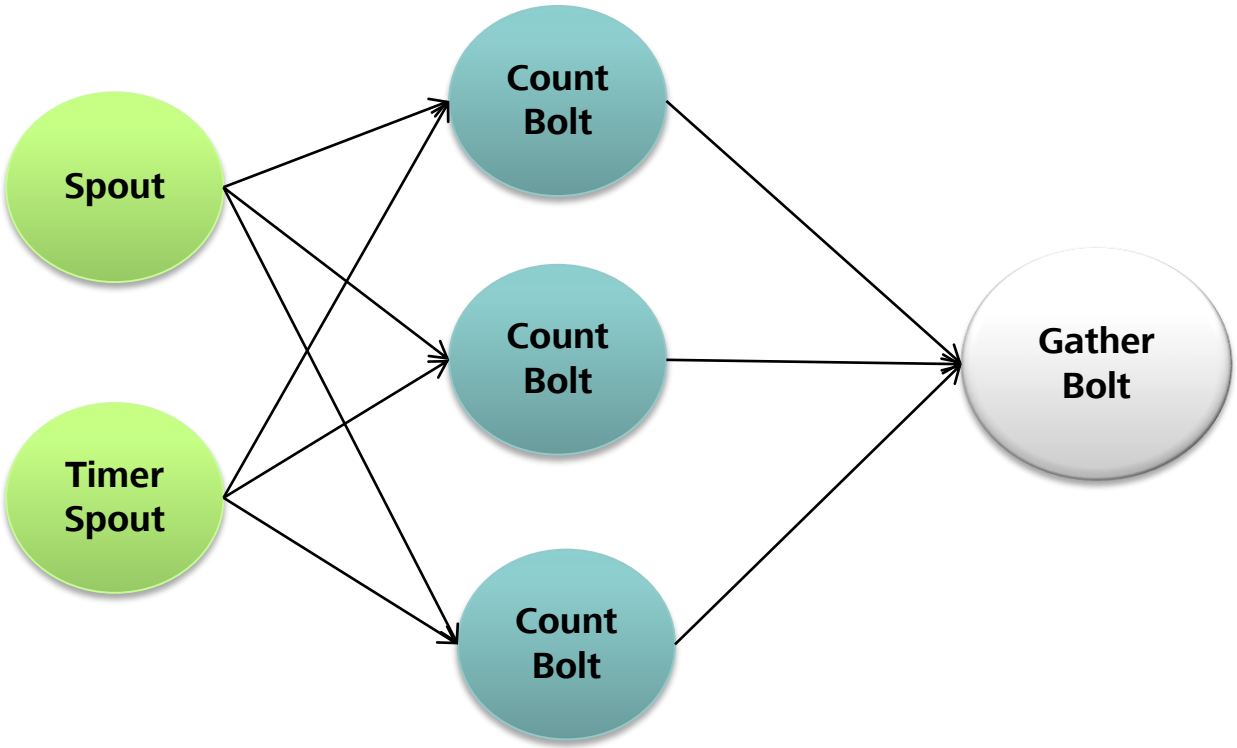
- 下载并安装**Streaming**客户端。
- 在**FusionInsight Manager**页面新建用户，用于登录与操作。
- 下载用户的凭据文件。
- 导入客户端中“**cql-examples**”样例工程到**Eclipse**开发环境。
- 配置认证凭据文件到样例工程的“**src/main/resources**”下
- 在本地开发**CQL**应用并保存到“**src/main/resources**”下，如“**example.cql**”。
- 适配**CQLExample.java**中的安全参数和待执行**CQL**文件地址
- 右键**->Run As->Java Application**执行**CQLExample.java**。

目录

固定时间窗口内TopN统计



业务模型/拓扑设计



Spout设计

功能	要求
数据接收	启动**TCP Server**
数据反序列化	需要对接收的数据进行反序列化处理
数据拆分	将反序列化后的数据根据预先设计好的**schema**进行拆分
数据筛选	对拆分后的数据进行筛选，筛选出关键信息
数据缓存	将提取的关键数（**username**）据缓存到固定大小的队列中
数据发送	从缓存中取出数据发送

Timer Spout设计

功能	要求
发送时间戳	每10秒发送一次时间戳

Counting Bolt设计

功能	要求
窗口定义	维护一个**10min**的窗口，窗口中缓存**<username, count>**，且只保留**count**值**Top10**数据。
刷新窗口	判断当前接收到的数据是不是时间戳，如果不是则刷新窗口中的数据并重新计算排序。
发送数据	判断当前接收到的数据是不是时间戳，如果是则将当前窗口内容发送给下一跳。

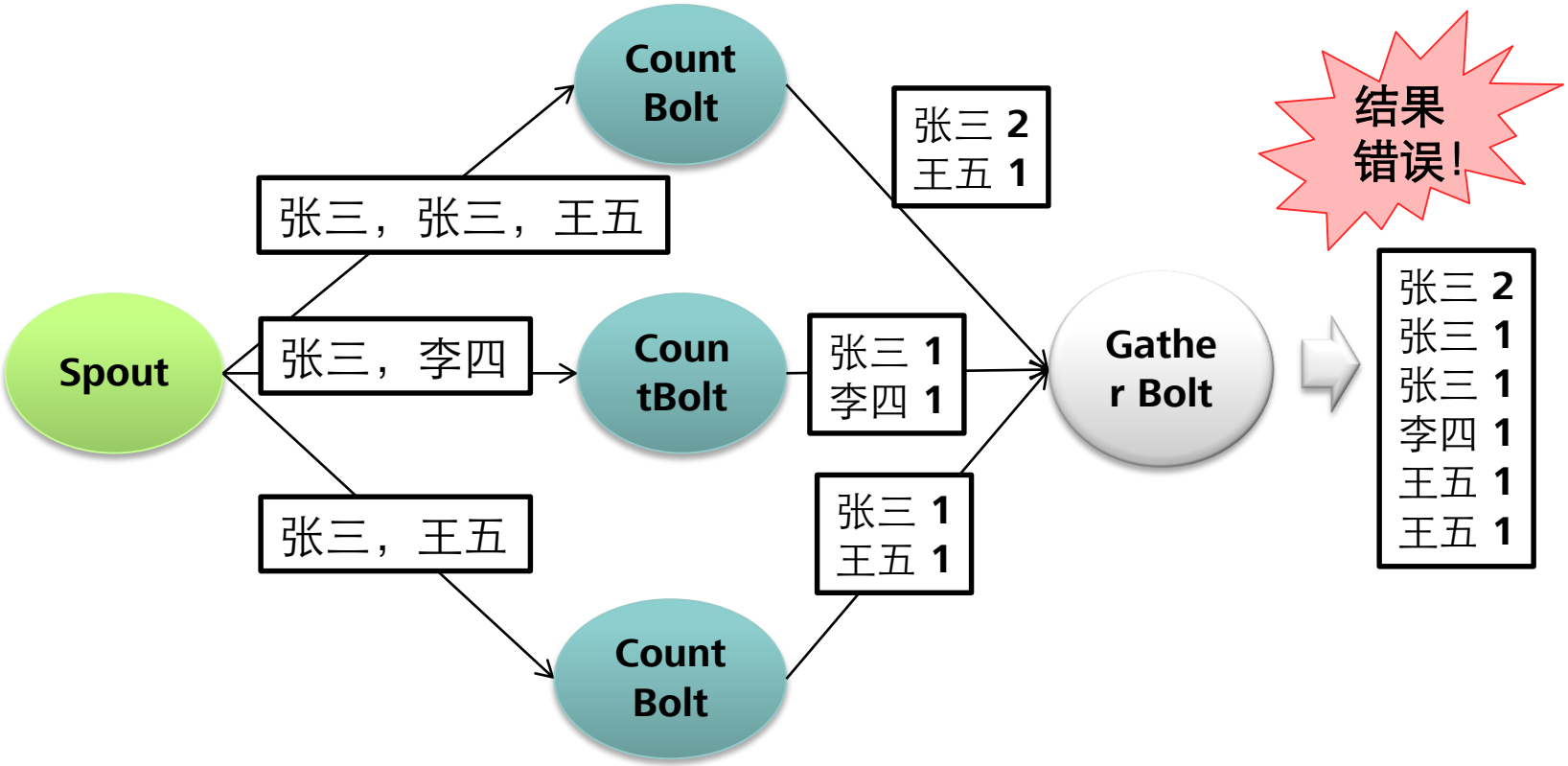
Gather Bolt设计

功能	要求
数据汇聚	将接收到的同一批次的数据汇总，根据时间戳频率，每**10**秒完成一次汇聚。
结果处理	将汇聚后的数据重新排序并保留**Top10**。
数据存储	将处理后的最终**Top10**结果存入**Kakfa/Redis/DB**等。

分组方式设计

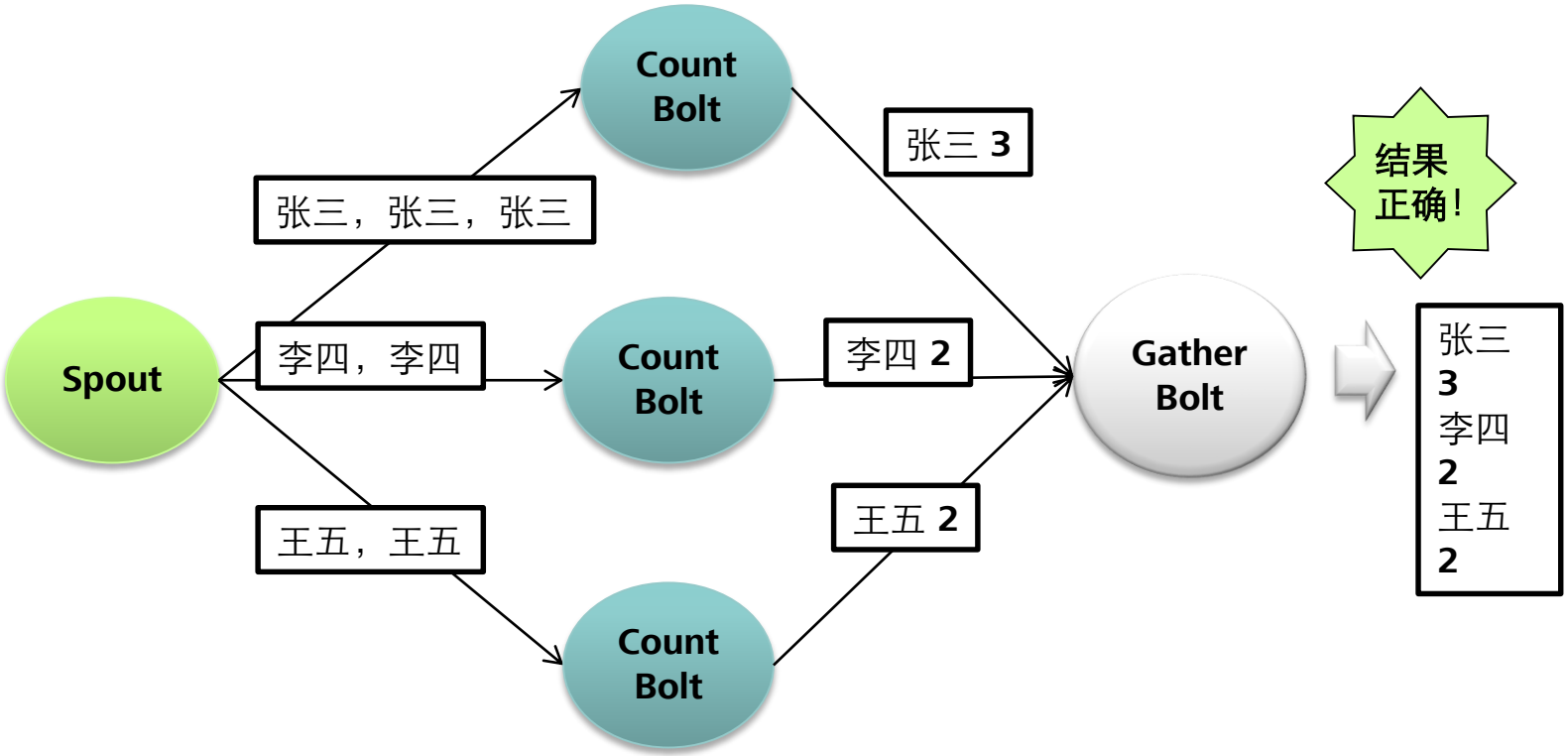
分组方式	功能介绍
fieldsGrouping（字段分组）	按照消息的哈希值分组发送给目标**Bolt**的**Task**。
globalGrouping（全局分组）	所有消息都发送给目标**Bolt**的固定一个**Task**。
shuffleGrouping（随机分组）	消息发送给目标**Bolt**的随机一个**task**。
localOrShuffleGrouping（本地或者随机分组）	如果目标**Bolt**在同一工作进程存在一个或多个**Task**，数据会随机分配给这些**Task**。否则，该分组方式与随机分组方式相同。
allGrouping（广播分组）	消息群发给目标**Bolt**的所有**Task**。
directGrouping（直接分组）	由数据生产者决定数据发送给目标**Bolt**的哪一个**Task**。需在发送时使用**emitDirect(taskID, tuple)**接口指定**TaskID**。
partialKeyGrouping（局部字段分组）	更均衡的字段分组。
noneGrouping（不分组）	当前和随机分组相同。

分组方式设计



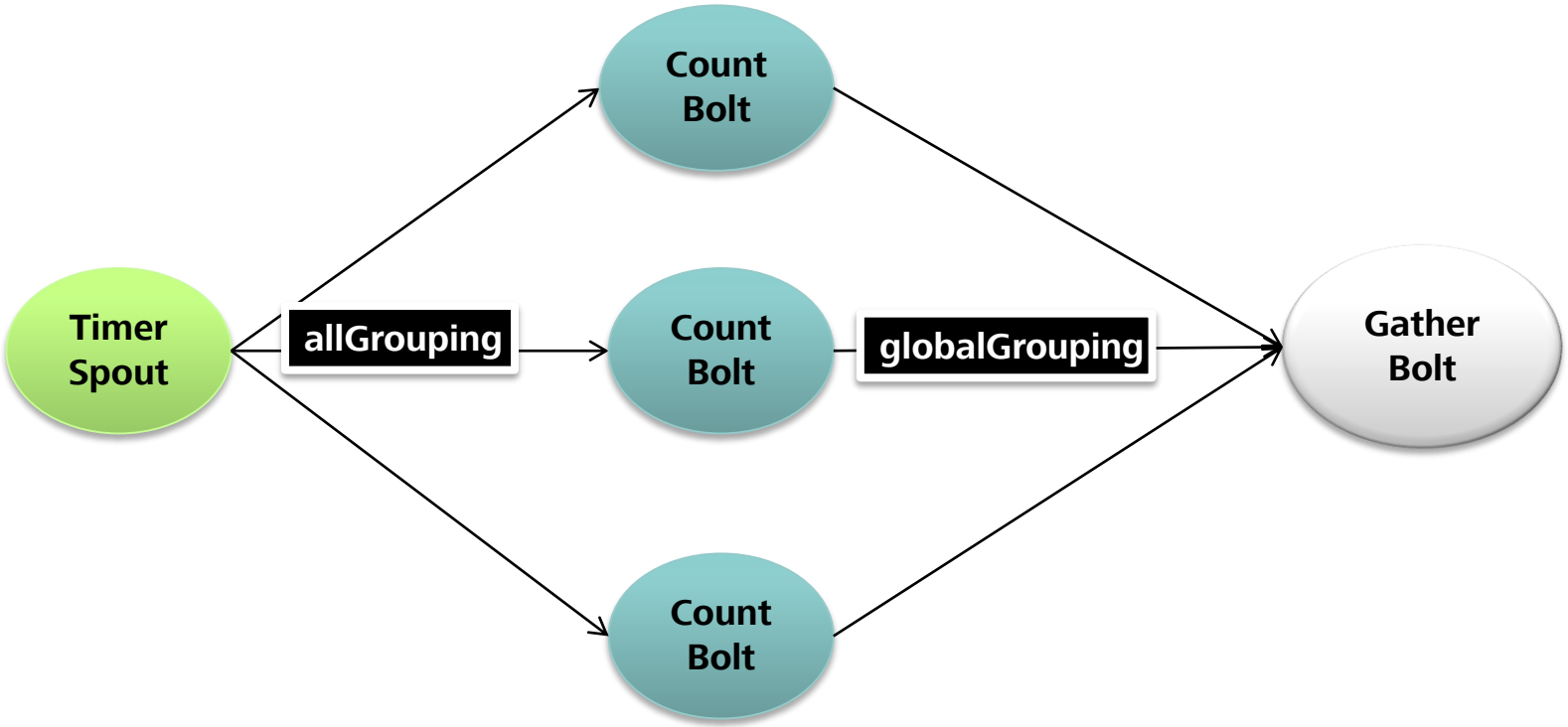
Spout和Count Bolt之间采用随机分组方式

分组方式设计



Spout和**Count Bolt**之间采用字段分组方式，以 **username**为关键字

分组方式设计



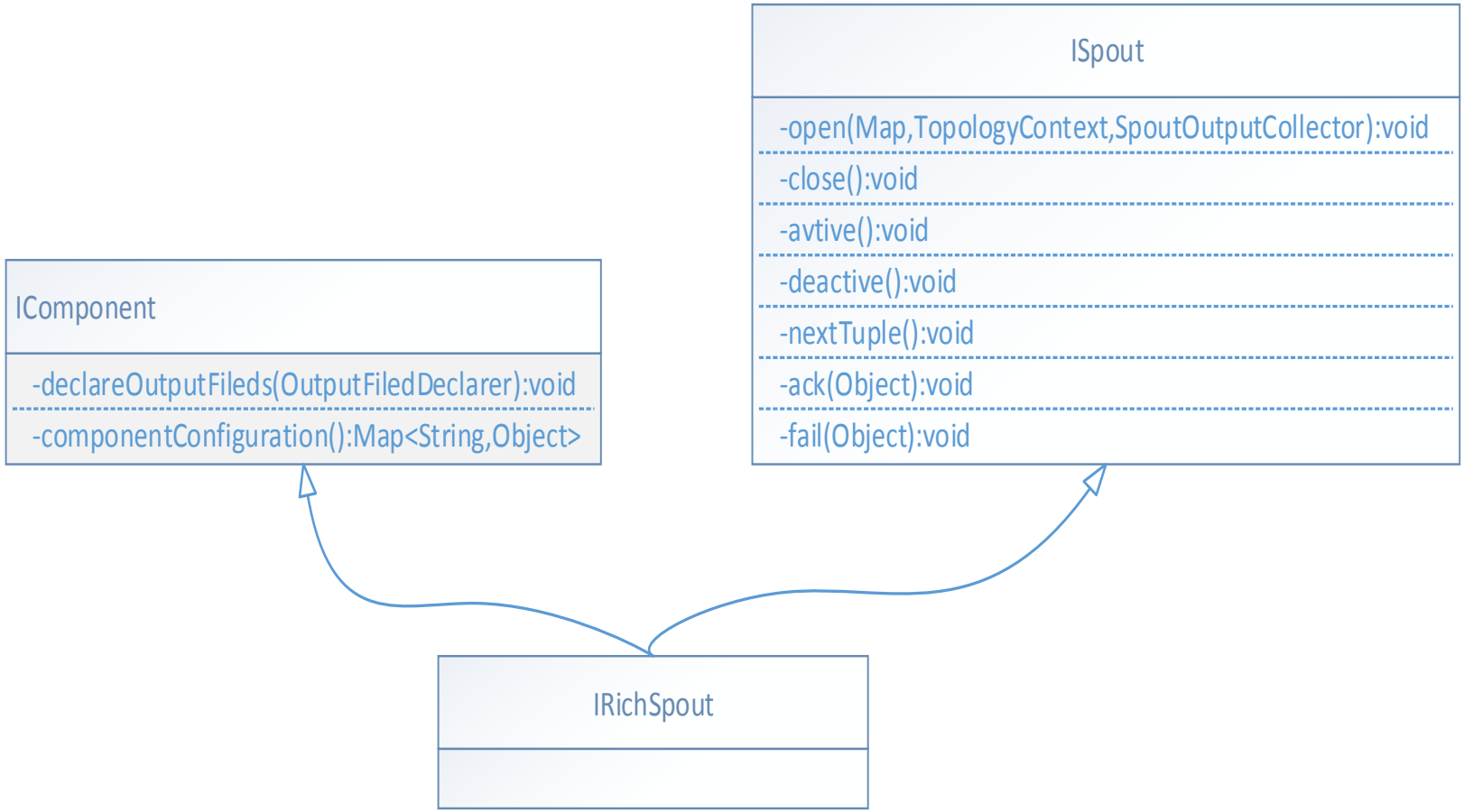
需要根据场景选择合适的分组方式，才能获得预期的结果

目录

Storm提供接口

- **REST** 接口
  - **REST**（**Representational State Transfer**）表述性状态转移接口。
- **Thrift**接口
  - 由**Nimbus**提供。**Thrift** 是一个基于静态代码生成的跨语言的**RPC**协议栈实现，它可以生成包括**C++, Java, Python, Ruby, PHP** 等主流语言的代码，这些代码实现了 **RPC** 的协议层和传输层功能，从而让用户可以集中精力于服务的调用和实现。

Spout接口



Spout接口

```
public class SampleSpout extends BaseRichSpout {
  SpoutOutputCollector _collector;

  public void open(Map conf, TopologyContext context, SpoutOutputCollector
collector) {}

  public void nextTuple() {}

  public void ack(Object id) {}

  public void fail(Object id) {}

  public void declareOutputFields(OutputFieldsDeclarer declarer) {}
}
```

1. **open**方法（**Spout**中第一个被调用的方法）中提供一个**SpoutOutputcollector**用来发射**tuple**。
2. **nextTuple**方法中从指定字符串列表中随机选择一个字符串，并发送出去，该方法被周期性调用。
3. **ack**方法用来处理消息发送成功后的逻辑，可以为空实现。
4. **fail**方法用来处理消息发送失败后的逻辑，一般这里用来实现消息重发逻辑。
5. **declareOutputFields**定义发送给下一跳的**tuple**中的字段。
6. **Storm**在检测到一个**Tuple**被整个**Topology**成功处理的时候调用**ack**，否则调用**fail**。

Spout接口

Spout的Ack开关

- 不启用**Ack**：

  在**nextTuple()**方法中使用不带**ack**的接口发送消息：

  collector.emit(new Values(sentence))。

- 启用**Ack**：

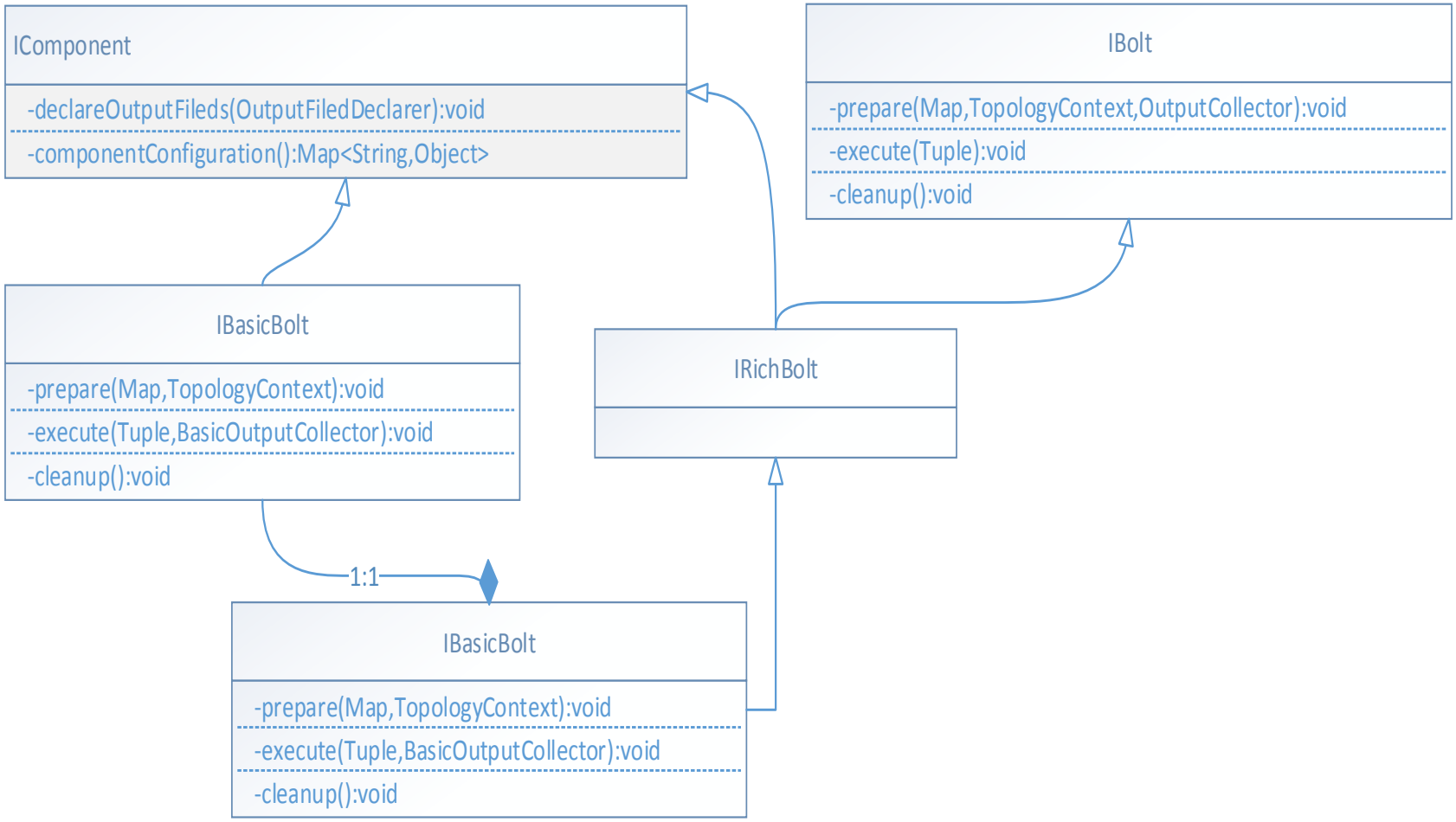
  - 构造一个全局唯一的**meaasgeId**

  如：**String meaasgeId = UUID.randomUUID().toString()**。

  - 在**nextTuple()**方法中使用带**ack**的接口发送消息：

  collector.emit(new Values(sentence), messageID)。

Bolt接口



Bolt接口

```
class SampleBolt implements IRichBolt {
    private OutputCollector collector;
    public void prepare(Map conf, TopologyContext context,
OutputCollector collector) {
        this.collector = collector;
    }

    public void execute(Tuple tuple) {}

    public void cleanup() {}

    public void declareOutputFields(OutputFieldsDeclarer declarer{}
}
```

1. **prepare**方法在**bolt**开始处理消息之前调用，提供一个**Outputcollector**用来发射**tuple**。
2. **execute**方法的入参就是上一跳发过来的**tuple**，在对接收到的**tuple**进行计算后将消息发送出去或者转储。
3. **declareOutputFields**定义发送给下一跳的**tuple**中的字段。
4. **cleanup**在**bolt**被销毁的时候调用。

Bolt接口

Bolt的Ack开关

- 不启用**Ack**：

  在**execute(Tuple tuple)**方法中使用如下的接口发送消息：

  collector.emit(new Values(sentence))

- 启用**Ack**：

  方法**1**：

  - 在**execute(Tuple tuple)**方法中使用如下接口发送消息，锚定上一条**tuple**：
    collector.emit(tuple, new Values(sentence));
  - 在**emit**执行成功后执行**collector.ack()**方法，或者**emit**执行失败后执行**collector.fail()**方法，向**Acker**应答此消息的发送结果。

  方法**2**：

  - **Bolt**继承**BaseBasicBolt**，该父类会自动锚定**tuple**并且自动应答，客户端只需要调用**collector.emit(new Values(sentence))**接口发送消息就可以了。

目录

创建输入流

```
CREATE INPUT STREAM example
(
    eventId INT,
    eventDesc STRING
)
  COMMENT "this is a example of create input stream."
  SERDE SimpleSerDe
    PROPERTIES (separator = "|")
  SOURCE TCPClientInput
    PROPERTIES ( server = "127.0.0.1",port = "9999" )
  PARALLEL 2;
```

Join

```
INSERT INTO STREAM rs
    SELECT * FROM S1[RANGE 20 SECONDS BATCH]
    JOIN S2[RANGE UNBOUNDED] ON s1.id=s2.id WHERE s1.id > 5;
```

```
INSERT INTO STREAM rs
    SELECT * FROM S1[ROWS 10 SLIDE]
    LEFT JOIN S2[range today ts] ON s1.id=s2.id;
```

窗口

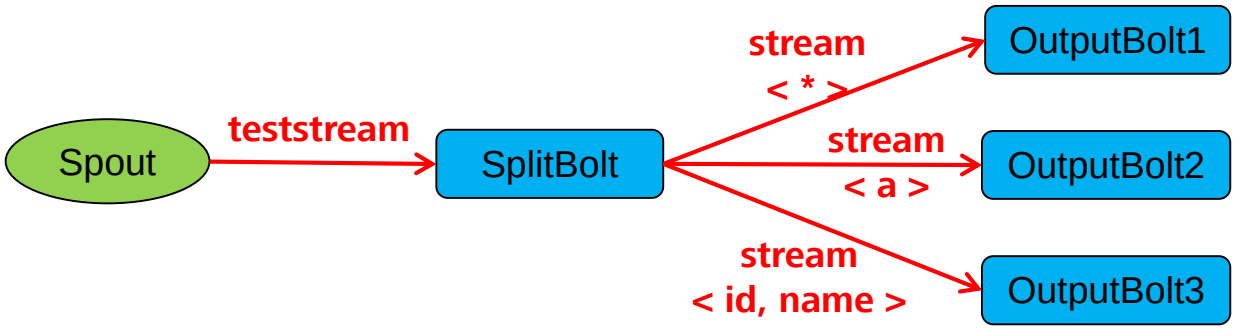
语法	名称	说明
S[ROWS N1 BATCH]	长度跳动窗	窗口内最大保存**N1**个事件，当有新事件产生的时候窗口内每攒满**N1**个事件，就过期依次。同时过期的所有事件处于同一批次。
S[RANGE T1 SLIDE]	时间滑动窗	窗口内保存最近**T1**时间范围内的数据，**T1**是一个时间单位，可以加入**Seconds**等时间单位。窗口内的事件依次过期。每个过期事件的批次都不同。
S[ROWS N1 SLIDE PARTITION BY EXP1]	分组长度滑动窗	同长度滑动窗，但是加入了分组的概念，事件归属于不同的分组，每个分组的长度为**N1**，逐个过期。
S[RANGE T1 SLIDE TRIGGER BY EXP1]	事件驱动时间滑动窗	窗口内保存最近**T1**时间单位的数据，**exp1**是一个返回值为时间类型的表达式，每次产生数据之后，都会和窗口内的数据做对比，然后将大于**T1**时间单位的数据吐出，每次只吐出一个数据。
……		

窗口

```
--按照type对窗口内数据进行分组，每组容量为10
SELECT * FROM transformEvent[ROWS 10 SLIDE PARTITION BY TYPE];
--时间排序窗，一般用来解决数据乱序问题
SELECT * FROM transformEvent[RANGE 1000 MILLISECONDS SORT BY dte];
--时间驱动滑动窗
INSERT INTO STREAM rs sum(OrderPrice),avg(OrderPrice),count(OrderPrice)
    FROM transformEvent[RANGE 10 SECONDS SLIDE TRIGGER by TS EXCLUDE now];
--保存周期为一个自然天的分组窗
INSERT INTO STREAM rs select id,name,count(id)
    FROM transformEvent[RANGE TODAY ts PARTITION BY TYPE]
    WHERE id > 5 GROUP BY TYPE HAVING id > 10;
```

Split

```
FROM teststream
    INSERT INTO STREAM s1 SELECT *
    INSERT INTO STREAM s2 SELECT a
    INSERT INTO STREAM s3 SELECT id, name WHERE id > 10
PRARLLEL 4;
```



思考题

1. **Streaming**应用开发过程中业务实现的关键步骤?
2. **Streaming**业务中如何保证消息可靠性?
3. **CQL**相比**Streaming**的**Java API**有哪些优势?

总结

- **Streaming**应用开发适用场景
- **Streaming&CQL**应用开发流程
- 分组方式
- 消息可靠性
- 常用**API**
- 应用开发实践

习题

- 判断题

1. 构造拓扑的时候各组件之间的分组方式可以随意指定。 （**T or F)**
2. **Spout**发送一条**Tuple**后**Acker**如果返回失败，则**Spout**会自动重发该条**Tuple**。（**T or F)**
3. 构造**Bolt**时，如果继承**BaseBasicBolt**，则不需要感知**ack**，会默认打开**ack**并自动响应。 （**T or F)**
4. 使用远程提交时，需要先将**storm-examples**工程下的**”src/main/java”**下的代码打成**jar**包。 （**T or F**）

习题

- 单选题

1. **Spout使用下面哪个接口发送tuple？ （ ）**

   A. open

   B. nextTuple

   C. emit

   D. execute

2. **下面哪个不是Storm支持的分组方式？ （ ）**

   A. fieldsGrouping

   B. allGrouping

   C. randomGrouping

   D. shuffleGrouping

习题

- 多选题

1. **Streaming**二次开发样例**WordCountTopology**中提供哪些提交模式？（　　）

   A. REMOTE

   B. LOCAL

   C. DISTRIBUTE

   D. CMD

2. 以下哪些属于**CQL**的提供的窗口类型？（　　）

   A.分组长度滑动窗

   B. 时间滑动窗口

   C. 长度跳动窗口

   D. 事件驱动时间滑动窗口

Thank you

www.huawei.com