## Flink的引入

这几年大数据的飞速发展，出现了很多热门的开源社区，其中著名的有 Hadoop、Storm，以及后来的 Spark，他们都有着各自专注的应用场景。Spark 掀开了内存计算的先河，也以内存为赌注，赢得了内存计算的飞速发展。Spark 的火热或多或少的掩盖了其他分布式计算的系统身影。就像 Flink，也就在这个时候默默的发展着。

在国外一些社区，有很多人将大数据的计算引擎分成了 4 代，当然，也有很多人不会认同。我们先姑且这么认为和讨论。

首先第一代的计算引擎，无疑就是 Hadoop 承载的 MapReduce。这里大家应该都不会对 MapReduce 陌生，它将计算分为两个阶段，分别为 Map 和 Reduce。对于上层应用来说，就不得不想方设法去拆分算法，甚至于不得不在上层应用实现多个 Job 的串联，以完成一个完整的算法，例如迭代计算。

由于这样的弊端，催生了支持 DAG 框架的产生。因此，支持 DAG 的框架被划分为第二代计算引擎。如 Tez 以及更上层的 Oozie。这里我们不去细究各种 DAG 实现之间的区别，不过对于当时的 Tez 和 Oozie 来说，大多还是批处理的任务。

接下来就是以 Spark 为代表的第三代的计算引擎。第三代计算引擎的特点主要是 Job 内部的 DAG 支持（不跨越 Job），以及强调的实时计算。在这里，很多人也会认为第三代计算引擎也能够很好的运行批处理的 Job。

随着第三代计算引擎的出现，促进了上层应用快速发展，例如各种迭代计算的性能以及对流计算和 SQL 等的支持。Flink 的诞生就被归在了第四代。这应该主要表现在 Flink 对流计算的支持，以及更一步的实时性上面。当然 Flink 也可以支持 Batch 的任务，以及 DAG 的运算。

首先，我们可以通过下面的性能测试初步了解两个框架的性能区别，它们都可以基于内存计算框架进行实时计算，所以都拥有非常好的计算性能。经过测试，Flink计算性能上略好。

测试环境：

1.CPU：7000个；

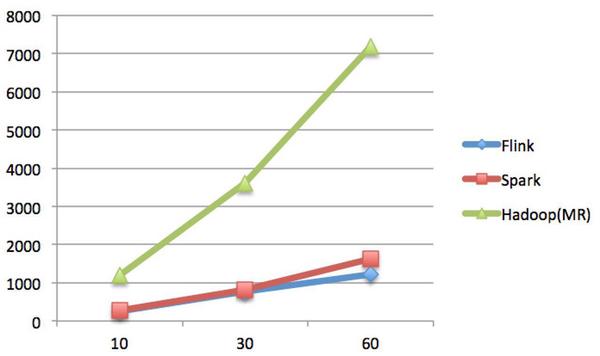
2.内存：单机128GB；

3.版本：Hadoop 2.3.0，Spark 1.4，Flink 0.9

4.数据：800MB，8GB，8TB；

5.算法：K-means：以空间中K个点为中心进行聚类，对最靠近它们的对象归类。通过迭代的方法，逐次更新各聚类中心的值，直至得到最好的聚类结果。

6.迭代：K=10，3组数据



迭代次数（纵坐标是秒，横坐标是次数）

Spark和Flink全部都运行在Hadoop YARN上，性能为Flink > Spark > Hadoop(MR)，迭代次数越多越明显，性能上，Flink优于Spark和Hadoop最主要的原因是Flink支持增量迭代，具有对迭代自动优化的功能

Flink和spark的差异

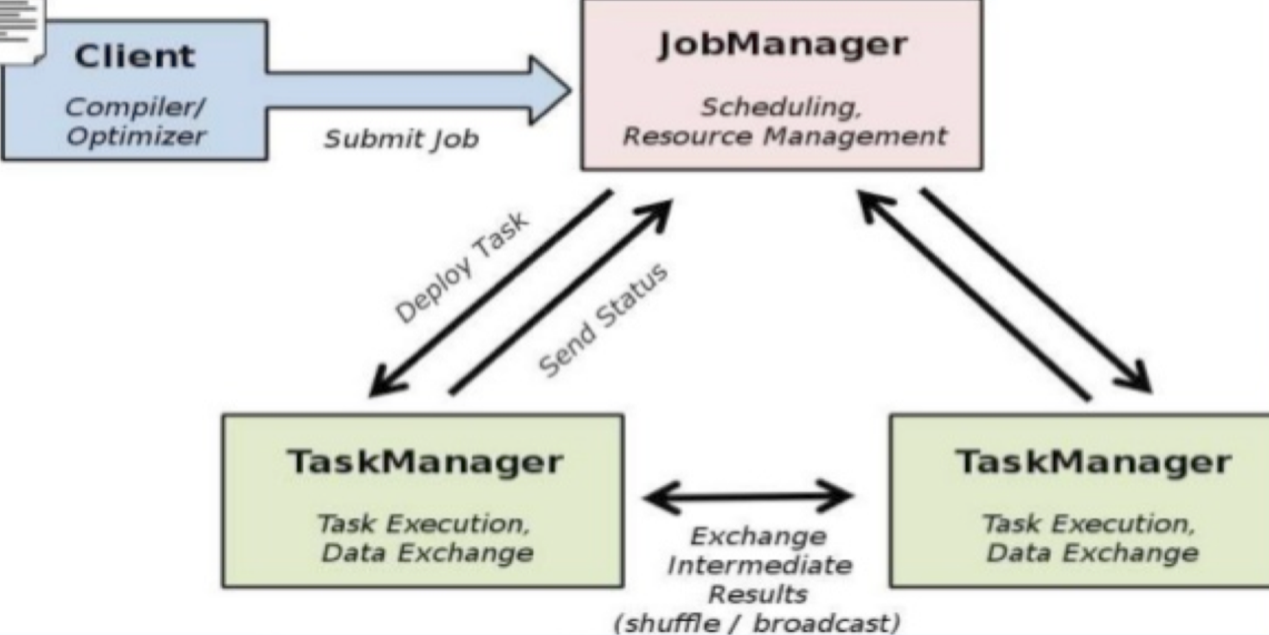
|  |  |  |
| --- | --- | --- |
|  | Spark | Flink |
| 定义 | 弹性的分布式数据集，并非真正的实时计算 | 真正的流计算，就像storm一样；  但flink同时支持有限的数据流计算（批处理） |
| 高容错 | 沉重 | 非常轻量级 |
| 内存管理 | JVM相关操作暴露给用户 | Flink在JVM中实现的是自己的内存管理 |
| 程序调优 | 只有SQL有自动优化机制 | 自动地优化一些场景，比如避免一些昂贵的操作（如shuffle和sorts），还有一些中间缓存 |

## Flink简介

很多人可能都是在 2015 年才听到 Flink 这个词，其实早在 2008 年，Flink 的前身已经是柏林理工大学一个研究性项目， 在 2014 被 Apache 孵化器所接受，然后迅速地成为了 ASF（Apache Software Foundation）的顶级项目之一。Flink 的最新版本目前已经更新到了 1.6了，在很多人感慨 Spark 的快速发展的同时，或许我们也该为 Flink 的发展速度点个赞。

Flink 是一个针对流数据和批数据的分布式处理引擎。它主要是由 Java 代码实现。目前主要还是依靠开源社区的贡献而发展。对 Flink 而言，其所要处理的主要场景就是流数据，批数据只是流数据的一个极限特例而已。再换句话说，Flink 会把所有任务当成流来处理，这也是其最大的特点。

Flink 可以支持本地的快速迭代，以及一些环形的迭代任务。并且 Flink 可以定制化内存管理。在这点，如果要对比 Flink 和 Spark 的话，Flink 并没有将内存完全交给应用层。这也是为什么 Spark 相对于 Flink，更容易出现 OOM 的原因（out of memory）。就框架本身与应用场景来说，Flink 更相似与 Storm。如果之前了解过 Storm 或者 Flume 的读者，可能会更容易理解 Flink 的架构和很多概念。下面让我们先来看下 Flink 的架构图。



我们可以了解到 Flink 几个最基础的概念，Client、JobManager 和 TaskManager。Client 用来提交任务给 JobManager，JobManager 分发任务给 TaskManager 去执行，然后 TaskManager 会心跳的汇报任务状态。看到这里，有的人应该已经有种回到 Hadoop 一代的错觉。确实，从架构图去看，JobManager 很像当年的 JobTracker，TaskManager 也很像当年的 TaskTracker。然而有一个最重要的区别就是 TaskManager 之间是是流（Stream）。其次，Hadoop 一代中，只有 Map 和 Reduce 之间的 Shuffle，而对 Flink 而言，可能是很多级，并且在 TaskManager 内部和 TaskManager 之间都会有数据传递，而不像 Hadoop，是固定的 Map 到 Reduce。

## 技术的特点（可选）

关于Flink所支持的特性，我这里只是通过分类的方式简单做一下梳理，涉及到具体的一些概念及其原理会在后面的部分做详细说明。

## 流处理特性

支持高吞吐、低延迟、高性能的流处理

支持带有事件时间的窗口（Window）操作

支持有状态计算的Exactly-once语义

支持高度灵活的窗口（Window）操作，支持基于time、count、session，以及data-driven的窗口操作

支持具有Backpressure功能的持续流模型

支持基于轻量级分布式快照（Snapshot）实现的容错

一个运行时同时支持Batch on Streaming处理和Streaming处理

Flink在JVM内部实现了自己的内存管理

支持迭代计算

支持程序自动优化：避免特定情况下Shuffle、排序等昂贵操作，中间结果有必要进行缓存

## API支持

对Streaming数据类应用，提供DataStream API

对批处理类应用，提供DataSet API（支持Java/Scala）

## Libraries支持

支持机器学习（FlinkML）

支持图分析（Gelly）

支持关系数据处理（Table）

支持复杂事件处理（CEP）

## 整合支持

支持Flink on YARN

支持HDFS

支持来自Kafka的输入数据

支持Apache HBase

支持Hadoop程序

支持Tachyon

支持ElasticSearch

支持RabbitMQ

支持Apache Storm

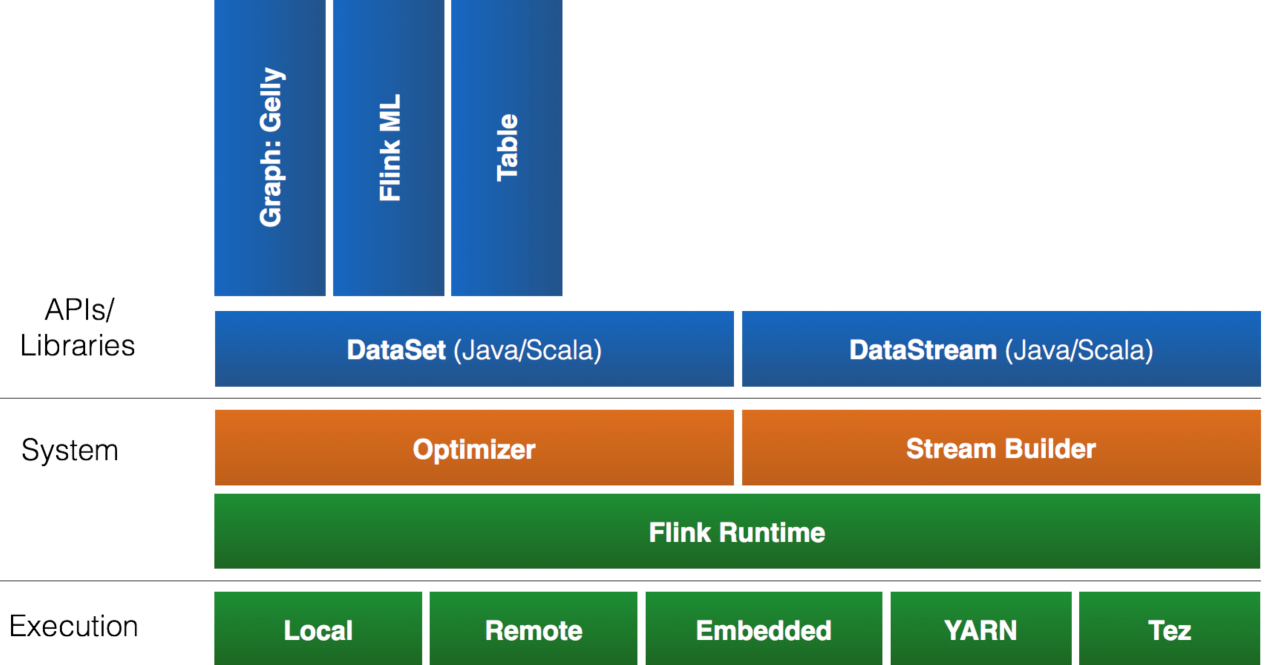
支持S3

支持XtreemFS

## Flink生态圈

一个计算框架要有长远的发展，必须打造一个完整的 Stack。不然就跟纸上谈兵一样，没有任何意义。只有上层有了具体的应用，并能很好的发挥计算框架本身的优势，那么这个计算框架才能吸引更多的资源，才会更快的进步。所以 Flink 也在努力构建自己的 Stack。

Flink 首先支持了 Scala 和 Java 的 API，Python 也正在测试中。Flink 通过 Gelly 支持了图操作，还有机器学习的 FlinkML。Table 是一种接口化的 SQL 支持，也就是 API 支持，而不是文本化的 SQL 解析和执行。对于完整的 Stack 我们可以参考下图。



Flink 为了更广泛的支持大数据的生态圈，其下也实现了很多 Connector 的子项目。最熟悉的，当然就是与 Hadoop HDFS 集成。其次，Flink 也宣布支持了 Tachyon、S3 以及 MapRFS。不过对于 Tachyon 以及 S3 的支持，都是通过 Hadoop HDFS 这层包装实现的，也就是说要使用 Tachyon 和 S3，就必须有 Hadoop，而且要更改 Hadoop 的配置（core-site.xml）。如果浏览 Flink 的代码目录，我们就会看到更多 Connector 项目，例如 Flume 和 Kafka。

## Standalone集群安装

下载方式：

直接上官网下载

上官网复制链接地址：wget 粘贴链接

### 上传安装包到linux系统

使用rz命令

### 解压

tar –zxvf flink-1.5.0-bin-hadoop24-scala\_2.11.tgz

### 修改配置文件

vim conf/flink-conf.yaml

|  |
| --- |
| jobmanager.rpc.address: hadoop01  jobmanager.rpc.port: 6123  jobmanager.heap.mb: 1024  taskmanager.heap.mb: 1024  taskmanager.numberOfTaskSlots: 2  taskmanager.memory.preallocate: false  parallelism.default: 1  jobmanager.web.port: 8081  taskmanager.tmp.dirs: /export/servers/flink-1.5.0/tmp |

配置参数解释：

|  |
| --- |
| jobmanager.rpc.address: localhost JobManager的外部地址，它是分布式系统的主/协调器（DEFAULT：localhost）设置成你master节点的IP地址  jobmanager.rpc.port: 6123 JobManager的端口号（DEFAULT：6123）  jobmanager.heap.mb: 1024 JobManager的默认JVM堆大小（以兆字节为单位）  taskmanager.heap.mb: 1024 用于TaskManagers的JVM堆大小（以兆字节为单位）  taskmanager.numberOfTaskSlots: 1 每台机器可用的CPU数量（默认值：1）  taskmanager.memory.preallocate: false 是否进行预分配内存，默认不进行预分配，这样在我们不使用flink集群时候不会占用集群资源  parallelism.default: 1 指定程序的默认并行度  jobmanager.web.port: 8081 JobManager的Web界面的端口（默认：8081）  taskmanager.tmp.dirs：临时文件的目录 |

### 启动flink集群

方式一：

#### 添加一个JobManager

bin/jobmanager.sh **((**start|start-foreground**)** cluster**)**|stop|stop-all

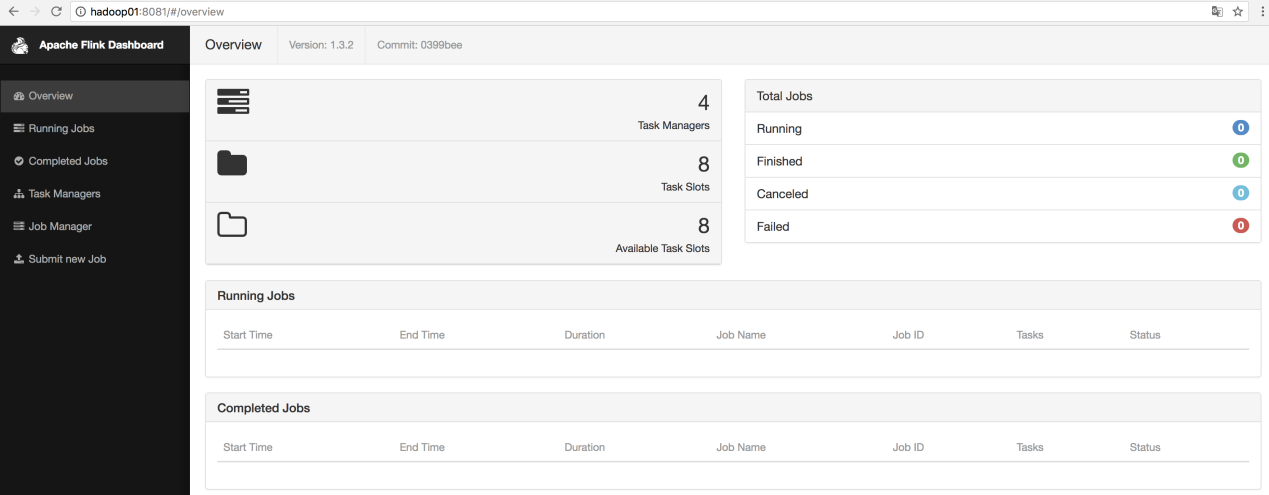
#### 添加一个TaskManager

bin/taskmanager.sh start|start-foreground|stop|stop-all

方式二：

bin/start-cluster.sh

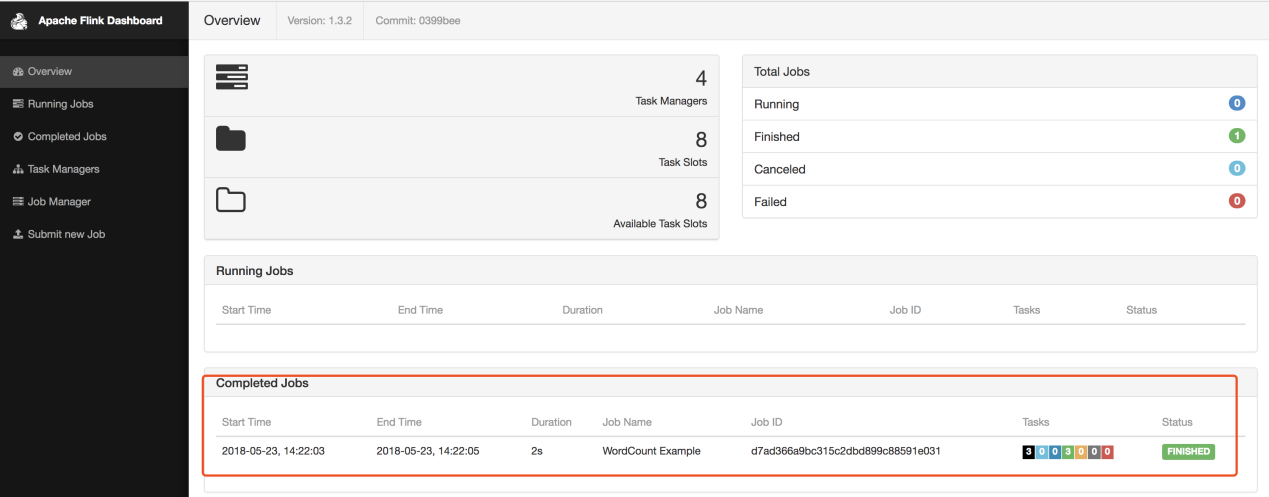
bin/stop-cluster.sh

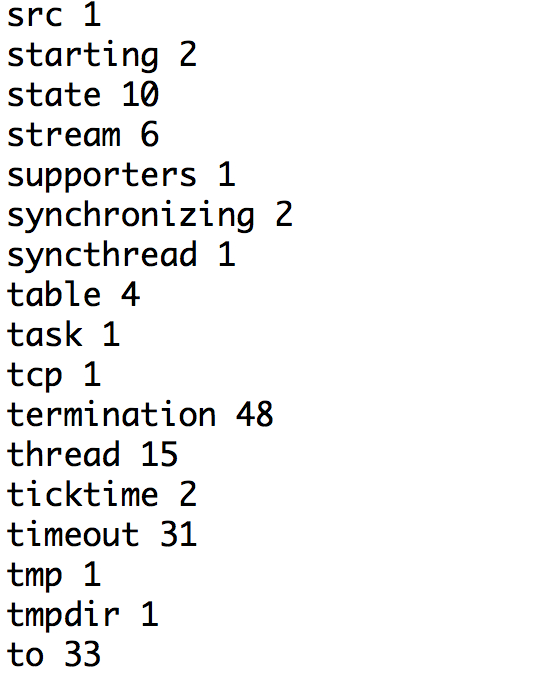


### 运行测试任务

bin/flink run /export/servers/flink-1.5.0/examples/batch/WordCount.jar --input /export/servers/zookeeper.out --output /export/servers/flink\_data







## 集群的HA高可用

对于一个企业级的应用，稳定性是首要要考虑的问题，然后才是性能，因此 HA 机制是必不可少的；

和 Hadoop 一代一样，从架构中我们可以很明显的发现 JobManager 有明显的单点问题（SPOF，single point of failure）。 JobManager 肩负着任务调度以及资源分配，一旦 JobManager 出现意外，其后果可想而知。Flink 对 JobManager HA 的处理方式，原理上基本和 Hadoop 一样；

对于 Standalone 来说，Flink 必须依赖于 Zookeeper 来实现 JobManager 的 HA（Zookeeper 已经成为了大部分开源框架 HA 必不可少的模块）。在 Zookeeper 的帮助下，一个 Standalone 的 Flink 集群会同时有多个活着的 JobManager，其中只有一个处于工作状态，其他处于 Standby 状态。当工作中的 JobManager 失去连接后（如宕机或 Crash），Zookeeper 会从 Standby 中选举新的 JobManager 来接管 Flink 集群。

(当然，对于flink的集群模式来说，除了standalone外，还有yarn cluster模式，这种模式的在hadoop节点的HA处搭建)

修改配置：

### 修改conf/flink-conf.yaml

vim conf/flink-conf.yaml

|  |
| --- |
| jobmanager.rpc.address: hadoop01【注意。HA的需要按照机器分配】  jobmanager.rpc.port: 6123  jobmanager.heap.mb: 1024  taskmanager.heap.mb: 1024  taskmanager.numberOfTaskSlots: 2  taskmanager.memory.preallocate: false  parallelism.default: 1  jobmanager.web.port: 8081  taskmanager.tmp.dirs: /export/servers/flink-1.5.0/tmp  #开启HA  state.backend: filesystem  state.backend.fs.checkpointdir: hdfs://hadoop01:9000/flink-checkpoints  high-availability: zookeeper  high-availability.storageDir: hdfs://hadoop01:9000/flink/ha/  high-availability.zookeeper.quorum: hadoop01:2181,hadoop02:2181,hadoop03:2181  high-availability.zookeeper.client.acl: open |

HA参数解释：

|  |
| --- |
| state.backend 启用检查点，支持两种后端备份点：  jobmanager：内存状态，备份到JobManager的 ZooKeeper的内存。应仅用于最小状态（Kafka偏移量）或测试和本地调试。  filesystem：状态在TaskManager的内存中，并且状态快照存储在文件系统中。支持Flink支持的所有文件系统，例如HDFS，S3 ...  state.backend.fs.checkpointdir：用于将检查点存储在Flink支持的文件系统中的目录。注意：状态后端必须可以从JobManager访问，file://仅用于本地设置  high-availability: zookeeper 定义用于群集执行的高可用性模式  high-availability.storageDir  用于存储JobManager元数据的目录; 这是持久的状态后端，只有一个指向这个状态的指针存储在ZooKeeper中。完全像检查点目录一样，它必须可以从JobManager访问  high-availability.zookeeper.quorum zookeeper的地址 |

### 修改conf/zoo.cfg[zookeeper是不存在]

|  |
| --- |
| # The number of milliseconds of each tick  tickTime=2000  # The number of ticks that the initial synchronization phase can take  initLimit=10  # The number of ticks that can pass between sending a request and getting an acknowledgement  syncLimit=5  # The directory where the snapshot is stored.  # dataDir=/tmp/zookeeper  # The port at which the clients will connect  clientPort=2181  # ZooKeeper quorum peers  server.1=hadoop01:2888:3888  server.2=hadoop02:2888:3888  server.3=hadoop03:2888:3888 |

### 修改conf/masters

|  |
| --- |
| hadoop01:8081  hadoop02:8082 |

### 修改conf/slave

|  |
| --- |
| hadoop01  hadoop02  hadoop03 |

### 启动HA的flink

1：启动zookeeper

|  |
| --- |
| Bin/zkServer.sh start (所有的zookeeper确保启动成功) |

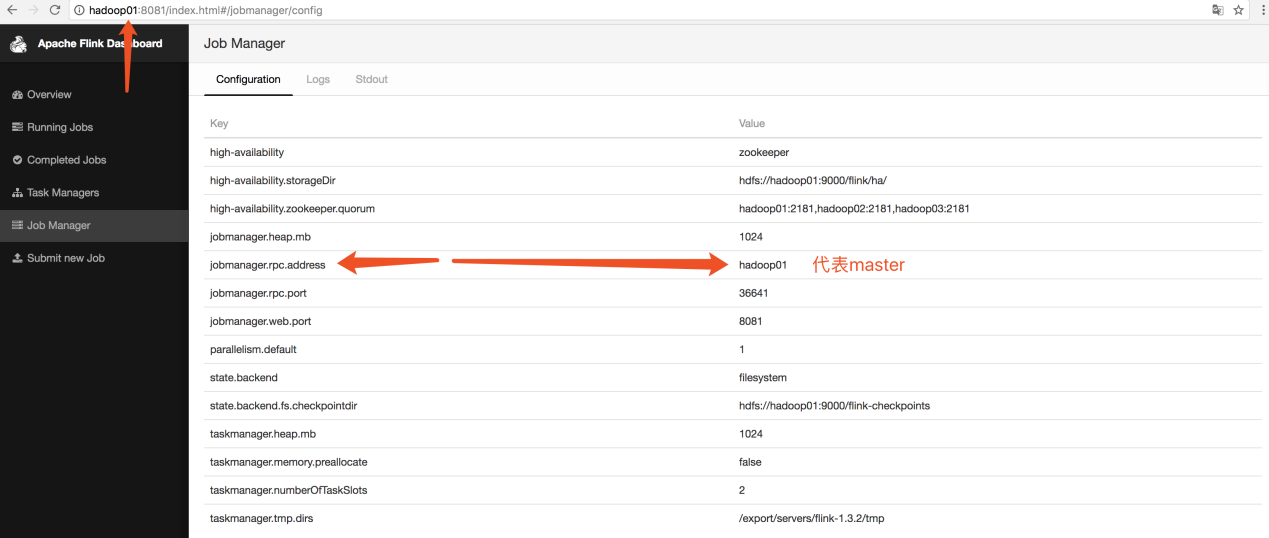
2：启动hdfs（检查点和元数据信息存储在了hdfs）

|  |
| --- |
| Start-dfs.sh |

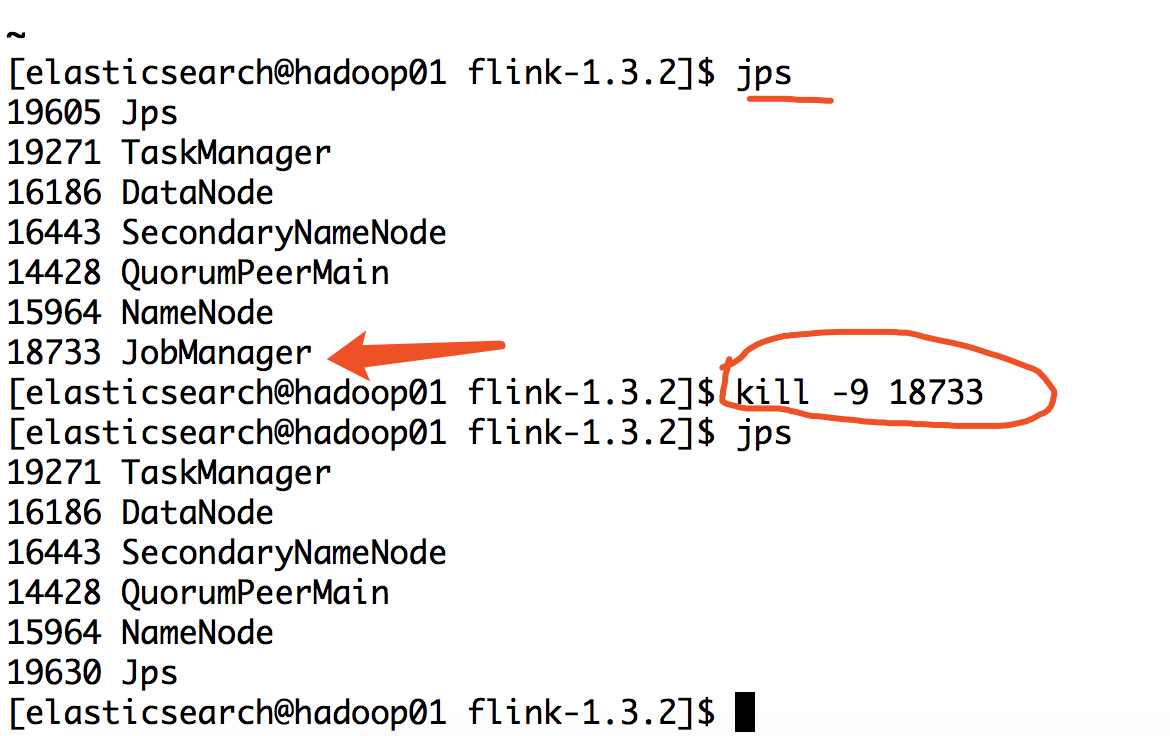
3：启动flink

|  |
| --- |
| bin/start-cluster.sh |

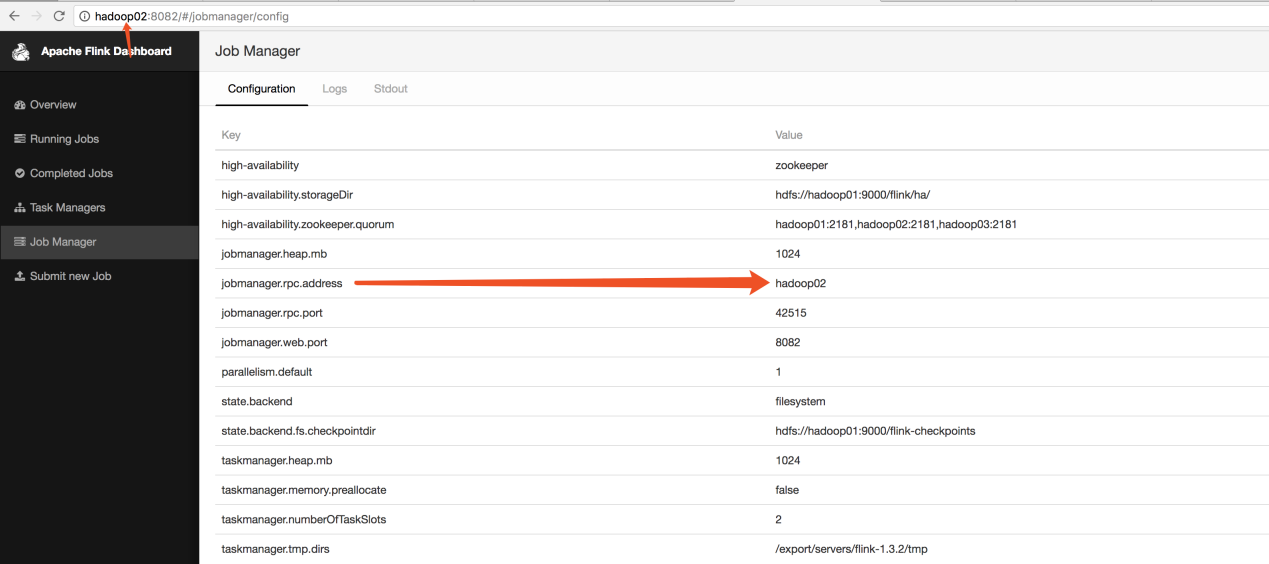
执行成功后，测试HA：



模拟突发宕机：



此时hadoop01这台机器的jobmanager出问题，然后访问另外节点的页面：hadoop02:8082



如果IP切换陈宫，代表当前的HA搭建完毕；

## Flink运行在yarn上

在一个企业中，为了最大化的利用集群资源，一般都会在一个集群中同时运行多种类型的 Workload。因此 Flink 也支持在 Yarn 上面运行；

**flink on yarn的前提是：hdfs、yarn均启动**

### 修改hadoop的配置参数：

vim etc/hadoop/yarn-site.xml

添加：

|  |
| --- |
| <property>  <name>yarn.nodemanager.vmem-check-enabled</name>  <value>false</value>  </property> |

是否启动一个线程检查每个任务正使用的虚拟内存量，如果任务超出分配值，则直接将其杀掉，默认是true。

在这里面我们需要关闭，因为对于flink使用yarn模式下，很容易内存超标，这个时候yarn会自动杀掉job

### 修改全局变量/etc/profile：

添加：

export HADOOP\_CONF\_DIR=/export/servers/hadoop/etc/Hadoop

YARN\_CONF\_DIR或者HADOOP\_CONF\_DIR必须将环境变量设置为读取YARN和HDFS配置

### 使用flink on yarn提交任务：

在YARN上启动一个Flink主要有两种方式：

(1)、启动一个YARN session(Start a long-running Flink cluster on YARN)；

(2)、直接在YARN上提交运行Flink作业(Run a Flink job on YARN)

#### 第一种方式：YARN session

这种模式下会启动yarn session，并且会启动Flink的两个必要服务：JobManager和TaskManagers，然后你可以向集群提交作业。同一个Session中可以提交多个Flink作业。**需要注意的是**，这种模式下Hadoop的版本至少是2.2，而且必须安装了HDFS（因为启动YARN session的时候会向HDFS上提交相关的jar文件和配置文件）

**通过./bin/yarn-session.sh脚本启动YARN Session**

脚本可以携带的参数：

|  |
| --- |
| Usage:  Required  -n,--container <arg> Number of YARN container to allocate (=Number of Task Managers)  Optional  -D <arg> Dynamic properties  **-d,--detached Start detached**  -id,--applicationId <arg> Attach to running YARN session  -j,--jar <arg> Path to Flink jar file  -jm,--jobManagerMemory <arg> Memory for JobManager Container [in MB]  -n,--container <arg> Number of YARN container to allocate (=Number of Task Managers)  -nm,--name <arg> Set a custom name for the application on YARN  -q,--query Display available YARN resources (memory, cores)  -qu,--queue <arg> Specify YARN queue.  -s,--slots <arg> Number of slots per TaskManager  -st,--streaming Start Flink in streaming mode  -t,--ship <arg> Ship files in the specified directory (t for transfer)  -tm,--taskManagerMemory <arg> Memory per TaskManager Container [in MB]  -z,--zookeeperNamespace <arg> Namespace to create the Zookeeper sub-paths for high availability mode |

注意：

如果不想让Flink YARN客户端始终运行，那么也可以启动分离的 YARN会话。该参数被称为-d或--detached。

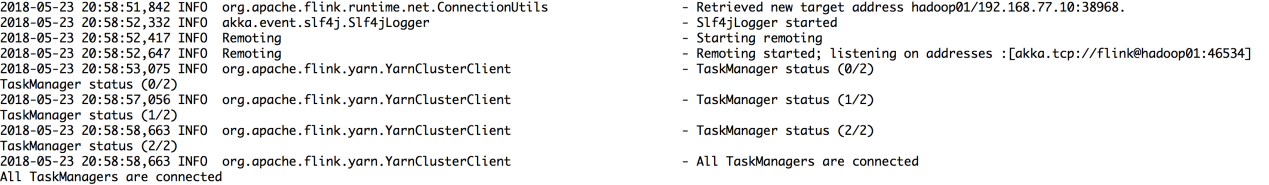
在这种情况下，Flink YARN客户端只会将Flink提交给群集，然后关闭它自己

启动：

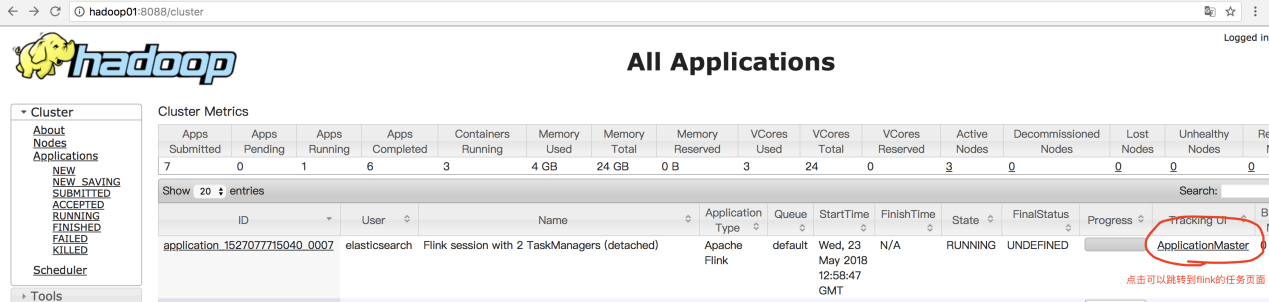
**bin/yarn-session.sh -n 2 -tm 800 -s 2**

上面的命令的意思是，同时向Yarn申请3个container（**即便只申请了两个，因为ApplicationMaster和Job Manager有一个额外的容器。一旦将Flink部署到YARN群集中，它就会显示Job Manager的连接详细信息。**），其中 2 个 Container 启动 TaskManager（-n 2），每个 TaskManager 拥有两个 Task Slot（-s 2），并且向每个 TaskManager 的 Container 申请 800M 的内存，以及一个ApplicationMaster（Job Manager）。

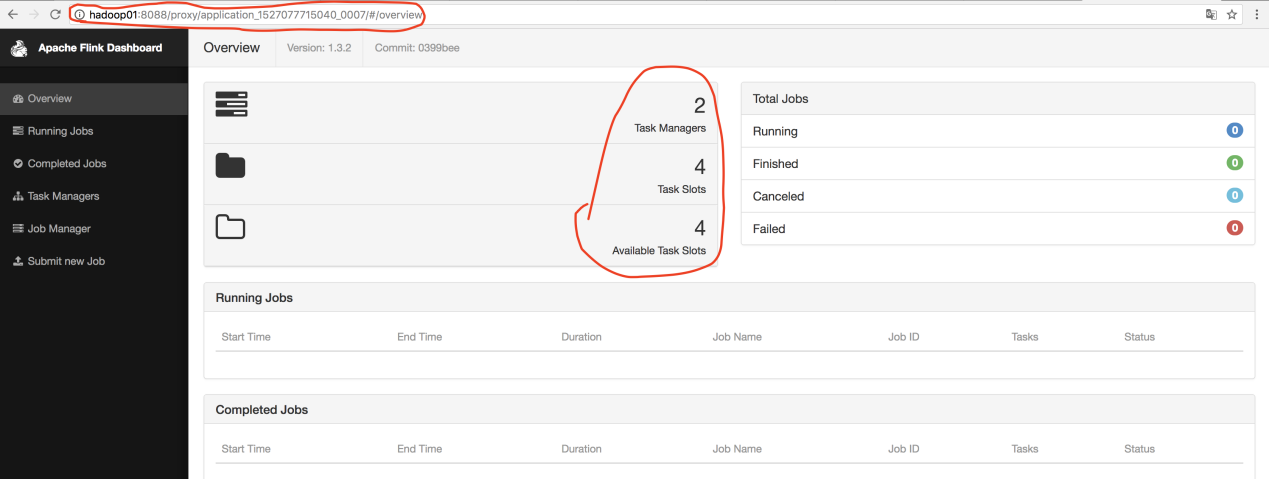
启动成功之后，控制台显示：



去yarn页面：ip:8088可以查看当前提交的flink session



点击ApplicationMaster进入任务页面：

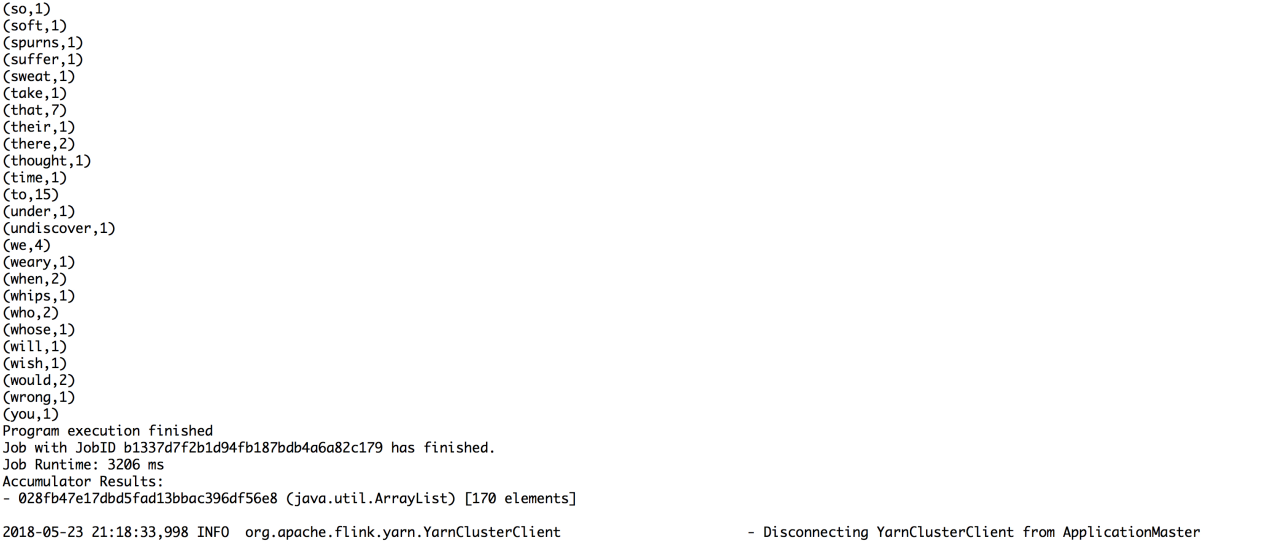


上面的页面就是使用：**yarn-session.sh**提交后的任务页面；

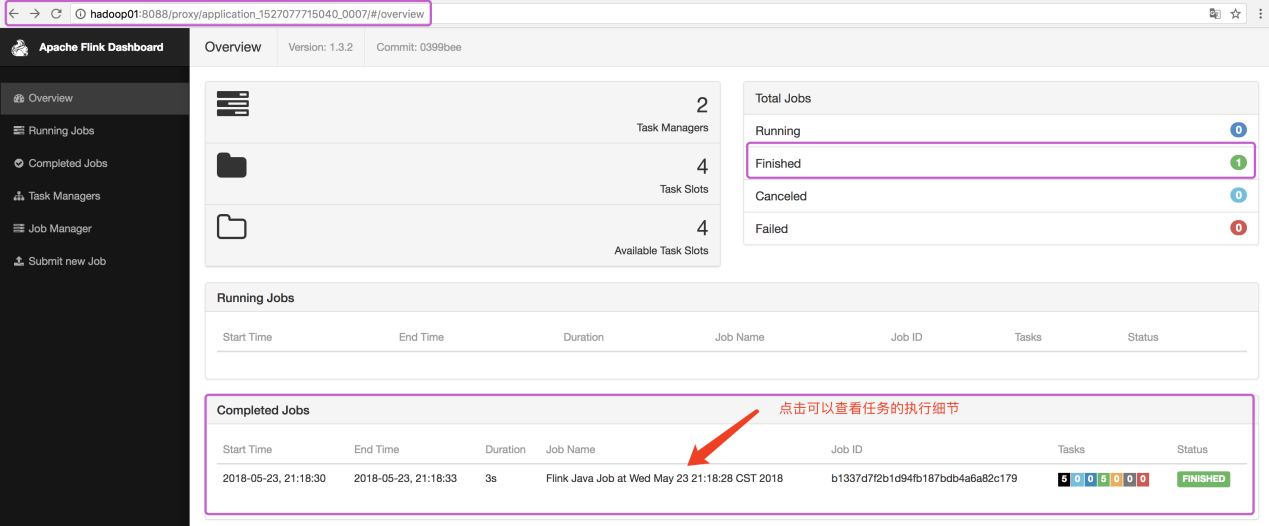
然后使用flink提交任务：

bin/flink run examples/batch/WordCount.jar

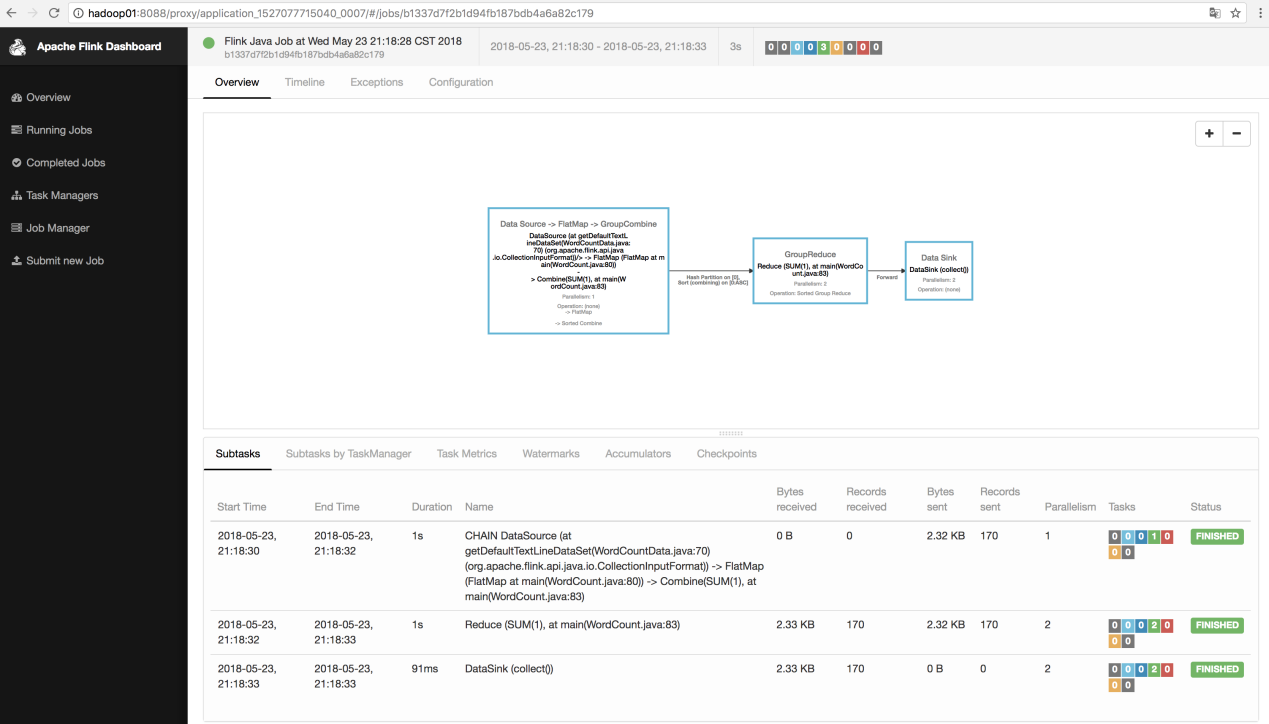
在控制台中可以看到wordCount.jar计算出来的任务结果；



在**yarn-session.sh**提交后的任务页面中也可以观察到当前提交的任务：



点击查看任务细节：



停止当前任务：

1：CTRL+C

2：stop命令

3：yarn application -kill application\_1527077715040\_0007

##### 分离的YARN会话

如果不想让Flink YARN客户端始终运行，那么也可以启动分离的 YARN会话。该参数被称为-d或--detached。

在这种情况下，Flink YARN客户端只会将Flink提交给群集，然后关闭它自己。请注意，在这种情况下，无法使用Flink停止YARN会话。

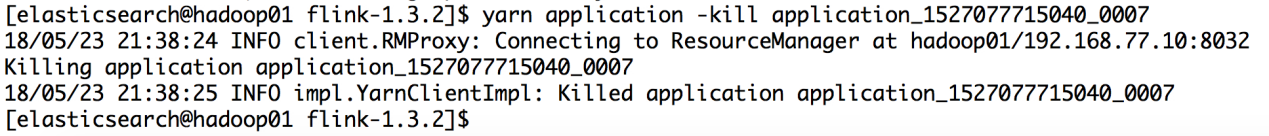
使用YARN实用程序（yarn application -kill <appId>）停止YARN会话

通过分离yarn会话来执行：

**bin/yarn-session.sh -n 2 -tm 800 -s 2 -d**

关闭：

yarn application -kill application\_1527077715040\_0007



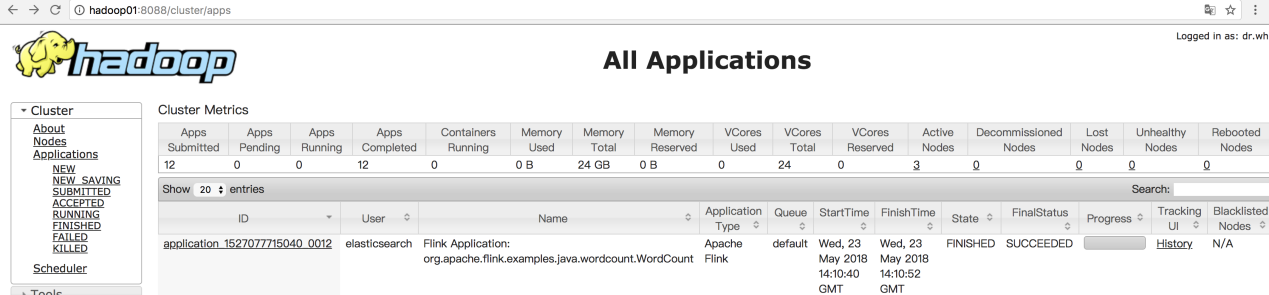
#### 第二种方式：在YARN上运行一个Flink作业

上面的YARN session是在Hadoop YARN环境下启动一个Flink cluster集群，里面的资源是可以共享给其他的Flink作业。我们还可以在YARN上启动一个Flink作业，这里我们还是使用./bin/flink，但是不需要事先启动YARN session：

|  |
| --- |
| bin/flink run -m **yarn-cluster** -yn 2 ./examples/batch/WordCount.jar |

以上命令在参数前加上y前缀，-yn表示TaskManager个数

在8088页面观察：



##### 停止yarn-cluster

yarn application -kill application的ID

注意：

|  |
| --- |
| 在创建集群的时候，集群的配置参数就写好了，但是往往因为业务需要，要更改一些配置参数，这个时候可以不必因为一个实例的提交而修改conf/flink-conf.yaml;  可以通过：-D <arg> Dynamic properties  来覆盖原有的配置信息：比如：  -Dfs.overwrite-files=true -Dtaskmanager.network.numberOfBuffers=16368 |

# Flink应用开发

flink和spark类似，也是一种一站式处理的框架；既可以进行批处理（DataSet），也可以进行实时处理（DataStream）

## 使用maven导入相关依赖

|  |
| --- |
| <**properties**>  <**maven.compiler.source**>1.8</**maven.compiler.source**>  <**maven.compiler.target**>1.8</**maven.compiler.target**>  <**encoding**>UTF-8</**encoding**>  <**scala.version**>2.11.2</**scala.version**>  <**scala.compat.version**>2.11</**scala.compat.version**>  <**hadoop.version**>2.6.2</**hadoop.version**>  <**flink.version**>1.5.0</**flink.version**> </**properties**>  <**dependencies**>  <**dependency**>  <**groupId**>org.scala-lang</**groupId**>  <**artifactId**>scala-library</**artifactId**>  <**version**>${scala.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-streaming-scala\_2.11</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-scala\_2.11</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-clients\_2.11</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table\_2.11</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.apache.hadoop</**groupId**>  <**artifactId**>hadoop-client</**artifactId**>  <**version**>${hadoop.version}</**version**>  </**dependency**>  <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>5.1.38</**version**>  </**dependency**>  <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>fastjson</**artifactId**>  <**version**>1.2.22</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-connector-kafka-0.9\_2.11</**artifactId**>  <**version**>${flink.version}</**version**>  </**dependency**> </**dependencies**> |

## DateSet开发

### 开发流程

1. 获得一个execution environment，
2. 加载/创建初始数据，
3. 指定这些数据的转换，
4. 指定将计算结果放在哪里，
5. 触发程序执行

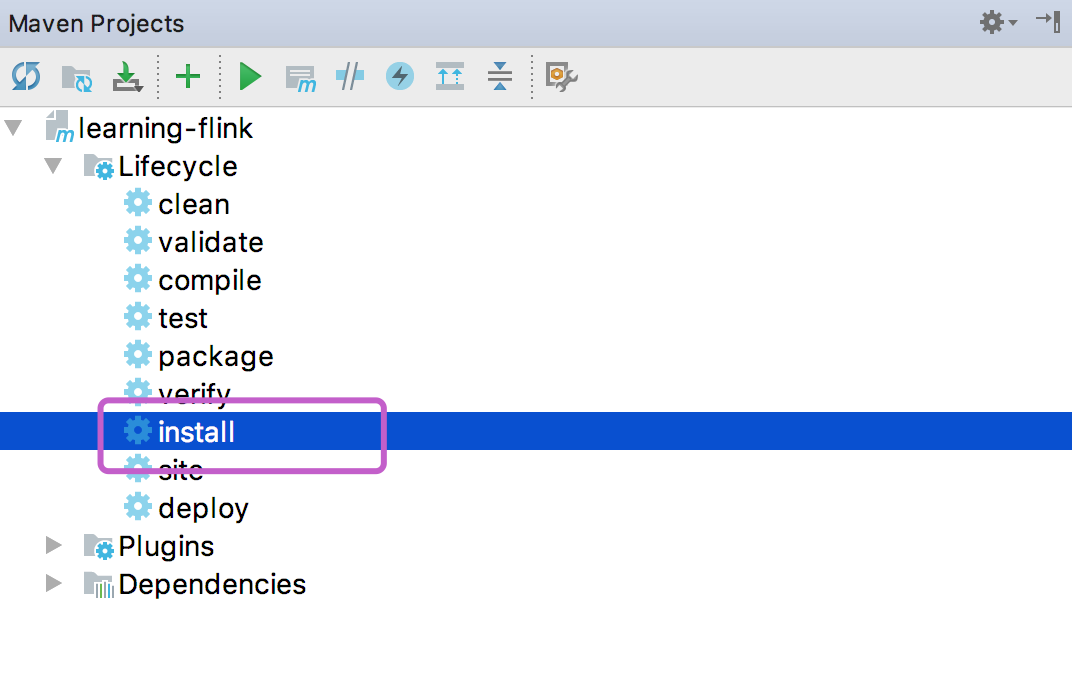
例子

|  |
| --- |
| **object** DataSet\_WordCount {  **def** main(args: Array[String]) {  *//****TODO 初始化环境* val** env = ExecutionEnvironment.*getExecutionEnvironment  //****TODO 加载/创建初始数据* val** text = env.fromElements(  **"Who's there?"**,  **"I think I hear them. Stand, ho! Who's there?"**)  *//****TODO 指定这些数据的转换* val** split\_words = text.flatMap(line => line.toLowerCase().split(**"\\W+"**))  **val** filter\_words = split\_words.filter(x=> x.nonEmpty)  **val** map\_words = filter\_words.map(x=> (x,1))  **val** groupBy\_words = map\_words.groupBy(0)  **val** sum\_words = groupBy\_words.sum(1)  *//****todo 指定将计算结果放在哪里*** *// sum\_words.setParallelism(1)//汇总结果* sum\_words.writeAsText(args(0))*//"/Users/niutao/Desktop/flink.txt"  //****TODO 触发程序执行*** env.execute(**"DataSet wordCount"**)  } } |

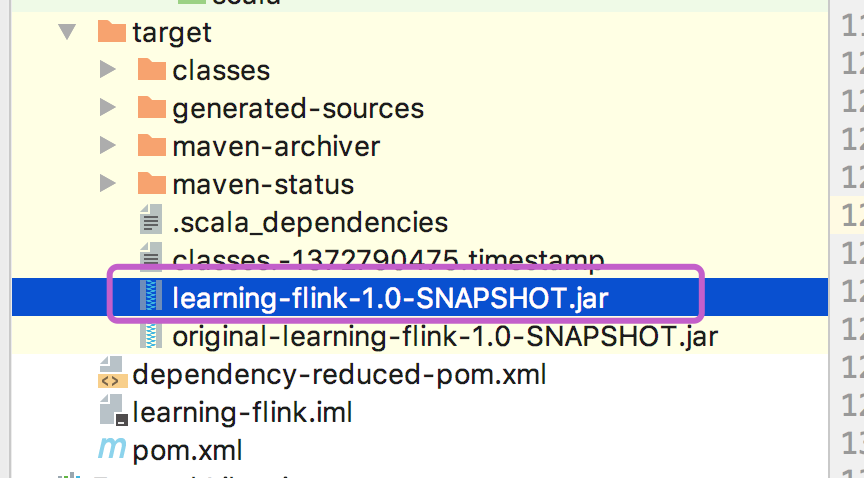
### 将程序打包，提交到yarn

添加maven打包插件

|  |
| --- |
| <**build**>  <**sourceDirectory**>src/main/java</**sourceDirectory**>  <**testSourceDirectory**>src/test/scala</**testSourceDirectory**>  <**plugins**>   <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-compiler-plugin</**artifactId**>  <**version**>2.5.1</**version**>  <**configuration**>  <**source**>1.7</**source**>  <**target**>1.7</**target**>  *<!--<encoding>${project.build.sourceEncoding}</encoding>-->* </**configuration**>  </**plugin**>   <**plugin**>  <**groupId**>net.alchim31.maven</**groupId**>  <**artifactId**>scala-maven-plugin</**artifactId**>  <**version**>3.2.0</**version**>  <**executions**>  <**execution**>  <**goals**>  <**goal**>compile</**goal**>  <**goal**>testCompile</**goal**>  </**goals**>  <**configuration**>  <**args**>  *<!--<arg>-make:transitive</arg>-->* <**arg**>-dependencyfile</**arg**>  <**arg**>${project.build.directory}/.scala\_dependencies</**arg**>  </**args**>   </**configuration**>  </**execution**>  </**executions**>  </**plugin**>  <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-surefire-plugin</**artifactId**>  <**version**>2.18.1</**version**>  <**configuration**>  <**useFile**>false</**useFile**>  <**disableXmlReport**>true</**disableXmlReport**>  <**includes**>  <**include**>\*\*/\*Test.\*</**include**>  <**include**>\*\*/\*Suite.\*</**include**>  </**includes**>  </**configuration**>  </**plugin**>   <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-shade-plugin</**artifactId**>  <**version**>2.3</**version**>  <**executions**>  <**execution**>  <**phase**>package</**phase**>  <**goals**>  <**goal**>shade</**goal**>  </**goals**>  <**configuration**>  <**filters**>  <**filter**>  <**artifact**>\*:\*</**artifact**>  <**excludes**>  *<!--  zip -d learn\_spark.jar META-INF/\*.RSA META-INF/\*.DSA META-INF/\*.SF  -->* <**exclude**>META-INF/\*.SF</**exclude**>  <**exclude**>META-INF/\*.DSA</**exclude**>  <**exclude**>META-INF/\*.RSA</**exclude**>  </**excludes**>  </**filter**>  </**filters**>  <**transformers**>  <**transformer implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer"**>  <**mainClass**>com.itcast.DEMO.WordCount</**mainClass**>  </**transformer**>  </**transformers**>  </**configuration**>  </**execution**>   </**executions**>  </**plugin**>  </**plugins**> </**build**> |



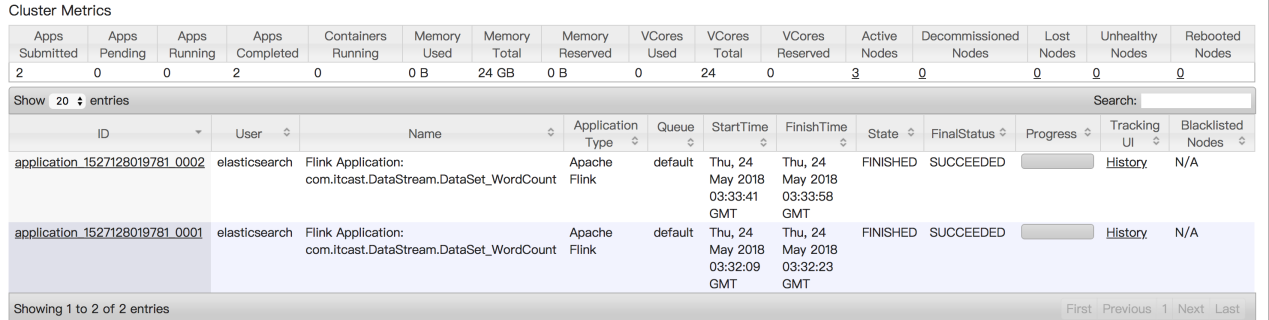
执行成功后，在target文件夹中生成jar包；



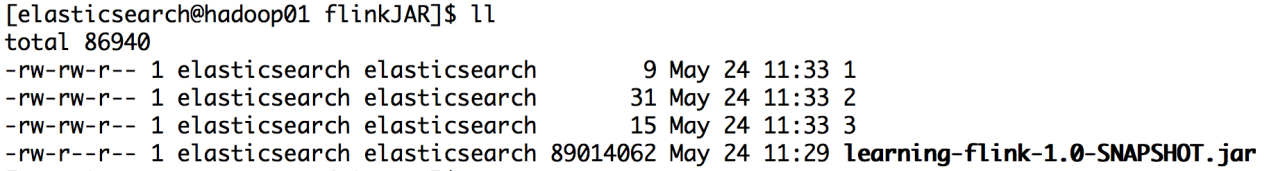
使用rz命令上传jar包，然后执行程序:

|  |
| --- |
| bin/flink run -m yarn-cluster -yn 2 /home/elasticsearch/flinkjar/itcast\_learn\_flink-1.0-SNAPSHOT.jar com.itcast.DEMO.WordCount |

在yarn的8088页面可以观察到提交的程序：



去/export/servers/flink-1.3.2/flinkJAR文件夹下可以找到输出的运行结果：



### DateSet的Transformation

| **Transformation** | **Description** |
| --- | --- |
| **Map** | Takes one element and produces one element.  data**.**map **{** x **=>** x**.**toInt **}** |
| **FlatMap** | Takes one element and produces zero, one, or more elements.  data**.**flatMap **{** str **=>** str**.**split**(**" "**)** **}** |
| **MapPartition** | Transforms a parallel partition in a single function call. The function get the partition as an `Iterator` and can produce an arbitrary number of result values. The number of elements in each partition depends on the degree-of-parallelism and previous operations.  data**.**mapPartition **{** in **=>** in map **{** **(\_,** 1**)** **}** **}** |
| **Filter** | Evaluates a boolean function for each element and retains those for which the function returns true. **IMPORTANT:** The system assumes that the function does not modify the element on which the predicate is applied. Violating this assumption can lead to incorrect results.  data**.**filter **{** **\_** **>** 1000 **}** |
| **Reduce** | Combines a group of elements into a single element by repeatedly combining two elements into one. Reduce may be applied on a full data set, or on a grouped data set.  data**.**reduce **{** **\_** **+** **\_** **}** |
| **ReduceGroup** | Combines a group of elements into one or more elements. ReduceGroup may be applied on a full data set, or on a grouped data set.  data**.**reduceGroup **{** elements **=>** elements**.**sum **}** |
| **Aggregate** | Aggregates a group of values into a single value. Aggregation functions can be thought of as built-in reduce functions. Aggregate may be applied on a full data set, or on a grouped data set.  **val** input**:** **DataSet[(Int**, **String**, **Double)]** **=** *// [...]*  **val** output**:** **DataSet[(Int**, **String**, **Doublr)]** **=** input**.**aggregate**(SUM,** 0**).**aggregate**(MIN,** 2**);**  You can also use short-hand syntax for minimum, maximum, and sum aggregations.  **val** input**:** **DataSet[(Int**, **String**, **Double)]** **=** *// [...]*  **val** output**:** **DataSet[(Int**, **String**, **Double)]** **=** input**.**sum**(**0**).**min**(**2**)** |
| **Distinct** | Returns the distinct elements of a data set. It removes the duplicate entries from the input DataSet, with respect to all fields of the elements, or a subset of fields.  data**.**distinct**()** |
| **Join** | Joins two data sets by creating all pairs of elements that are equal on their keys. Optionally uses a JoinFunction to turn the pair of elements into a single element, or a FlatJoinFunction to turn the pair of elements into arbitrarily many (including none) elements. See the [keys section](http://doc.flink-china.org/1.2.0/dev/api_concepts.html" \l "specifying-keys) to learn how to define join keys.  *// In this case tuple fields are used as keys. "0" is the join field on the first tuple*  *// "1" is the join field on the second tuple.*  **val** result **=** input1**.**join**(**input2**).**where**(**0**).**equalTo**(**1**)**  You can specify the way that the runtime executes the join via *Join Hints*. The hints describe whether the join happens through partitioning or broadcasting, and whether it uses a sort-based or a hash-based algorithm. Please refer to the [Transformations Guide](http://doc.flink-china.org/1.2.0/dev/batch/dataset_transformations.html" \l "join-algorithm-hints) for a list of possible hints and an example. If no hint is specified, the system will try to make an estimate of the input sizes and pick the best strategy according to those estimates.  *// This executes a join by broadcasting the first data set*  *// using a hash table for the broadcasted data*  **val** result **=** input1**.**join**(**input2**,** **JoinHint.BROADCAST\_HASH\_FIRST)**  **.**where**(**0**).**equalTo**(**1**)**  Note that the join transformation works only for equi-joins. Other join types need to be expressed using OuterJoin or CoGroup. |
| **OuterJoin** | Performs a left, right, or full outer join on two data sets. Outer joins are similar to regular (inner) joins and create all pairs of elements that are equal on their keys. In addition, records of the "outer" side (left, right, or both in case of full) are preserved if no matching key is found in the other side. Matching pairs of elements (or one element and a `null` value for the other input) are given to a JoinFunction to turn the pair of elements into a single element, or to a FlatJoinFunction to turn the pair of elements into arbitrarily many (including none) elements. See the [keys section](http://doc.flink-china.org/1.2.0/dev/api_concepts.html" \l "specifying-keys) to learn how to define join keys.  **val** joined **=** left**.**leftOuterJoin**(**right**).**where**(**0**).**equalTo**(**1**)** **{**  **(**left**,** right**)** **=>**  **val** a **=** **if** **(**left **==** **null)** "none" **else** left**.**\_1  **(**a**,** right**)**  **}** |
| **CoGroup** | The two-dimensional variant of the reduce operation. Groups each input on one or more fields and then joins the groups. The transformation function is called per pair of groups. See the [keys section](http://doc.flink-china.org/1.2.0/dev/api_concepts.html" \l "specifying-keys) to learn how to define coGroup keys.  data1**.**coGroup**(**data2**).**where**(**0**).**equalTo**(**1**)** |
| **Cross** | Builds the Cartesian product (cross product) of two inputs, creating all pairs of elements. Optionally uses a CrossFunction to turn the pair of elements into a single element  **val** data1**:** **DataSet[Int]** **=** *// [...]*  **val** data2**:** **DataSet[String]** **=** *// [...]*  **val** result**:** **DataSet[(Int**, **String)]** **=** data1**.**cross**(**data2**)**  Note: Cross is potentially a **very** compute-intensive operation which can challenge even large compute clusters! It is adviced to hint the system with the DataSet sizes by using *crossWithTiny()* and *crossWithHuge()*. |
| **Union** | Produces the union of two data sets.  data**.**union**(**data2**)** |
| **Rebalance** | Evenly rebalances the parallel partitions of a data set to eliminate data skew. Only Map-like transformations may follow a rebalance transformation.  **val** data1**:** **DataSet[Int]** **=** *// [...]*  **val** result**:** **DataSet[(Int**, **String)]** **=** data1**.**rebalance**().**map**(...)** |
| **Hash-Partition** | Hash-partitions a data set on a given key. Keys can be specified as position keys, expression keys, and key selector functions.  **val** in**:** **DataSet[(Int**, **String)]** **=** *// [...]*  **val** result **=** in**.**partitionByHash**(**0**).**mapPartition **{** **...** **}** |
| **Range-Partition** | Range-partitions a data set on a given key. Keys can be specified as position keys, expression keys, and key selector functions.  **val** in**:** **DataSet[(Int**, **String)]** **=** *// [...]*  **val** result **=** in**.**partitionByRange**(**0**).**mapPartition **{** **...** **}** |
| **Custom Partitioning** | Manually specify a partitioning over the data.  *Note*: This method works only on single field keys.  **val** in**:** **DataSet[(Int**, **String)]** **=** *// [...]*  **val** result **=** in  **.**partitionCustom**(**partitioner**:** **Partitioner[K],** key**)** |
| **Sort Partition** | Locally sorts all partitions of a data set on a specified field in a specified order. Fields can be specified as tuple positions or field expressions. Sorting on multiple fields is done by chaining sortPartition() calls.  **val** in**:** **DataSet[(Int**, **String)]** **=** *// [...]*  **val** result **=** in**.**sortPartition**(**1**,** **Order.ASCENDING).**mapPartition **{** **...** **}** |
| **First-n** | Returns the first n (arbitrary) elements of a data set. First-n can be applied on a regular data set, a grouped data set, or a grouped-sorted data set. Grouping keys can be specified as key-selector functions, tuple positions or case class fields.  **val** in**:** **DataSet[(Int**, **String)]** **=** *// [...]*  *// regular data set*  **val** result1 **=** in**.**first**(**3**)**  *// grouped data set*  **val** result2 **=** in**.**groupBy**(**0**).**first**(**3**)**  *// grouped-sorted data set*  **val** result3 **=** in**.**groupBy**(**0**).**sortGroup**(**1**,** **Order.ASCENDING).**first**(**3**)** |

#### 1：map函数

#### 2：flatMap函数

|  |
| --- |
| *//初始化执行环境* **val** env: ExecutionEnvironment = ExecutionEnvironment.*getExecutionEnvironment //加载数据* **val** data = env.fromElements((**"A"** , 1) , (**"B"** , 1) , (**"C"** , 1)) *//使用trasformation加载这些数据 //****TODO map* val** map\_result = data.map(line => line.\_1+line.\_2) map\_result.print() *//****TODO flatmap* val** flatmap\_result = data.flatMap(line => line.\_1+line.\_2) flatmap\_result.print() |

练习：如下数据

|  |
| --- |
| A;B;C;D;B;D;C  B;D;A;E;D;C  A;B |

要求：统计相邻字符串出现的次数

|  |
| --- |
| **import** org.apache.flink.api.scala.{DataSet, ExecutionEnvironment} **import** org.apache.flink.streaming.api.scala.\_ */\*\*  \* Created by angel；  \*/* **object** demo { */\*\* A;B;C;D;B;D;C B;D;A;E;D;C A;B 统计相邻字符串出现的次数(A+B , 2) (B+C , 1)....  \* \*/* **def** main(args: Array[String]): Unit = {  **val** env = ExecutionEnvironment.*getExecutionEnvironment* **val** data = env.fromElements(**"A;B;C;D;B;D;C;B;D;A;E;D;C;A;B"**)  **val** map\_data: DataSet[Array[String]] = data.map(line => line.split(**";"**))  *//[A,B,C,D] ---"A,B,C,D"  //[A,B,C,D] ---> (x,1) , (y,1) -->groupBy--->sum--total* **val** tupe\_data = map\_data.flatMap{  line =>  **for**(index <- 0 until line.length-1) **yield** (line(index)+**"+"**+line(index+1) , 1)  }  **val** gropudata = tupe\_data.groupBy(0)  **val** result = gropudata.sum(1)  result.print()  } } |

#### 3：mapPartition函数

|  |
| --- |
| *//****TODO mapPartition* val** ele\_partition = elements.setParallelism(2)*//将分区设置为2* **val** partition = ele\_partition.mapPartition(line => line.map(x=> x+**"======"**))*//line是每个分区下面的数据* partition.print() |

mapPartition：是一个分区一个分区拿出来的

好处就是以后我们操作完数据了需要存储到mysql中，这样做的好处就是几个分区拿几个连接，如果用map的话，就是多少条数据拿多少个mysql的连接

#### 4：filter函数

Filter函数在实际生产中特别实用，数据处理阶段可以过滤掉大部分不符合业务的内容，可以极大减轻整体flink的运算压力

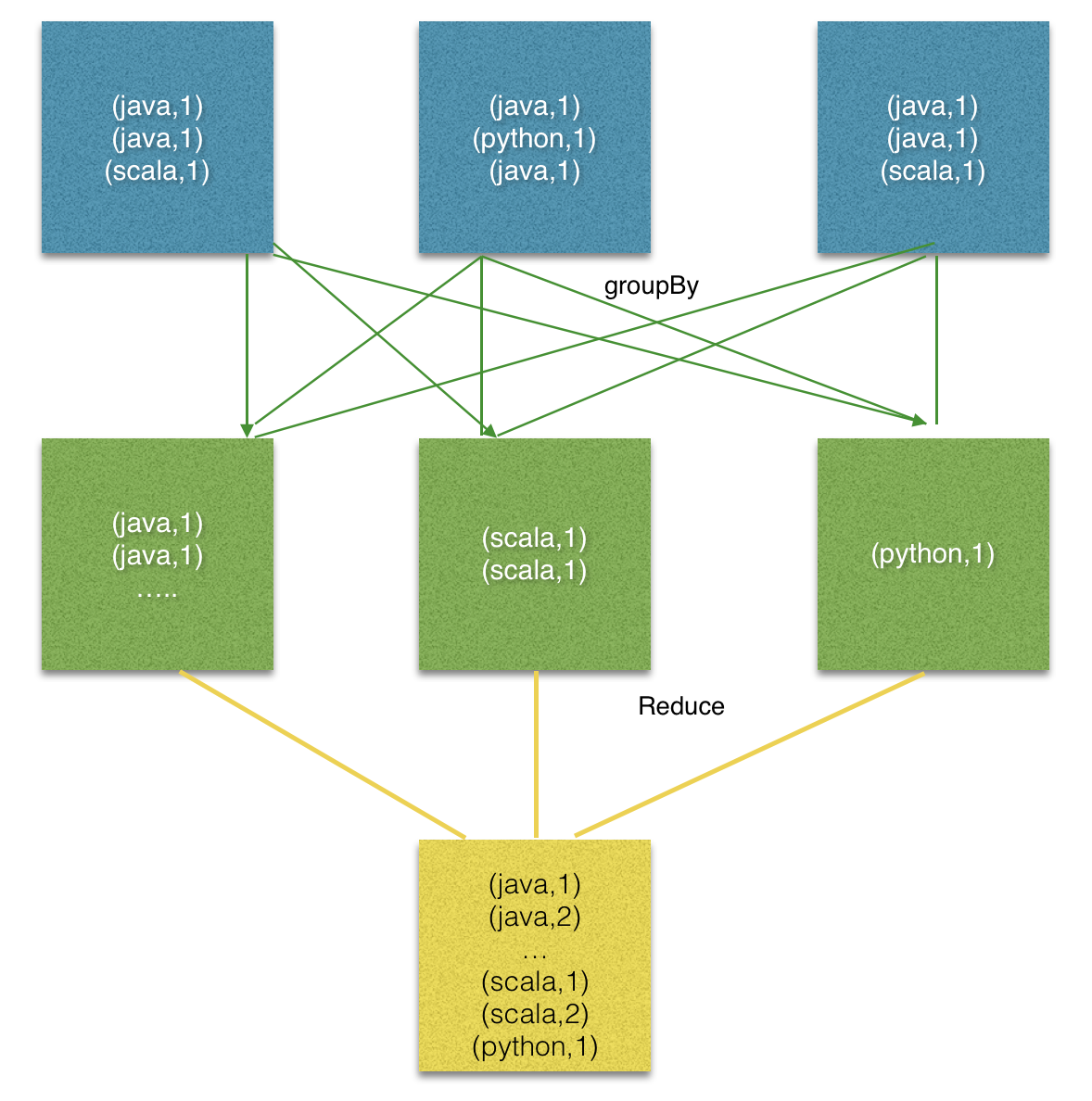
|  |
| --- |
| *//****TODO fileter* val** filter:DataSet[String] = elements.filter(line => line.contains(**"java"**))*//过滤出带java的数据* filter.print() |

#### 5：reduce函数

|  |
| --- |
| *//****TODO reduce* val** elements:DataSet[List[Tuple2[String , Int]]] = env.fromElements(*List*((**"java"** , 1) , (**"scala"** , 1) , (**"java"** , 1)))  **val** tuple\_map = elements.flatMap(x=> x)*//拆开里面的list，编程tuple* **val** group\_map = tuple\_map.groupBy(x => x.\_1)*//按照单词聚合* **val** reduce = group\_map.reduce((x,y) => (x.\_1 ,x.\_2+y.\_2)) reduce.print() |

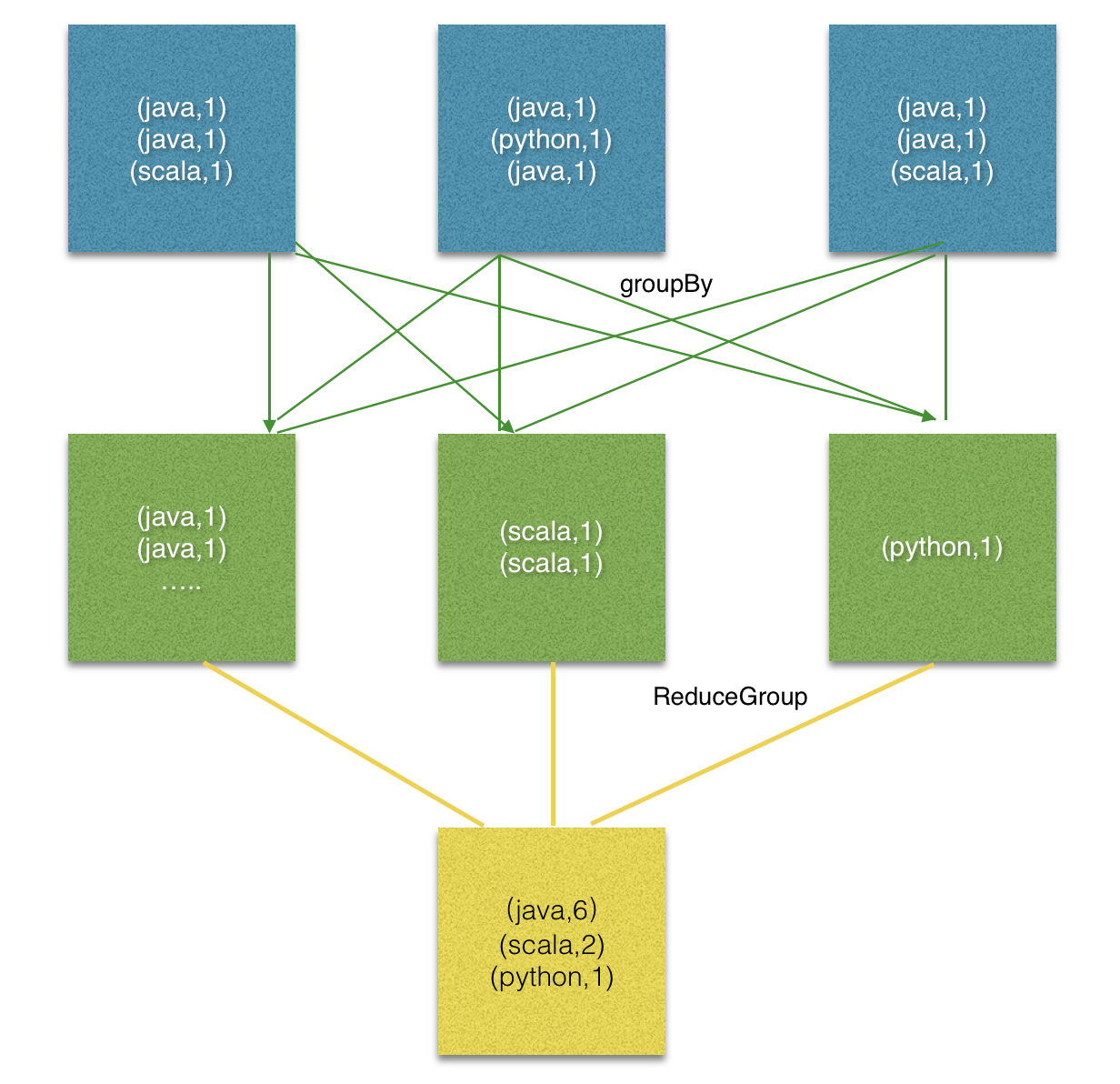
#### 6：reduceGroup

普通的reduce函数



reduceGroup是reduce的一种优化方案；

它会先分组reduce，然后在做整体的reduce；这样做的好处就是可以减少网络IO；



|  |
| --- |
| *//****TODO reduceGroup* val** elements:DataSet[List[Tuple2[String , Int]]] = env.fromElements(*List*((**"java"** , 1) ,(**"java"** , 1), (**"scala"** , 1)))  **val** tuple\_words = elements.flatMap(x=>x)  **val** group\_words = tuple\_words.groupBy(x => x.\_1)  **val** a = group\_words.reduceGroup{  (in:Iterator[(String,Int)],out:Collector[(String , Int)]) =>  **val** result = in.reduce((x, y) => (x.\_1, x.\_2+y.\_2))  out.collect(result)  }  a.print() } |

#### 7：GroupReduceFunction和GroupCombineFunction（自定义函数）

|  |
| --- |
| **import** collection.JavaConverters.\_ **class** Tuple3GroupReduceWithCombine **extends** GroupReduceFunction[( String , Int), (String, Int)] **with** GroupCombineFunction[(String, Int), (String, Int)] {  **override def** reduce(values: Iterable[(String, Int)], out: Collector[(String, Int)]): Unit = {   **for**(in <- values.asScala){  out.collect((in.\_1 , in.\_2))  }  }   **override def** combine(values: Iterable[(String, Int)], out: Collector[(String, Int)]): Unit = {  **val** map = **new** mutable.HashMap[String , Int]()  **var** num = 0  **var** s = **""  for**(in <- values.asScala){  num += in.\_2  s = in.\_1  }  out.collect((s , num))  } } |

|  |
| --- |
| *//* ***TODO GroupReduceFunction GroupCombineFunction* val** env = ExecutionEnvironment.*getExecutionEnvironment* **val** elements:DataSet[List[Tuple2[String , Int]]] = env.fromElements(*List*((**"java"** , 3) ,(**"java"** , 1), (**"scala"** , 1))) **val** collection = elements.flatMap(line => line) **val** groupDatas:GroupedDataSet[(String, Int)] = collection.groupBy(line => line.\_1) *//在reduceGroup下使用自定义的reduce和combiner函数* **val** result = groupDatas.reduceGroup(**new** Tuple3GroupReduceWithCombine()) **val** result\_sort = result.collect().sortBy(x=>x.\_1) *println*(result\_sort) |

#### 8：combineGroup

使用之前的group操作，比如：reduceGroup或者GroupReduceFuncation；这种操作很容易造成内存溢出；因为要一次性把所有的数据一步转化到位，所以需要足够的内存支撑，如果内存不够的情况下，那么需要使用combineGroup；

combineGroup在分组数据集上应用GroupCombineFunction。

GroupCombineFunction类似于GroupReduceFunction，但不执行完整的数据交换。

【注意】：使用combineGroup可能得不到完整的结果而是部分的结果

|  |
| --- |
| **import** collection.JavaConverters.\_ **class** MycombineGroup **extends** GroupCombineFunction[Tuple1[String] , (String , Int)]{  **override def** combine(iterable: Iterable[Tuple1[String]], out: Collector[(String, Int)]): Unit = {  **var** key: String = **null  var** count = 0  **for**(line <- iterable.asScala){  key = line.\_1  count += 1  }  out.collect((key, count))   } } |

|  |
| --- |
| *//****TODO combineGroup* val** input = env.fromElements(**"a"**, **"b"**, **"c"**, **"a"**).map(*Tuple1*(\_)) **val** combinedWords = input.groupBy(0).combineGroup(**new** MycombineGroup()) combinedWords.print() |

#### 9：Aggregate

在数据集上进行聚合求最值（最大值、最小值）

**Aggregate只能作用于元组上**

|  |
| --- |
| *//****TODO Aggregate* val** data = **new** mutable.MutableList[(Int, String, Double)] data.+=((1, **"yuwen"**, 89.0)) data.+=((2, **"shuxue"**, 92.2)) data.+=((3, **"yingyu"**, 89.99)) data.+=((4, **"wuli"**, 98.9)) data.+=((1, **"yuwen"**, 88.88)) data.+=((1, **"wuli"**, 93.00)) data.+=((1, **"yuwen"**, 94.3)) *// //fromCollection将数据转化成DataSet* **val** input: DataSet[(Int, String, Double)] = env.fromCollection(Random.shuffle(data)) **val** output = input.groupBy(1).aggregate(Aggregations.*MAX*, 2) output.print() |

#### 10：minBy和maxBy

|  |
| --- |
| *//****TODO MinBy / MaxBy* val** data = **new** mutable.MutableList[(Int, String, Double)] data.+=((1, **"yuwen"**, 90.0)) data.+=((2, **"shuxue"**, 20.0)) data.+=((3, **"yingyu"**, 30.0)) data.+=((4, **"wuli"**, 40.0)) data.+=((5, **"yuwen"**, 50.0)) data.+=((6, **"wuli"**, 60.0)) data.+=((7, **"yuwen"**, 70.0)) *// //fromCollection将数据转化成DataSet* **val** input: DataSet[(Int, String, Double)] = env.fromCollection(Random.shuffle(data)) **val** output: DataSet[(Int, String, Double)] = input  .groupBy(1)  *//求每个学科下的最小分数  //minBy的参数代表要求哪个字段的最小值* .minBy(2) output.print() |

#### 11：distinct去重

|  |
| --- |
| *//****TODO distinct 去重* val** data = **new** mutable.MutableList[(Int, String, Double)]  data.+=((1, **"yuwen"**, 90.0))  data.+=((2, **"shuxue"**, 20.0))  data.+=((3, **"yingyu"**, 30.0))  data.+=((4, **"wuli"**, 40.0))  data.+=((5, **"yuwen"**, 50.0))  data.+=((6, **"wuli"**, 60.0))  data.+=((7, **"yuwen"**, 70.0))  *// //fromCollection将数据转化成DataSet* **val** input: DataSet[(Int, String, Double)] = env.fromCollection(Random.shuffle(data))  **val** distinct = input.distinct(1)  distinct.print() |

#### 12：join

Flink在操作过程中，有时候也会遇到关联组合操作，这样可以方便的返回想要的关联结果，比如：

求每个班级的每个学科的最高分数

|  |
| --- |
| *//****TODO join* val** data1 = **new** mutable.MutableList[(Int, String, Double)]  *//学生学号---学科---分数* data1.+=((1, **"yuwen"**, 90.0))  data1.+=((2, **"shuxue"**, 20.0))  data1.+=((3, **"yingyu"**, 30.0))  data1.+=((4, **"yuwen"**, 40.0))  data1.+=((5, **"shuxue"**, 50.0))  data1.+=((6, **"yingyu"**, 60.0))  data1.+=((7, **"yuwen"**, 70.0))  data1.+=((8, **"yuwen"**, 20.0))  **val** data2 = **new** mutable.MutableList[(Int, String)]  *//学号 ---班级* data2.+=((1,**"class\_1"**))  data2.+=((2,**"class\_1"**))  data2.+=((3,**"class\_2"**))  data2.+=((4,**"class\_2"**))  data2.+=((5,**"class\_3"**))  data2.+=((6,**"class\_3"**))  data2.+=((7,**"class\_4"**))  data2.+=((8,**"class\_1"**))  **val** input1: DataSet[(Int, String, Double)] = env.fromCollection(Random.shuffle(data1))  **val** input2: DataSet[(Int, String)] = env.fromCollection(Random.shuffle(data2))  *//求每个班级下每个学科最高分数* **val** joindata = input2.join(input1).where(0).equalTo(0){  (input2 , input1) => (input2.\_1 , input2.\_2 , input1.\_2 , input1.\_3)  } *// joindata.print() // println("===================")* **val** aggregateDataSet = joindata.groupBy(1,2).aggregate(Aggregations.*MAX* , 3)  aggregateDataSet.print() |

#### 13：cross交叉操作

和join类似，但是这种交叉操作会产生笛卡尔积，在数据比较大的时候，是非常消耗内存的操作；

|  |
| --- |
| *//****TODO Cross 交叉操作，会产生笛卡尔积* val** data1 = **new** mutable.MutableList[(Int, String, Double)] *//学生学号---学科---分数* data1.+=((1, **"yuwen"**, 90.0)) data1.+=((2, **"shuxue"**, 20.0)) data1.+=((3, **"yingyu"**, 30.0)) data1.+=((4, **"yuwen"**, 40.0)) data1.+=((5, **"shuxue"**, 50.0)) data1.+=((6, **"yingyu"**, 60.0)) data1.+=((7, **"yuwen"**, 70.0)) data1.+=((8, **"yuwen"**, 20.0)) **val** data2 = **new** mutable.MutableList[(Int, String)] *//学号 ---班级* data2.+=((1,**"class\_1"**)) data2.+=((2,**"class\_1"**)) data2.+=((3,**"class\_2"**)) data2.+=((4,**"class\_2"**)) data2.+=((5,**"class\_3"**)) data2.+=((6,**"class\_3"**)) data2.+=((7,**"class\_4"**)) data2.+=((8,**"class\_1"**)) **val** input1: DataSet[(Int, String, Double)] = env.fromCollection(Random.shuffle(data1)) **val** input2: DataSet[(Int, String)] = env.fromCollection(Random.shuffle(data2)) **val** cross = input1.cross(input2){  (input1 , input2) => (input1.\_1,input1.\_2,input1.\_3,input2.\_2) } cross.print() |

#### 14：union

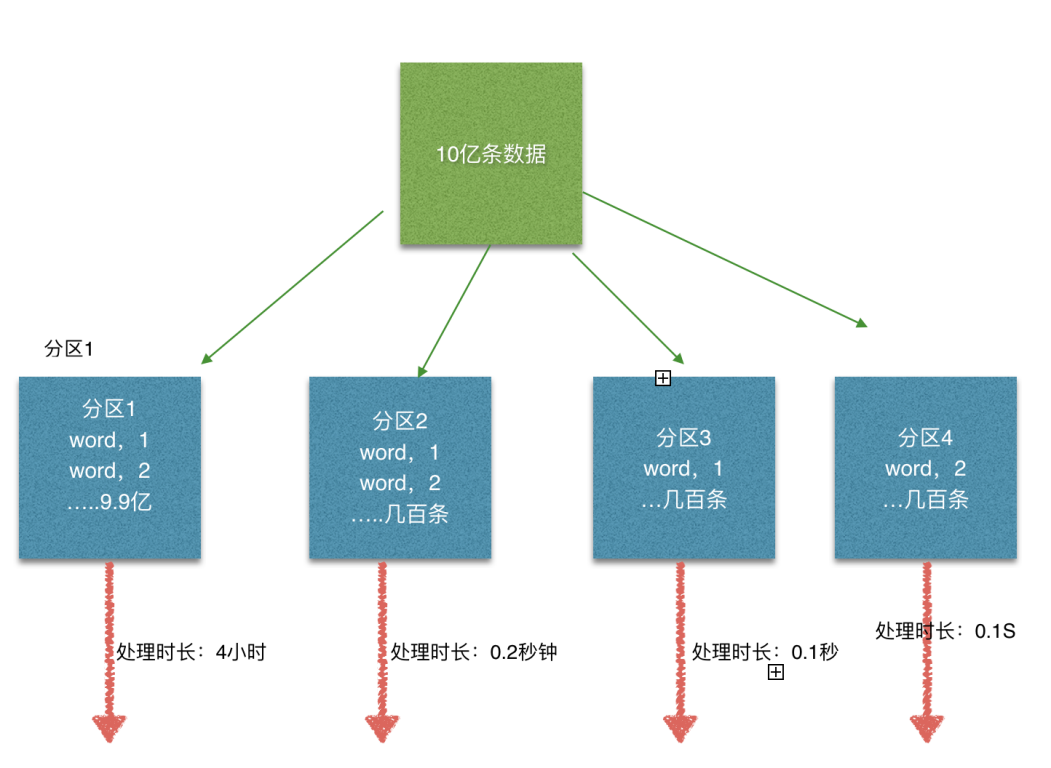
**将多个DataSet合并成一个DataSet**

【注意】：union合并的DataSet的类型必须是一致的

|  |
| --- |
| *//****TODO union联合操作* val** elements1 = env.fromElements((**"123"**)) **val** elements2 = env.fromElements((**"456"**)) **val** elements3 = env.fromElements((**"123"**)) **val** union = elements1.union(elements2).union(elements3).distinct(line => line) union.print() |

#### 15：rebalance

Flink也有数据倾斜的时候，比如当前有数据量大概10亿条数据需要处理，在处理过程中可能会发生如图所示的状况：



这个时候本来总体数据量只需要10分钟解决的问题，出现了数据倾斜，机器1上的任务需要4个小时才能完成，那么其他3台机器执行完毕也要等待机器1执行完毕后才算整体将任务完成；

所以在实际的工作中，出现这种情况比较好的解决方案就是本节课要讲解的—rebalance（内部使用round robin方法将数据均匀打散。这对于数据倾斜时是很好的选择。）

### 

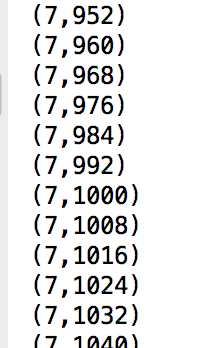
举例：

1：在不使用rebalance的情况下，观察每一个线程执行的任务特点

|  |
| --- |
| **val** ds = env.generateSequence(1, 3000)  **val** rebalanced = ds.filter(\_ > 780) *// val rebalanced = skewed.rebalance()* **val** countsInPartition = rebalanced.map( **new** RichMapFunction[Long, (Int, Long)] {  **def** map(in: Long) = {  *//获取并行时子任务的编号getRuntimeContext.getIndexOfThisSubtask* (getRuntimeContext.getIndexOfThisSubtask, in)  }  })  countsInPartition.print()  【数据随机的分发给各个子任务（分区）】 |

2：使用rebalance

|  |
| --- |
| *//****TODO rebalance* val** ds = env.generateSequence(1, 3000) **val** skewed = ds.filter(\_ > 780) **val** rebalanced = skewed.rebalance() **val** countsInPartition = rebalanced.map( **new** RichMapFunction[Long, (Int, Long)] {  **def** map(in: Long) = {  *//获取并行时子任务的编号getRuntimeContext.getIndexOfThisSubtask* (getRuntimeContext.getIndexOfThisSubtask, in)  } }) countsInPartition.print() |

每隔8一次循环（数据使用轮询的方式在各个子任务中执行）

#### 16:分区

##### （1）：partitionByHash

|  |
| --- |
| *//****TODO*** partitionByHash**val** data = **new** mutable.MutableList[(Int, Long, String)] data.+=((1, 1L, **"Hi"**)) data.+=((2, 2L, **"Hello"**)) data.+=((3, 2L, **"Hello world"**)) data.+=((4, 3L, **"Hello world, how are you?"**)) data.+=((5, 3L, **"I am fine."**)) data.+=((6, 3L, **"Luke Skywalker"**)) data.+=((7, 4L, **"Comment#1"**)) data.+=((8, 4L, **"Comment#2"**)) data.+=((9, 4L, **"Comment#3"**)) data.+=((10, 4L, **"Comment#4"**)) data.+=((11, 5L, **"Comment#5"**)) data.+=((12, 5L, **"Comment#6"**)) data.+=((13, 5L, **"Comment#7"**)) data.+=((14, 5L, **"Comment#8"**)) data.+=((15, 5L, **"Comment#9"**)) data.+=((16, 6L, **"Comment#10"**)) data.+=((17, 6L, **"Comment#11"**)) data.+=((18, 6L, **"Comment#12"**)) data.+=((19, 6L, **"Comment#13"**)) data.+=((20, 6L, **"Comment#14"**)) data.+=((21, 6L, **"Comment#15"**)) **val** collection = env.fromCollection(Random.shuffle(data)) **val** unique = collection.partitionByHash(1).mapPartition{  line =>  line.map(x => (x.\_1 , x.\_2 , x.\_3)) }  unique.writeAsText(**"hashPartition"**, WriteMode.*NO\_OVERWRITE*) env.execute() |

##### （2）：Range-Partition

|  |
| --- |
| *//****TODO Range-Partition* val** data = **new** mutable.MutableList[(Int, Long, String)] data.+=((1, 1L, **"Hi"**)) data.+=((2, 2L, **"Hello"**)) data.+=((3, 2L, **"Hello world"**)) data.+=((4, 3L, **"Hello world, how are you?"**)) data.+=((5, 3L, **"I am fine."**)) data.+=((6, 3L, **"Luke Skywalker"**)) data.+=((7, 4L, **"Comment#1"**)) data.+=((8, 4L, **"Comment#2"**)) data.+=((9, 4L, **"Comment#3"**)) data.+=((10, 4L, **"Comment#4"**)) data.+=((11, 5L, **"Comment#5"**)) data.+=((12, 5L, **"Comment#6"**)) data.+=((13, 5L, **"Comment#7"**)) data.+=((14, 5L, **"Comment#8"**)) data.+=((15, 5L, **"Comment#9"**)) data.+=((16, 6L, **"Comment#10"**)) data.+=((17, 6L, **"Comment#11"**)) data.+=((18, 6L, **"Comment#12"**)) data.+=((19, 6L, **"Comment#13"**)) data.+=((20, 6L, **"Comment#14"**)) data.+=((21, 6L, **"Comment#15"**)) **val** collection = env.fromCollection(Random.shuffle(data)) **val** unique = collection.partitionByRange(x => x.\_1).mapPartition(line => line.map{  x=>  (x.\_1 , x.\_2 , x.\_3) }) unique.writeAsText(**"rangePartition"**, WriteMode.*OVERWRITE*) env.execute() |

##### （3）：sortPartition

根据指定的字段值进行分区的排序；

|  |
| --- |
| *//****TODO Sort Partition* val** data = **new** mutable.MutableList[(Int, Long, String)]  data.+=((1, 1L, **"Hi"**))  data.+=((2, 2L, **"Hello"**))  data.+=((3, 2L, **"Hello world"**))  data.+=((4, 3L, **"Hello world, how are you?"**))  data.+=((5, 3L, **"I am fine."**))  data.+=((6, 3L, **"Luke Skywalker"**))  data.+=((7, 4L, **"Comment#1"**))  data.+=((8, 4L, **"Comment#2"**))  data.+=((9, 4L, **"Comment#3"**))  data.+=((10, 4L, **"Comment#4"**))  data.+=((11, 5L, **"Comment#5"**))  data.+=((12, 5L, **"Comment#6"**))  data.+=((13, 5L, **"Comment#7"**))  data.+=((14, 5L, **"Comment#8"**))  data.+=((15, 5L, **"Comment#9"**))  data.+=((16, 6L, **"Comment#10"**))  data.+=((17, 6L, **"Comment#11"**))  data.+=((18, 6L, **"Comment#12"**))  data.+=((19, 6L, **"Comment#13"**))  data.+=((20, 6L, **"Comment#14"**))  data.+=((21, 6L, **"Comment#15"**))  **val** ds = env.fromCollection(Random.shuffle(data))  **val** result = ds  .map { x => x }.setParallelism(2)  .sortPartition(1, Order.*DESCENDING*)*//第一个参数代表按照哪个字段进行分区* .mapPartition(line => line)  .collect()  *println*(result)  } } |

#### 17：first

|  |
| --- |
| *//****TODO first-取前N个* val** data = **new** mutable.MutableList[(Int, Long, String)]  data.+=((1, 1L, **"Hi"**))  data.+=((2, 2L, **"Hello"**))  data.+=((3, 2L, **"Hello world"**))  data.+=((4, 3L, **"Hello world, how are you?"**))  data.+=((5, 3L, **"I am fine."**))  data.+=((6, 3L, **"Luke Skywalker"**))  data.+=((7, 4L, **"Comment#1"**))  data.+=((8, 4L, **"Comment#2"**))  data.+=((9, 4L, **"Comment#3"**))  data.+=((10, 4L, **"Comment#4"**))  data.+=((11, 5L, **"Comment#5"**))  data.+=((12, 5L, **"Comment#6"**))  data.+=((13, 5L, **"Comment#7"**))  data.+=((14, 5L, **"Comment#8"**))  data.+=((15, 5L, **"Comment#9"**))  data.+=((16, 6L, **"Comment#10"**))  data.+=((17, 6L, **"Comment#11"**))  data.+=((18, 6L, **"Comment#12"**))  data.+=((19, 6L, **"Comment#13"**))  data.+=((20, 6L, **"Comment#14"**))  data.+=((21, 6L, **"Comment#15"**))  **val** ds = env.fromCollection(Random.shuffle(data)) *// ds.first(10).print()  //还可以先goup分组，然后在使用first取值* ds.groupBy(line => line.\_2).first(2).print() |

### 输入数据集Data Sources

flink在批处理中常见的source

flink在批处理中常见的source主要有两大类。

1.基于本地集合的source（Collection-based-source）

2.基于文件的source（File-based-source）

#### 1.基于本地集合的source

在flink最常见的创建DataSet方式有三种。

1.使用env.fromElements()，这种方式也支持Tuple，自定义对象等复合形式。

2.使用env.fromCollection(),这种方式支持多种Collection的具体类型

3.使用env.generateSequence()方法创建基于Sequence的DataSet

|  |
| --- |
| import org.apache.flink.api.scala.{DataSet, ExecutionEnvironment, \_}  import scala.collection.immutable.{Queue, Stack}  import scala.collection.mutable  import scala.collection.mutable.{ArrayBuffer, ListBuffer}  object DataSource001 {  def main(args: Array[String]): Unit = {  val env = ExecutionEnvironment.getExecutionEnvironment  //0.用element创建DataSet(fromElements)  val ds0: DataSet[String] = env.fromElements("spark", "flink")  ds0.print()  //1.用Tuple创建DataSet(fromElements)  val ds1: DataSet[(Int, String)] = env.fromElements((1, "spark"), (2, "flink"))  ds1.print()  //2.用Array创建DataSet  val ds2: DataSet[String] = env.fromCollection(Array("spark", "flink"))  ds2.print()  //3.用ArrayBuffer创建DataSet  val ds3: DataSet[String] = env.fromCollection(ArrayBuffer("spark", "flink"))  ds3.print()  //4.用List创建DataSet  val ds4: DataSet[String] = env.fromCollection(List("spark", "flink"))  ds4.print()  //5.用List创建DataSet  val ds5: DataSet[String] = env.fromCollection(ListBuffer("spark", "flink"))  ds5.print()  //6.用Vector创建DataSet  val ds6: DataSet[String] = env.fromCollection(Vector("spark", "flink"))  ds6.print()  //7.用Queue创建DataSet  val ds7: DataSet[String] = env.fromCollection(Queue("spark", "flink"))  ds7.print()  //8.用Stack创建DataSet  val ds8: DataSet[String] = env.fromCollection(Stack("spark", "flink"))  ds8.print()  //9.用Stream创建DataSet（Stream相当于lazy List，避免在中间过程中生成不必要的集合）  val ds9: DataSet[String] = env.fromCollection(Stream("spark", "flink"))  ds9.print()  //10.用Seq创建DataSet  val ds10: DataSet[String] = env.fromCollection(Seq("spark", "flink"))  ds10.print()  //11.用Set创建DataSet  val ds11: DataSet[String] = env.fromCollection(Set("spark", "flink"))  ds11.print()  //12.用Iterable创建DataSet  val ds12: DataSet[String] = env.fromCollection(Iterable("spark", "flink"))  ds12.print()  //13.用ArraySeq创建DataSet  val ds13: DataSet[String] = env.fromCollection(mutable.ArraySeq("spark", "flink"))  ds13.print()  //14.用ArrayStack创建DataSet  val ds14: DataSet[String] = env.fromCollection(mutable.ArrayStack("spark", "flink"))  ds14.print()  //15.用Map创建DataSet  val ds15: DataSet[(Int, String)] = env.fromCollection(Map(1 -> "spark", 2 -> "flink"))  ds15.print()  //16.用Range创建DataSet  val ds16: DataSet[Int] = env.fromCollection(Range(1, 9))  ds16.print()  //17.用fromElements创建DataSet  val ds17: DataSet[Long] = env.generateSequence(1,9)  ds17.print()  }  } |

#### 2.基于文件的source（File-based-source）

##### （1）：读取本地文件

|  |
| --- |
| *//****TODO 使用readTextFile读取本地文件*** *//****TODO 初始化环境* val** environment: ExecutionEnvironment = ExecutionEnvironment.*getExecutionEnvironment //****TODO 加载数据* val** datas: DataSet[String] = environment.readTextFile(**"data.txt"**) *//****TODO 指定数据的转化* val** flatmap\_data: DataSet[String] = datas.flatMap(line => line.split(**"\\W+"**)) **val** tuple\_data: DataSet[(String, Int)] = flatmap\_data.map(line => (line , 1)) **val** groupData: GroupedDataSet[(String, Int)] = tuple\_data.groupBy(line => line.\_1) **val** result: DataSet[(String, Int)] = groupData.reduce((x, y) => (x.\_1 , x.\_2+y.\_2)) result.print() |

##### （2）：读取hdfs数据

|  |
| --- |
| *//****TODO readTextFile读取hdfs数据*** *//****todo 初始化环境* val** environment: ExecutionEnvironment = ExecutionEnvironment.*getExecutionEnvironment //****TODO 加载数据* val** file: DataSet[String] = environment.readTextFile(**"hdfs://hadoop01:9000/README.txt"**) **val** flatData: DataSet[String] = file.flatMap(line => line.split(**"\\W+"**)) **val** map\_data: DataSet[(String, Int)] = flatData.map(line => (line , 1)) **val** groupdata: GroupedDataSet[(String, Int)] = map\_data.groupBy(line => line.\_1) **val** result\_data: DataSet[(String, Int)] = groupdata.reduce((x, y) => (x.\_1 , x.\_2+y.\_2)) result\_data.print() |

##### （3）：读取CSV数据

|  |
| --- |
| *//****TODO 读取csv数据* val** environment: ExecutionEnvironment = ExecutionEnvironment.*getExecutionEnvironment* **val** path = **"data2.csv" val** ds3 = environment.readCsvFile[(String, String, String, String,String,Int,Int,Int)](  filePath = path,  lineDelimiter = **"\n"**,  fieldDelimiter = **","**,  lenient = **false**,  ignoreFirstLine = **true**,  includedFields = *Array*(0, 1, 2, 3 , 4 , 5 , 6 , 7)) **val** first = ds3.groupBy(0 , 1).first(50) first.print() |

#### 3.基于文件的source（遍历目录）

flink支持对一个文件目录内的所有文件，包括所有子目录中的所有文件的遍历访问方式。

对于从文件中读取数据，当读取的数个文件夹的时候，嵌套的文件默认是不会被读取的，只会读取第一个文件，其他的都会被忽略。所以我们需要使用recursive.file.enumeration进行递归读取

|  |
| --- |
| **val** env = ExecutionEnvironment.*getExecutionEnvironment* **val** parameters = **new** Configuration *// recursive.file.enumeration 开启递归* parameters.setBoolean(**"recursive.file.enumeration"**, **true**) **val** ds1 = env.readTextFile(**"test"**).withParameters(parameters) ds1.print() |

#### 4.读取压缩文件

对于以下压缩类型，不需要指定任何额外的inputformat方法，flink可以自动识别并且解压。但是，压缩文件可能不会并行读取，可能是顺序读取的，这样可能会影响作业的可伸缩性。



|  |
| --- |
| *//****TODO 读取压缩文件* val** env = ExecutionEnvironment.*getExecutionEnvironment* **val** file = env.readTextFile(**"test/data1/zookeeper.out.gz"**).print()  tar -czvf \*\*\*.tar.gz |

### 数据输出Data Sinks

flink在批处理中常见的sink

1.基于本地集合的sink（Collection-based-sink）

2.基于文件的sink（File-based-sink）

#### 1、基于本地集合的sink（Collection-based-sink）

|  |
| --- |
| *//1.定义环境* **val** env = ExecutionEnvironment.*getExecutionEnvironment //2.定义数据 stu(age,name,height)* **val** stu: DataSet[(Int, String, Double)] = env.fromElements(  (19, **"zhangsan"**, 178.8),  (17, **"lisi"**, 168.8),  (18, **"wangwu"**, 184.8),  (21, **"zhaoliu"**, 164.8) ) *//3.****TODO sink到标准输出*** stu.print  *//3.****TODO sink到标准error输出*** stu.printToErr()  *//4.****TODO sink到本地Collection*** *print*(stu.collect()) |

#### 2、基于文件的sink（File-based-sink）

flink支持多种存储设备上的文件，包括本地文件，hdfs文件等。

flink支持多种文件的存储格式，包括text文件，CSV文件等。

* writeAsText()：TextOuputFormat - 将元素作为字符串写入行。字符串是通过调用每个元素的toString()方法获得的。

##### 1、将数据写入本地文件

|  |
| --- |
| *//0.主意：不论是本地还是hdfs.若Parallelism>1将把path当成目录名称，若Parallelism=1将把path当成文件名。* **val** env = ExecutionEnvironment.*getExecutionEnvironment* **val** ds1: DataSource[Map[Int, String]] = env.fromElements(*Map*(1 -> **"spark"** , 2 -> **"flink"**)) *//1.****TODO 写入到本地，文本文档,NO\_OVERWRITE模式下如果文件已经存在，则报错，OVERWRITE模式下如果文件已经存在，则覆盖*** ds1.setParallelism(1).writeAsText(**"test/data1/aa"**, WriteMode.*OVERWRITE*) env.execute() |

##### 2、将数据写入HDFS

|  |
| --- |
| *//****TODO writeAsText将数据写入HDFS* val** env = ExecutionEnvironment.*getExecutionEnvironment* **val** ds1: DataSource[Map[Int, String]] = env.fromElements(*Map*(1 -> **"spark"** , 2 -> **"flink"**)) ds1.setParallelism(1).writeAsText(**"hdfs://hadoop01:9000/a"**, WriteMode.*OVERWRITE*) env.execute() |

可以使用sortPartition对数据进行排序后再sink到外部系统。

|  |
| --- |
| *//****TODO 使用sortPartition对数据进行排序后再sink到外部系统* val** env = ExecutionEnvironment.*getExecutionEnvironment //stu(age,name,height)* **val** stu: DataSet[(Int, String, Double)] = env.fromElements(  (19, **"zhangsan"**, 178.8),  (17, **"lisi"**, 168.8),  (18, **"wangwu"**, 184.8),  (21, **"zhaoliu"**, 164.8) ) *//1.以age从小到大升序排列(0->9)* stu.sortPartition(0, Order.*ASCENDING*).print *//2.以name从大到小降序排列(z->a)* stu.sortPartition(1, Order.*ASCENDING*).print *//3.以age升序，height降序排列* stu.sortPartition(0, Order.*ASCENDING*).sortPartition(2, Order.*DESCENDING*).print *//4.所有字段升序排列* stu.sortPartition(**"\_"**, Order.*ASCENDING*).print *//5.以Student.name升序 //5.1准备数据* **case class** Student(name: String, age: Int) **val** ds1: DataSet[(Student, Double)] = env.fromElements(  (*Student*(**"zhangsan"**, 18), 178.5),  (*Student*(**"lisi"**, 19), 176.5),  (*Student*(**"wangwu"**, 17), 168.5) ) **val** ds2 = ds1.sortPartition(**"\_1.age"**, Order.*ASCENDING*).setParallelism(1) *//5.2写入到hdfs,文本文档* **val** outPath1=**"hdfs://hadoop01:9000/Student001.txt"** ds2.writeAsText(outPath1, WriteMode.*OVERWRITE*) env.execute() *//5.3写入到hdfs,CSV文档* **val** outPath2=**"hdfs://hadoop01:9000/Student002.csv"** ds2.writeAsCsv(outPath2, **"\n"**, **"|||"**,WriteMode.*OVERWRITE*) env.execute() |

### 本地执行和集群执行

#### 本地执行

##### 1：local环境

LocalEnvironment是Flink程序本地执行的句柄。用它在本地JVM中运行程序 - 独立运行或嵌入其他程序中。

本地环境通过该方法实例化ExecutionEnvironment.createLocalEnvironment()。默认情况下，它将使用尽可能多的本地线程执行，因为您的机器具有CPU核心（硬件上下文）。您也可以指定所需的并行性。本地环境可以配置为使用enableLogging()/ 登录到控制台disableLogging()。

在大多数情况下，ExecutionEnvironment.getExecutionEnvironment()是更好的方式。LocalEnvironment当程序在本地启动时（命令行界面外），该方法会返回一个程序，并且当程序由[命令行界面](https://ci.apache.org/projects/flink/flink-docs-release-1.5/ops/cli.html)调用时，它会返回一个预配置的群集执行环境。

注意：本地执行环境不启动任何Web前端来监视执行。

|  |
| --- |
| **object** LocalEven {  **def** main(args: Array[String]): Unit = {  *//****TODO 初始化本地执行环境* val** env: ExecutionEnvironment = ExecutionEnvironment.*createLocalEnvironment*()  **val** path = **"data2.csv"  val** data = env.readCsvFile[(String, String, String, String,String,Int,Int,Int)](  filePath = path,  lineDelimiter = **"\n"**,  fieldDelimiter = **","**,  ignoreFirstLine = **true** )  data.groupBy(0,1).first(100).print()  } } |

##### 2：集合环境

使用集合的执行CollectionEnvironment是执行Flink程序的低开销方法。这种模式的典型用例是自动化测试，调试和代码重用。

用户也可以使用为批处理实施的算法，以便更具交互性的案例

**请注意**，基于集合的Flink程序的执行仅适用于适合JVM堆的小数据。集合上的执行不是多线程的，只使用一个线程

|  |
| --- |
| *//****TODO createCollectionsEnvironment* val** collectionENV = ExecutionEnvironment.*createCollectionsEnvironment* **val** path = **"data2.csv" val** data = collectionENV.readCsvFile[(String, String, String, String,String,Int,Int,Int)](  filePath = path,  lineDelimiter = **"\n"**,  fieldDelimiter = **","**,  ignoreFirstLine = **true** ) data.groupBy(0,1).first(50).print() |

#### 集群执行：

Flink程序可以在许多机器的集群上分布运行。有两种方法可将程序发送到群集以供执行：

##### 第一种方法：命令行界面：

|  |
| --- |
| * ./bin/flink run ./examples/batch/WordCount.jar \ * --input file:///home/user/hamlet.txt --output file:///home/user/wordcount\_out |

##### 第二种方法：使用代码中的远程环境提交

远程环境允许您直接在群集上执行Flink Java程序。远程环境指向要在其上执行程序的群集

Maven打包：

|  |
| --- |
| <**build**>  <**plugins**>  <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-jar-plugin</**artifactId**>  <**version**>2.6</**version**>  <**configuration**>  <**archive**>  <**manifest**>  <**addClasspath**>true</**addClasspath**>  <**classpathPrefix**>lib/</**classpathPrefix**>  <**mainClass**>com.flink.DataStream.RemoteEven</**mainClass**>  </**manifest**>  </**archive**>  </**configuration**>  </**plugin**>  <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-dependency-plugin</**artifactId**>  <**version**>2.10</**version**>  <**executions**>  <**execution**>  <**id**>copy-dependencies</**id**>  <**phase**>package</**phase**>  <**goals**>  <**goal**>copy-dependencies</**goal**>  </**goals**>  <**configuration**>  <**outputDirectory**>${project.build.directory}/lib</**outputDirectory**>  </**configuration**>  </**execution**>  </**executions**>  </**plugin**>  </**plugins**> </**build**> |

|  |
| --- |
| **val** env: ExecutionEnvironment = ExecutionEnvironment.*createRemoteEnvironment*(**"hadoop01"**, 8081, **"target/learning-flink-1.0-SNAPSHOT.jar"**) **val** data: DataSet[String] = env.readTextFile(**"hdfs://hadoop01:9000/README.txt"**) **val** flatMap\_data: DataSet[String] = data.flatMap(line => line.toLowerCase().split(**"\\W+"**)) **val** mapdata: DataSet[(String, Int)] = flatMap\_data.map(line => (line , 1)) **val** groupData: GroupedDataSet[(String, Int)] = mapdata.groupBy(line => line.\_1) **val** result = groupData.reduce((x , y) => (x.\_1 , x.\_2+y.\_2)) result.writeAsText(**"hdfs://hadoop01:9000/remote"**) env.execute() |

#### Flink的广播变量

Flink支持广播变量，就是将数据广播到具体的taskmanager上，数据存储在内存中，这样可以减缓大量的shuffle操作；

比如在数据join阶段，不可避免的就是大量的shuffle操作，我们可以把其中一个dataSet广播出去，一直加载到taskManager的内存中，可以直接在内存中拿数据，避免了大量的shuffle，导致集群性能下降；

**注意**：**因为广播变量是要把dataset广播到内存中，所以广播的数据量不能太大，否则会出现OOM这样的问题**

* + Broadcast：Broadcast是通过withBroadcastSet(dataset，string)来注册的
  + Access：通过getRuntimeContext().getBroadcastVariable(String)访问广播变量

|  |
| --- |
| */\*\*  \* Created by angel；  \*/* **object** BrodCast {  **def** main(args: Array[String]): Unit = {  **val** env: ExecutionEnvironment = ExecutionEnvironment.*getExecutionEnvironment  //****TODO data2 join data3的数据，使用广播变量完成* val** data2 = **new** mutable.MutableList[(Int, Long, String)]  data2.+=((1, 1L, **"Hi"**))  data2.+=((2, 2L, **"Hello"**))  data2.+=((3, 2L, **"Hello world"**))  **val** ds1 = env.fromCollection(Random.shuffle(data2))  **val** data3 = **new** mutable.MutableList[(Int, Long, Int, String, Long)]  data3.+=((1, 1L, 0, **"Hallo"**, 1L))  data3.+=((2, 2L, 1, **"Hallo Welt"**, 2L))  data3.+=((2, 3L, 2, **"Hallo Welt wie"**, 1L))  **val** ds2 = env.fromCollection(Random.shuffle(data3))  *//****todo 使用内部类RichMapFunction，提供open和map，可以完成join的操作* val** result = ds1.map(**new** RichMapFunction[(Int , Long , String) , ArrayBuffer[(Int , Long , String , String)]] {   **var** *brodCast*:mutable.Buffer[(Int, Long, Int, String, Long)] = **null   override def** open(parameters: Configuration): Unit = {  **import** scala.collection.JavaConverters.\_  *//asScala需要使用隐式转换  brodCast* = **this**.getRuntimeContext.getBroadcastVariable[(Int, Long, Int, String, Long)](**"ds2"**).asScala  }  **override def** map(value: (Int, Long, String)):ArrayBuffer[(Int , Long , String , String)] = {  **val** toArray: Array[(Int, Long, Int, String, Long)] = *brodCast*.toArray  **val** array = **new** mutable.ArrayBuffer[(Int , Long , String , String)]  **var** index = 0   **var** a:(Int, Long, String, String) = **null  while**(index < toArray.size){  **if**(value.\_2 == toArray(index).\_5){  a = (value.\_1 , value.\_2 , value.\_3 , toArray(index).\_4)  array += a  }  index = index + 1  }  array  }  }).withBroadcastSet(ds2 , **"ds2"**)  *println*(result.collect())  } } |

#### Flink的分布式缓存

Flink提供了一个类似于Hadoop的分布式缓存，让并行运行实例的函数可以在本地访问。这个功能可以被使用来分享外部静态的数据，例如：机器学习的逻辑回归模型等！

缓存的使用流程：

 使用ExecutionEnvironment实例对本地的或者远程的文件（例如：HDFS上的文件）,为缓存文件指定一个名字注册该缓存文件！当程序执行时候，Flink会自动将复制文件或者目录到所有worker节点的本地文件系统中，函数可以根据名字去该节点的本地文件系统中检索该文件！

【注意】广播是将变量分发到各个worker节点的内存上，分布式缓存是将文件缓存到各个worker节点上；

|  |
| --- |
| **package** com.flink.DEMO.dataset  **import** org.apache.flink.api.common.functions.RichMapFunction **import** org.apache.flink.api.scala.{DataSet, ExecutionEnvironment} **import** org.apache.flink.configuration.Configuration  **import** scala.collection.mutable.{ArrayBuffer, ListBuffer} **import** scala.io.Source **import** org.apache.flink.streaming.api.scala.\_ */\*\*  \* Created by angel；  \*/* **object** Distribute\_cache {  **def** main(args: Array[String]): Unit = {  **val** env = ExecutionEnvironment.*getExecutionEnvironment  //1"开启分布式缓存* **val** path = **"hdfs://hadoop01:9000/score"** env.registerCachedFile(path , **"Distribute\_cache"**)   *//2：加载本地数据* **val** clazz:DataSet[Clazz] = env.fromElements(  *Clazz*(1,**"class\_1"**),  *Clazz*(2,**"class\_1"**),  *Clazz*(3,**"class\_2"**),  *Clazz*(4,**"class\_2"**),  *Clazz*(5,**"class\_3"**),  *Clazz*(6,**"class\_3"**),  *Clazz*(7,**"class\_4"**),  *Clazz*(8,**"class\_1"**)  )   *//3:开始进行关联操作* clazz.map(**new** MyJoinmap()).print()  } } **class** MyJoinmap() **extends** RichMapFunction[Clazz , ArrayBuffer[INFO]]{  **private var** *myLine* = **new** ListBuffer[String]  **override def** open(parameters: Configuration): Unit = {  **val** file = getRuntimeContext.getDistributedCache.getFile(**"Distribute\_cache"**)  **val** lines: Iterator[String] = Source.*fromFile*(file.getAbsoluteFile).getLines()  lines.foreach( line =>{  *myLine*.append(line)  })  }   *//在map函数下进行关联操作* **override def** map(value: Clazz): ArrayBuffer[INFO] = {  **var** stoNO = 0  **var** subject = **""  var** score = 0.0  **var** array = **new** collection.mutable.ArrayBuffer[INFO]()  *//(学生学号---学科---分数)* **for**(str <- *myLine*){  **val** tokens = str.split(**","**)  stoNO = tokens(0).toInt  subject = tokens(1)  score = tokens(2).toDouble  **if**(tokens.length == 3){  **if**(stoNO == value.stu\_no){  array += *INFO*(value.stu\_no , value.clazz\_no , subject , score)  }  }  }  array  } } *//(学号 ， 班级) join (学生学号---学科---分数) ==(学号 ， 班级 ， 学科 ， 分数)* **case class** INFO(stu\_no:Int , clazz\_no:String , subject:String , score:Double) **case class** Clazz(stu\_no:Int , clazz\_no:String) |

## DataStream开发

Flink中的DataStream程序是实现数据流转换（例如，过滤，更新状态，定义窗口，聚合）的常规程序。数据流最初由各种来源（例如，消息队列，套接字流，文件）创建。结果通过接收器返回，例如可以将数据写入文件，或者写入标准输出（例如命令行终端）。Flink程序可以在各种情况下运行，可以独立运行，也可以嵌入其他程序中。执行可以发生在本地JVM或许多机器的集群中。

列子：wordcount

|  |
| --- |
| **object** WordCount {  **def** main(args: Array[String]): Unit = { **val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment* **val** textStream: DataStream[String] = env.socketTextStream(**"localhost"** , 9999) **val** lowerData: DataStream[String] = textStream.flatMap(line => line.toLowerCase().split(**"\\W+"**)) **val** nonEmpty\_data: DataStream[String] = lowerData.filter(line => line.nonEmpty) **val** mapData: DataStream[(String, Int)] = nonEmpty\_data.map(line => (line , 1)) *//基于指定的key进行数据分组* **val** keybyData = mapData.keyBy(0) **val** sum: DataStream[(String, Int)] = keybyData.sum(1) sum.print() env.execute(**"start streaming window wordCount"**)  } } |

### DataStream的Transformation

和dataset一样，dataStream也包括一系列的**Transformation操作：**

<https://ci.apache.org/projects/flink/flink-docs-release-1.5/dev/stream/operators/index.html>

| **Transformation** | **Description** |
| --- | --- |
| **Map** DataStream → DataStream | Takes one element and produces one element. A map function that doubles the values of the input stream:  dataStream**.**map **{** x **=>** x **\*** 2 **}** |
| **FlatMap** DataStream → DataStream | Takes one element and produces zero, one, or more elements. A flatmap function that splits sentences to words:  dataStream**.**flatMap **{** str **=>** str**.**split**(**" "**)** **}** |
| **Filter** DataStream → DataStream | Evaluates a boolean function for each element and retains those for which the function returns true. A filter that filters out zero values:  dataStream**.**filter **{** **\_** **!=** 0 **}** |
| **KeyBy** DataStream → KeyedStream | Logically partitions a stream into disjoint partitions, each partition containing elements of the same key. Internally, this is implemented with hash partitioning. See [keys](https://ci.apache.org/projects/flink/flink-docs-release-1.5/dev/api_concepts.html" \l "specifying-keys) on how to specify keys. This transformation returns a KeyedStream.  dataStream**.**keyBy**(**"someKey"**)** *// Key by field "someKey"*  dataStream**.**keyBy**(**0**)** *// Key by the first element of a Tuple* |
| **Reduce** KeyedStream → DataStream | A "rolling" reduce on a keyed data stream. Combines the current element with the last reduced value and emits the new value.   A reduce function that creates a stream of partial sums:  keyedStream**.**reduce **{** **\_** **+** **\_** **}**    </p> |
| **Fold** KeyedStream → DataStream | A "rolling" fold on a keyed data stream with an initial value. Combines the current element with the last folded value and emits the new value.   A fold function that, when applied on the sequence (1,2,3,4,5), emits the sequence "start-1", "start-1-2", "start-1-2-3", ...  **val** result**:** **DataStream[String]** **=**  keyedStream**.**fold**(**"start"**)((**str**,** i**)** **=>** **{** str **+** "-" **+** i **})** |
| **Aggregations** KeyedStream → DataStream | Rolling aggregations on a keyed data stream. The difference between min and minBy is that min returns the minimum value, whereas minBy returns the element that has the minimum value in this field (same for max and maxBy).  keyedStream**.**sum**(**0**)**  keyedStream**.**sum**(**"key"**)**  keyedStream**.**min**(**0**)**  keyedStream**.**min**(**"key"**)**  keyedStream**.**max**(**0**)**  keyedStream**.**max**(**"key"**)**  keyedStream**.**minBy**(**0**)**  keyedStream**.**minBy**(**"key"**)**  keyedStream**.**maxBy**(**0**)**  keyedStream**.**maxBy**(**"key"**)** |
| **Window** KeyedStream → WindowedStream | Windows can be defined on already partitioned KeyedStreams. Windows group the data in each key according to some characteristic (e.g., the data that arrived within the last 5 seconds). See [windows](https://ci.apache.org/projects/flink/flink-docs-release-1.5/dev/stream/operators/windows.html) for a description of windows.  dataStream**.**keyBy**(**0**).**window**(TumblingEventTimeWindows.**of**(Time.**seconds**(**5**)))** *// Last 5 seconds of data* |
| **WindowAll** DataStream → AllWindowedStream | Windows can be defined on regular DataStreams. Windows group all the stream events according to some characteristic (e.g., the data that arrived within the last 5 seconds). See [windows](https://ci.apache.org/projects/flink/flink-docs-release-1.5/dev/stream/operators/windows.html) for a complete description of windows.  **WARNING:** This is in many cases a **non-parallel** transformation. All records will be gathered in one task for the windowAll operator.  dataStream**.**windowAll**(TumblingEventTimeWindows.**of**(Time.**seconds**(**5**)))** *// Last 5 seconds of data* |
| **Window Apply** WindowedStream → DataStream AllWindowedStream → DataStream | Applies a general function to the window as a whole. Below is a function that manually sums the elements of a window.  **Note:** If you are using a windowAll transformation, you need to use an AllWindowFunction instead.  windowedStream**.**apply **{** **WindowFunction** **}**  *// applying an AllWindowFunction on non-keyed window stream*  allWindowedStream**.**apply **{** **AllWindowFunction** **}** |
| **Window Reduce** WindowedStream → DataStream | Applies a functional reduce function to the window and returns the reduced value.  windowedStream**.**reduce **{** **\_** **+** **\_** **}** |
| **Window Fold** WindowedStream → DataStream | Applies a functional fold function to the window and returns the folded value. The example function, when applied on the sequence (1,2,3,4,5), folds the sequence into the string "start-1-2-3-4-5":  **val** result**:** **DataStream[String]** **=**  windowedStream**.**fold**(**"start"**,** **(**str**,** i**)** **=>** **{** str **+** "-" **+** i **})** |
| **Aggregations on windows** WindowedStream → DataStream | Aggregates the contents of a window. The difference between min and minBy is that min returns the minimum value, whereas minBy returns the element that has the minimum value in this field (same for max and maxBy).  windowedStream**.**sum**(**0**)**  windowedStream**.**sum**(**"key"**)**  windowedStream**.**min**(**0**)**  windowedStream**.**min**(**"key"**)**  windowedStream**.**max**(**0**)**  windowedStream**.**max**(**"key"**)**  windowedStream**.**minBy**(**0**)**  windowedStream**.**minBy**(**"key"**)**  windowedStream**.**maxBy**(**0**)**  windowedStream**.**maxBy**(**"key"**)** |
| **Union** DataStream\* → DataStream | Union of two or more data streams creating a new stream containing all the elements from all the streams. Note: If you union a data stream with itself you will get each element twice in the resulting stream.  dataStream**.**union**(**otherStream1**,** otherStream2**,** **...)** |
| **Window Join** DataStream,DataStream → DataStream | Join two data streams on a given key and a common window.  dataStream**.**join**(**otherStream**)**  **.**where**(<**key selector**>).**equalTo**(<**key selector**>)**  **.**window**(TumblingEventTimeWindows.**of**(Time.**seconds**(**3**)))**  **.**apply **{** **...** **}** |
| **Window CoGroup** DataStream,DataStream → DataStream | Cogroups two data streams on a given key and a common window.  dataStream**.**coGroup**(**otherStream**)**  **.**where**(**0**).**equalTo**(**1**)**  **.**window**(TumblingEventTimeWindows.**of**(Time.**seconds**(**3**)))**  **.**apply **{}** |
| **Connect** DataStream,DataStream → ConnectedStreams | "Connects" two data streams retaining their types, allowing for shared state between the two streams.  someStream **:** **DataStream[Int]** **=** **...**  otherStream **:** **DataStream[String]** **=** **...**  **val** connectedStreams **=** someStream**.**connect**(**otherStream**)** |
| **CoMap, CoFlatMap** ConnectedStreams → DataStream | Similar to map and flatMap on a connected data stream  connectedStreams**.**map**(**  **(\_** **:** **Int)** **=>** **true,**  **(\_** **:** **String)** **=>** **false**  **)**  connectedStreams**.**flatMap**(**  **(\_** **:** **Int)** **=>** **true,**  **(\_** **:** **String)** **=>** **false**  **)** |
| **Split** DataStream → SplitStream | Split the stream into two or more streams according to some criterion.  **val** split **=** someDataStream**.**split**(**  **(**num**:** **Int)** **=>**  **(**num **%** 2**)** **match** **{**  **case** 0 **=>** **List(**"even"**)**  **case** 1 **=>** **List(**"odd"**)**  **}**  **)** |
| **Select** SplitStream → DataStream | Select one or more streams from a split stream.  **val** even **=** split select "even"  **val** odd **=** split select "odd"  **val** all **=** split**.**select**(**"even"**,**"odd"**)** |
| **Iterate** DataStream → IterativeStream → DataStream | Creates a "feedback" loop in the flow, by redirecting the output of one operator to some previous operator. This is especially useful for defining algorithms that continuously update a model. The following code starts with a stream and applies the iteration body continuously. Elements that are greater than 0 are sent back to the feedback channel, and the rest of the elements are forwarded downstream. See [iterations](https://ci.apache.org/projects/flink/flink-docs-release-1.5/dev/stream/operators/index.html" \l "iterations) for a complete description.  initialStream**.**iterate **{**  iteration **=>** **{**  val iterationBody **=** iteration**.**map **{***/\*do something\*/***}**  **(**iterationBody**.**filter**(**\_ **>** 0**),** iterationBody**.**filter**(**\_ **<=** 0**))**  **}**  **}** |
| **Extract Timestamps** DataStream → DataStream | Extracts timestamps from records in order to work with windows that use event time semantics. See [Event Time](https://ci.apache.org/projects/flink/flink-docs-release-1.5/dev/event_time.html).  stream**.**assignTimestamps **{** timestampExtractor **}** |

#### **KeyBy**

逻辑上将一个流分成不相交的分区，每个分区包含相同键的元素。在内部，这是通过散列分区来实现的

|  |
| --- |
| **object** Keyby {  **def** main(args: Array[String]): Unit = {  **val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment* env.setParallelism(3)  **val** textStream: DataStream[String] = env.socketTextStream(**"localhost"** , 12345)  **val** flatMap\_data: DataStream[String] = textStream.flatMap(line => line.split(**"\t"**))  **val** map\_data: DataStream[(String, Int)] = flatMap\_data.map(line => (line , 1))  *//****TODO 逻辑上将一个流分成不相交的分区，每个分区包含相同键的元素。在内部，这是通过散列分区来实现的* val** keyByData: KeyedStream[(String, Int), String] = map\_data.keyBy(line => line.\_1)  keyByData.writeAsText(**"keyByData"**)  env.execute()  } } |

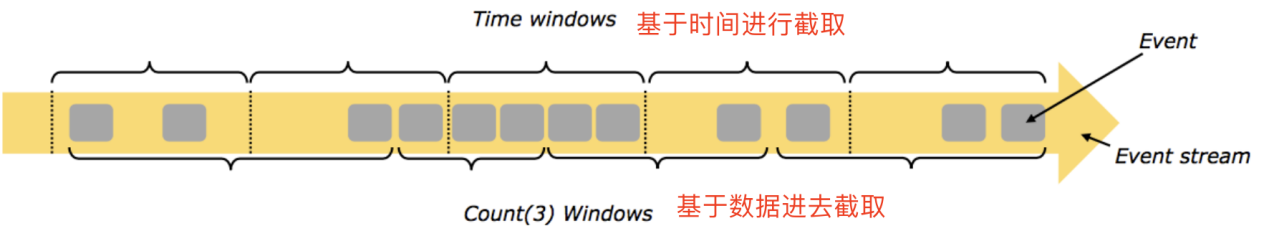
#### Windows

在讲解windows的众多操作之前，需要讲解一个概念：

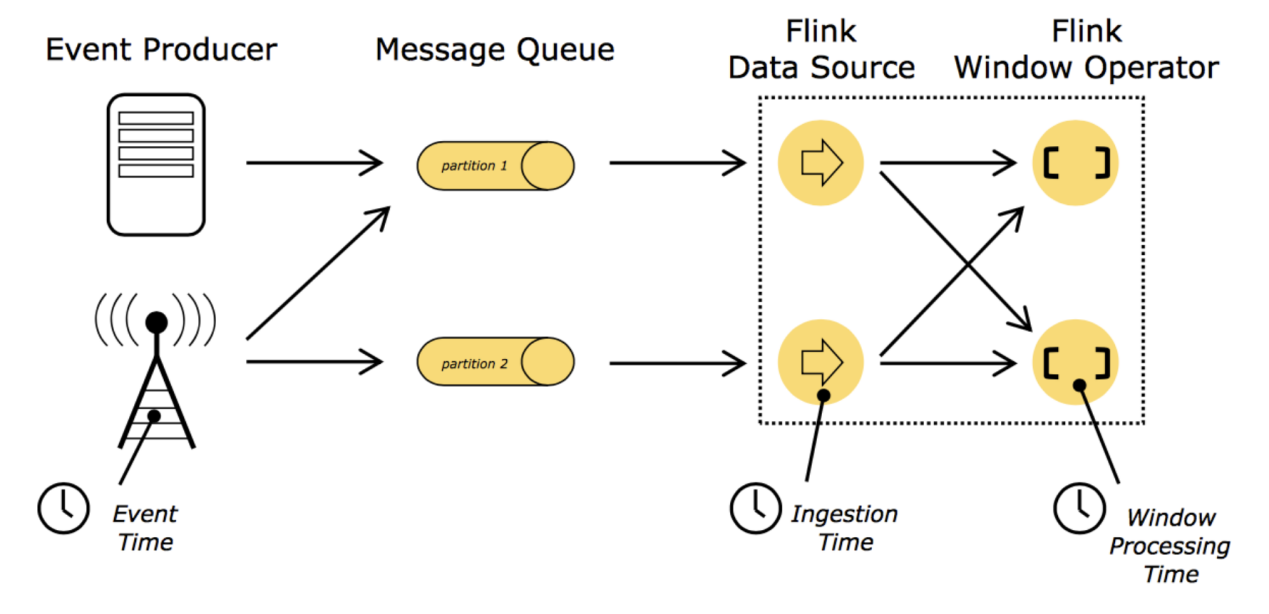
源源不断的数据流是无法进行统计工作的，因为数据流没有边界，就无法统计到底有多少数据经过了这个流。也无法统计数据流中的最大值，最小值，平均值，累加值等信息。

如果在数据流上，截取固定大小的一部分，这部分是可以进行统计的。 截取方式主要有两种，

|  |
| --- |
| 1.根据时间进行截取(time-driven-window)，比如每1分钟统计一次或每10分钟统计一次。  2.根据数据进行截取(data-driven-window)，比如每5个数据统计一次或每50个数据统计一次。 |



基于流的操作图：

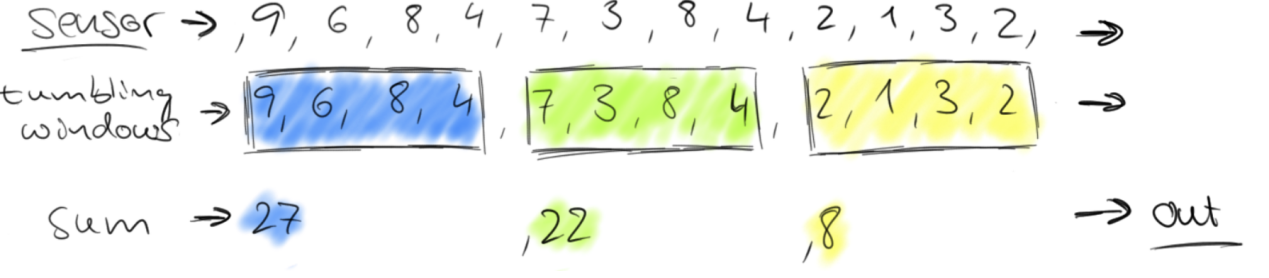


##### 关于window的理论+实践

###### 一：tumbling-time-window (无重叠数据)

1.红绿灯路口会有汽车通过，一共会有多少汽车通过，无法计算。因为车流源源不断，计算没有边界。

2.统计每15秒钟通过红路灯的汽车数量，第一个15秒为2辆，第二个15秒为3辆，第三个15秒为1辆。。。



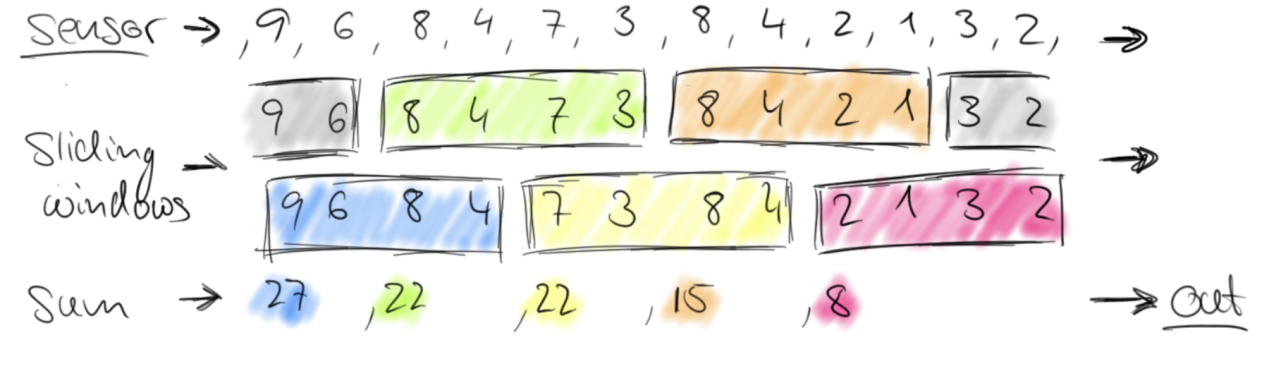
1.tumbling-time-window (无重叠数据)实战

|  |
| --- |
| 1.发送命令  nc -lk 9999  2.发送内容  9,3  9,2  9,7  4,9  2,6  1,5  2,3  5,7  5,4 |

程序：

|  |
| --- |
| **object** Window {  **def** main(args: Array[String]): Unit = {  *//****TODO time-window*** *//1.创建运行环境* **val** env = StreamExecutionEnvironment.*getExecutionEnvironment   //2.定义数据流来源* **val** text = env.socketTextStream(**"localhost"**, 9999)   *//3.转换数据格式，text->CarWc* **case class** CarWc(sensorId: Int, carCnt: Int)  **val** ds1: DataStream[CarWc] = text.map {  line => {  **val** tokens = line.split(**","**)  *CarWc*(tokens(0).trim.toInt, tokens(1).trim.toInt)  }  }   *//4.执行统计操作，每个sensorId一个tumbling窗口，窗口的大小为5秒  //也就是说，每5秒钟统计一次，在这过去的5秒钟内，各个路口通过红绿灯汽车的数量。* **val** ds2: DataStream[CarWc] = ds1  .keyBy(**"sensorId"**)  .timeWindow(Time.*seconds*(5))  .sum(**"carCnt"**)   *//5.显示统计结果* ds2.print()   *//6.触发流计算* env.execute(**this**.getClass.getName)   } } |

###### 二、sliding-time-window (有重叠数据)



|  |
| --- |
| *//****TODO 2.tumbling-time-window（有重叠）*** *//1.创建运行环境* **val** env = StreamExecutionEnvironment.*getExecutionEnvironment  //2.定义数据流来源* **val** text = env.socketTextStream(**"localhost"**, 9999)  *//3.转换数据格式，text->CarWc* **case class** CarWc(sensorId: Int, carCnt: Int) **val** ds1: DataStream[CarWc] = text.map {  line => {  **val** tokens = line.split(**","**)  *CarWc*(tokens(0).trim.toInt, tokens(1).trim.toInt)  } } *//4.执行统计操作，每个sensorId一个sliding窗口，窗口时间10秒,滑动时间5秒 //也就是说，每5秒钟统计一次，在这过去的10秒钟内，各个路口通过红绿灯汽车的数量。* **val** ds2: DataStream[CarWc] = ds1  .keyBy(**"sensorId"**)  .timeWindow(Time.*seconds*(10), Time.*seconds*(5))  .sum(**"carCnt"**)  *//5.显示统计结果* ds2.print()  *//6.触发流计算* env.execute(**this**.getClass.getName) |

###### 三：tumbling-count-window (无重叠数据)

按照个数进行统计，比如：

每个路口分别统计，收到关于它的5条消息时统计在最近5条消息中，各自路口通过的汽车数量

|  |
| --- |
| 1.发送命令  nc -lk 9999  2.发送内容  9,3  9,2  9,7  4,9  2,6  1,5  2,3  5,7  5,4 |

程序：

|  |
| --- |
| *//****TODO tumbling-count-window (无重叠数据)*** *//1.创建运行环境* **val** env = StreamExecutionEnvironment.*getExecutionEnvironment  //2.定义数据流来源* **val** text = env.socketTextStream(**"localhost"**, 9999)  *//3.转换数据格式，text->CarWc* **case class** CarWc(sensorId: Int, carCnt: Int) **val** ds1: DataStream[CarWc] = text.map {  (f) => {  **val** tokens = f.split(**","**)  *CarWc*(tokens(0).trim.toInt, tokens(1).trim.toInt)  } } *//4.执行统计操作，每个sensorId一个tumbling窗口，窗口的大小为5 //按照key进行收集，对应的key出现的次数达到5次作为一个结果* **val** ds2: DataStream[CarWc] = ds1  .keyBy(**"sensorId"**)  .countWindow(5)  .sum(**"carCnt"**)  *//5.显示统计结果* ds2.print()  *//6.触发流计算* env.execute(**this**.getClass.getName) |

###### 四：sliding-count-window (有重叠数据)

同样也是窗口长度和滑动窗口的操作：窗口长度是5，滑动长度是3

|  |
| --- |
| *//****TODO sliding-count-window（有重叠）*** *//1.创建运行环境* **val** env = StreamExecutionEnvironment.*getExecutionEnvironment  //2.定义数据流来源* **val** text = env.socketTextStream(**"localhost"**, 9999)  *//3.转换数据格式，text->CarWc* **case class** CarWc(sensorId: Int, carCnt: Int) **val** ds1: DataStream[CarWc] = text.map {  (f) => {  **val** tokens = f.split(**","**)  *CarWc*(tokens(0).trim.toInt, tokens(1).trim.toInt)  } } *//4.执行统计操作，每个sensorId一个sliding窗口，窗口大小3条数据,窗口滑动为3条数据 //也就是说，每个路口分别统计，收到关于它的3条消息时统计在最近5条消息中，各自路口通过的汽车数量* **val** ds2: DataStream[CarWc] = ds1  .keyBy(**"sensorId"**)  .countWindow(5, 3)  .sum(**"carCnt"**)  *//5.显示统计结果* ds2.print()  *//6.触发流计算* env.execute(**this**.getClass.getName) |

#### window总结

1.flink支持两种划分窗口的方式（time和count）

如果根据时间划分窗口，那么它就是一个time-window

如果根据数据划分窗口，那么它就是一个count-window

2.flink支持窗口的两个重要属性（size和interval）

如果size=interval,那么就会形成tumbling-window(无重叠数据)

如果size>interval,那么就会形成sliding-window(有重叠数据)

如果size<interval,那么这种窗口将会丢失数据。比如每5秒钟，统计过去3秒的通过路口汽车的数据，将会漏掉2秒钟的数据。

3.通过组合可以得出四种基本窗口

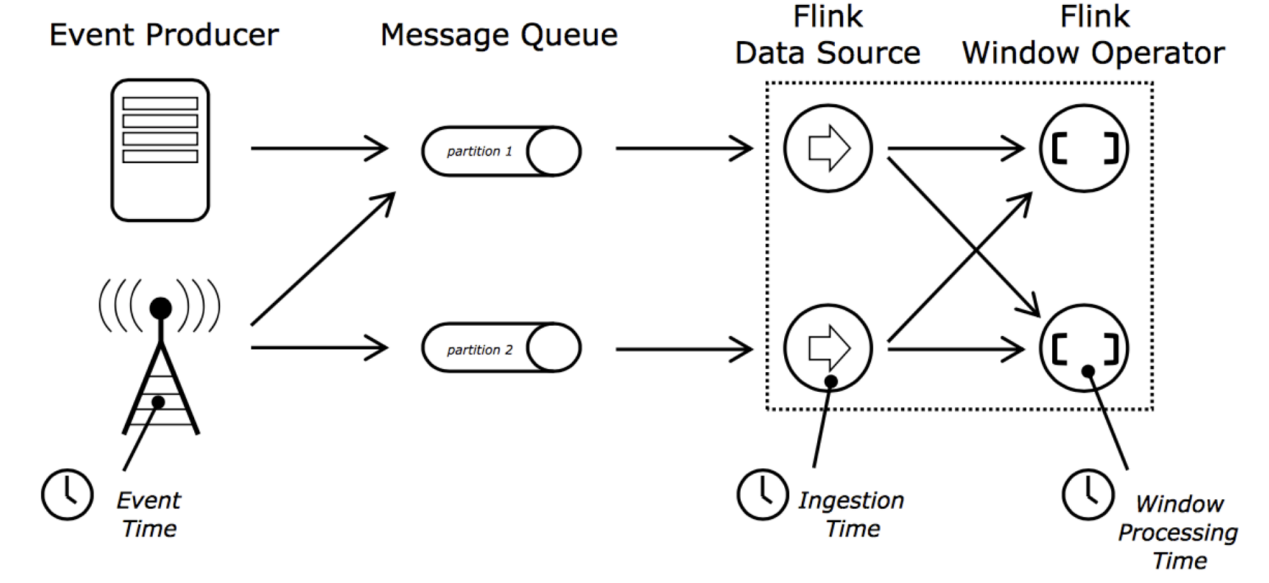
time-tumbling-window 无重叠数据的时间窗口，设置方式举例：timeWindow(Time.seconds(5))

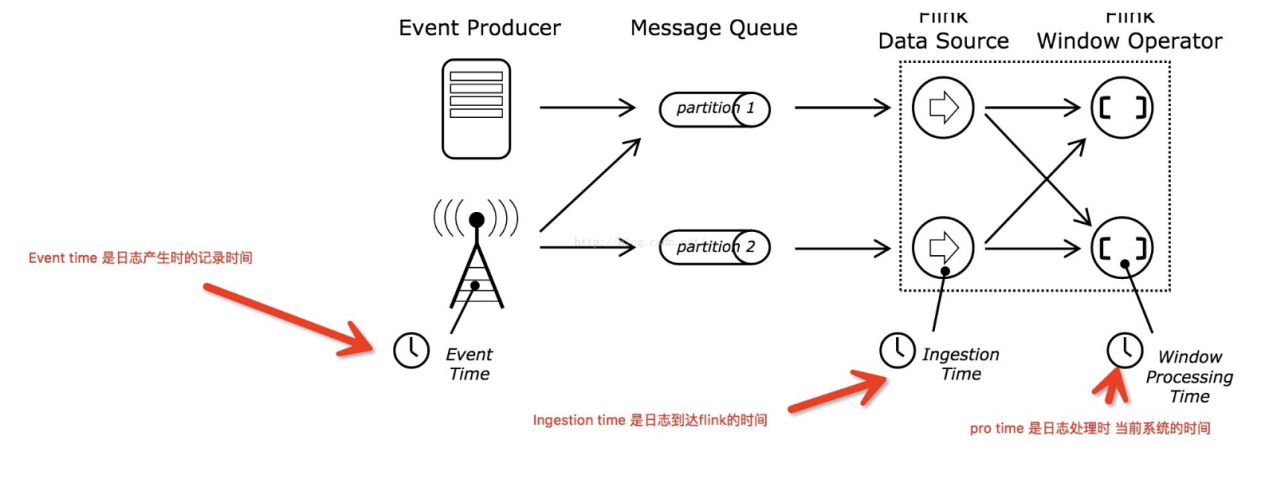
time-sliding-window 有重叠数据的时间窗口，设置方式举例：timeWindow(Time.seconds(5), Time.seconds(3))

count-tumbling-window无重叠数据的数量窗口，设置方式举例：countWindow(5)

count-sliding-window 有重叠数据的数量窗口，设置方式举例：countWindow(5,3)

#### time-window的高级用法





1.现实世界中的时间是不一致的，在flink中被划分为事件时间，提取时间，处理时间三种。

2.如果以EventTime为基准来定义时间窗口那将形成EventTimeWindow,要求消息本身就应该携带EventTime

2.如果以IngesingtTime为基准来定义时间窗口那将形成IngestingTimeWindow,以source的systemTime为准。

2.如果以ProcessingTime基准来定义时间窗口那将形成ProcessingTimeWindow，以operator的systemTime为准。

##### EventTime

**1.要求消息本身就应该携带EventTime**

2.时间对应关系如下

|  |  |
| --- | --- |
| **2018-06-02 11:45:55** | 1527911155000 |
| **2018-06-02 11:45:56** | 1527911156000 |
| **2018-06-02 11:45:56** | 1527911157000 |

需求：

以EventTime划分窗口，计算3秒钟内出价最高的产品

数据：

|  |
| --- |
| 1527911155000,boos1,pc1,100.0  1527911156000,boos2,pc1,200.0  1527911157000,boos1,pc1,300.0  1527911158000,boos2,pc1,500.0  1527911159000,boos1,pc1,600.0  1527911160000,boos1,pc1,700.0  1527911161000,boos2,pc2,700.0  1527911162000,boos2,pc2,900.0  1527911163000,boos2,pc2,1000.0  1527911164000,boos2,pc2,1100.0  1527911165000,boos1,pc2,1100.0  1527911166000,boos2,pc2,1300.0  1527911167000,boos2,pc2,1400.0  1527911168000,boos2,pc2,1600.0  1527911169000,boos1,pc2,1300.0 |

代码实现：

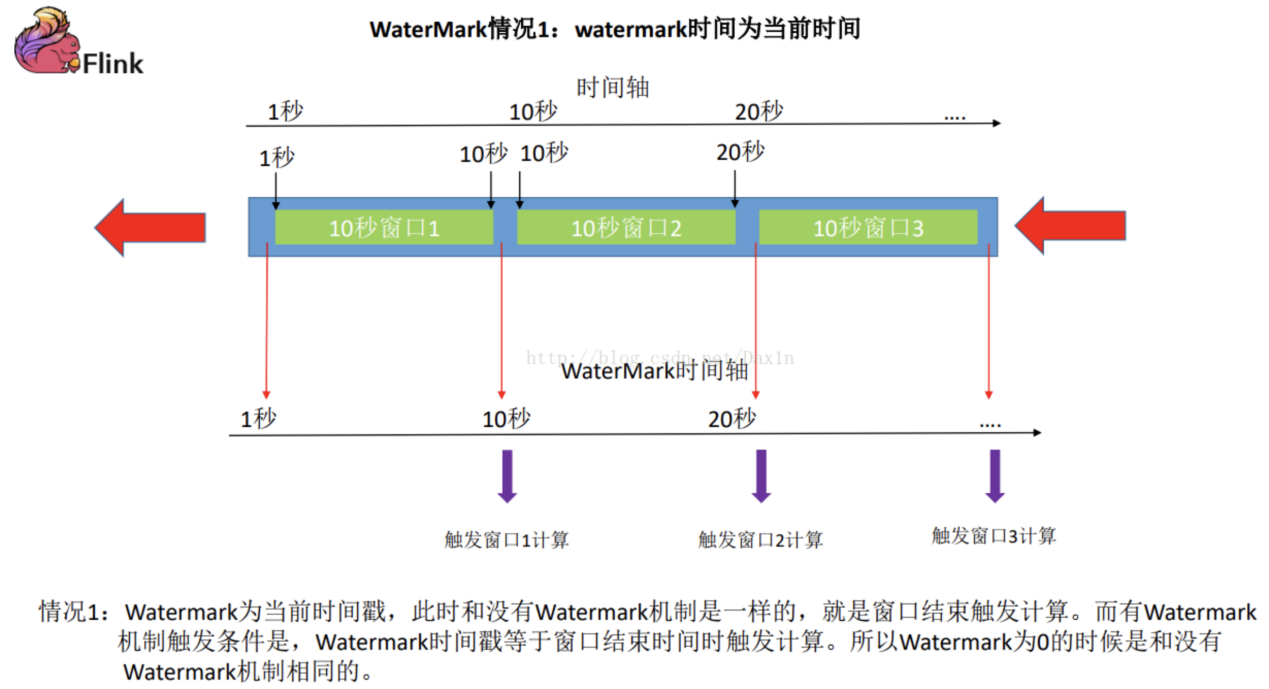
|  |
| --- |
| **object** EventTimeExample {  **def** main(args: Array[String]) {   *//1.创建执行环境，并设置为使用EventTime* **val** env = StreamExecutionEnvironment.*getExecutionEnvironment  //置为使用EventTime* env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)   *//2.创建数据流，并进行数据转化* **val** source = env.socketTextStream(**"localhost"**, 9999)  **case class** SalePrice(time: Long, boosName: String, productName: String, price: Double)  **val** dst1: DataStream[SalePrice] = source.map(value => {  **val** columns = value.split(**","**)  SalePrice(columns(0).toLong, columns(1), columns(2), columns(3).toDouble)  })   *//3.使用EventTime进行求最值操作* **val** dst2: DataStream[SalePrice] = dst1  *//提取消息中的时间戳属性* .assignAscendingTimestamps(\_.time)  .keyBy(\_.productName)  .timeWindow(Time.*seconds*(3))*//设置window方法一* .max(**"price"**)   *//4.显示结果* dst2.print()   *//5.触发流计算* env.execute()  } } |

当前代码理论上看没有任何问题，在实际使用的时候就会出现很多问题,甚至接收不到数据或者接收到的数据是不准确的；这是因为对于flink最初设计的时候，就考虑到了网络延迟，网络乱序等问题，所以提出了一个抽象概念基座水印（WaterMark）；

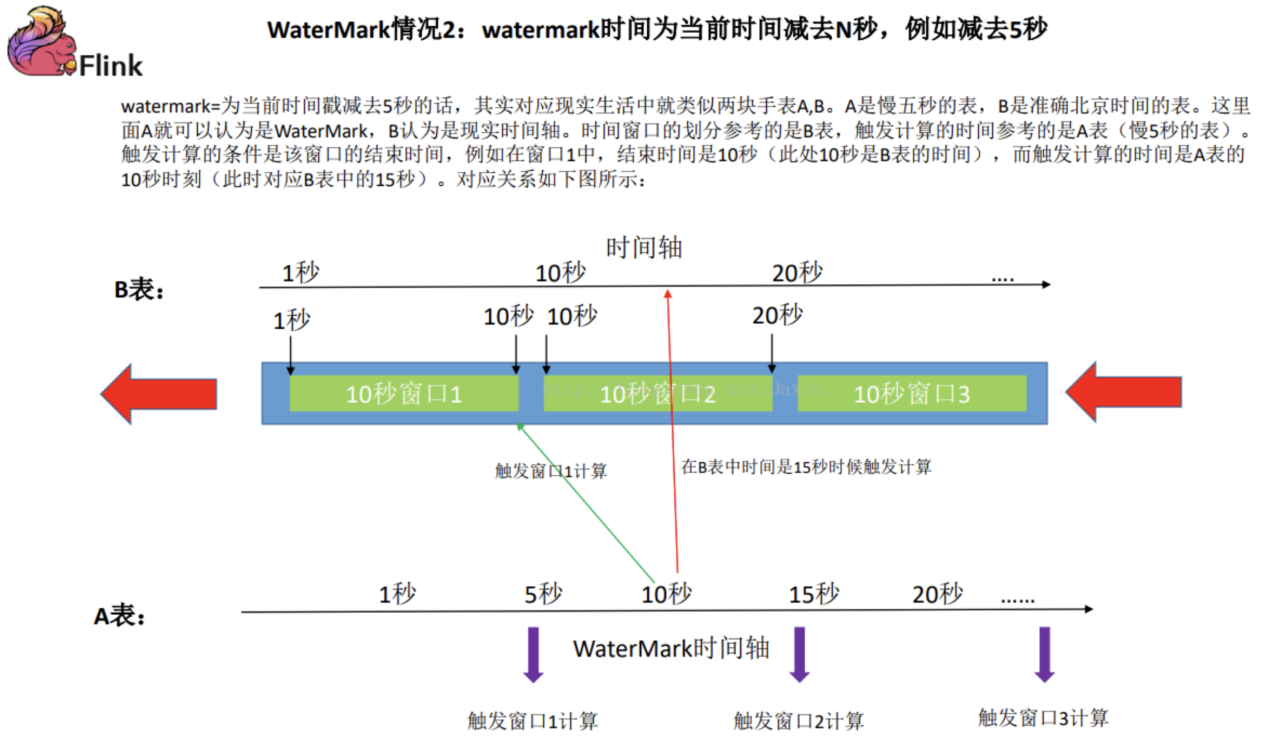


水印分成两种形式：

第一种：



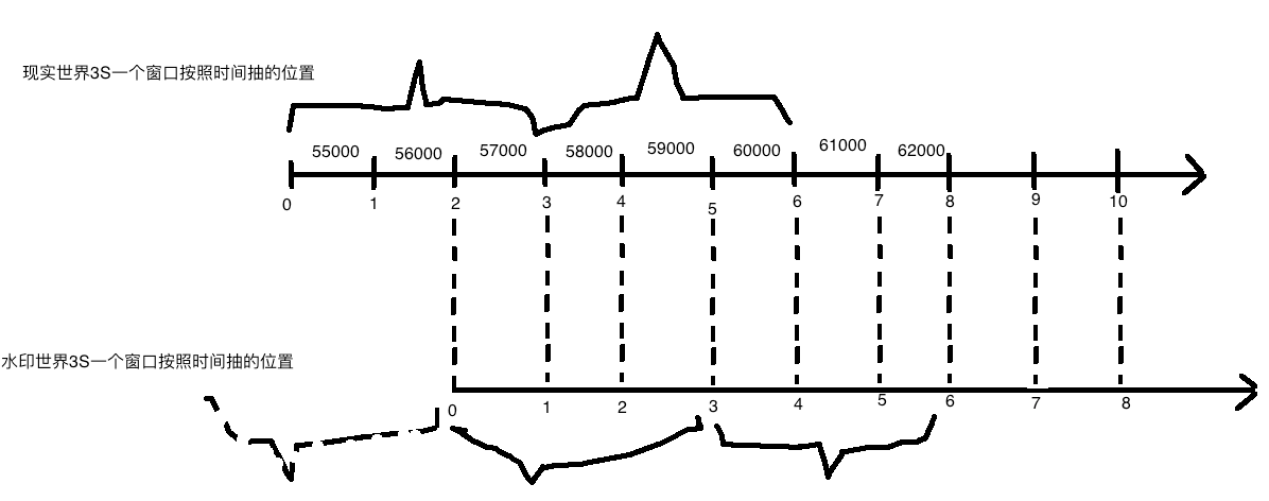
第二种:



所以，我们需要考虑到网络延迟的状况，那么代码中就需要添加水印操作：

|  |
| --- |
| **object** EventTimeOperator {  **def** main(args: Array[String]): Unit = {  *//创建执行环境，并设置为使用EventTime* **val** env = StreamExecutionEnvironment.*getExecutionEnvironment* env.setParallelism(1)*//注意控制并发数  //置为使用EventTime* env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)  **val** source = env.socketTextStream(**"localhost"**, 9999)  **val** dst1: DataStream[SalePrice] = source.map(value => {  **val** columns = value.split(**","**)  *SalePrice*(columns(0).toLong, columns(1), columns(2), columns(3).toDouble)  })  *//****todo 水印时间 assignTimestampsAndWatermarks*****val** timestamps\_data = dst1.assignTimestampsAndWatermarks(**new** AssignerWithPeriodicWatermarks[SalePrice]{   **var** *currentMaxTimestamp*:Long = 0  **val** *maxOutOfOrderness* = 2000L *//最大允许的乱序时间是2s* **var** *wm* : Watermark = **null  val** *format* = **new** SimpleDateFormat(**"yyyy-MM-dd HH:mm:ss.SSS"**)  **override def** getCurrentWatermark: Watermark = {  *wm* = **new** Watermark(*currentMaxTimestamp* - *maxOutOfOrderness*) *wm* }   **override def** extractTimestamp(element: SalePrice, previousElementTimestamp: Long): Long = {  **val** timestamp = element.time  *currentMaxTimestamp* = Math.*max*(timestamp, *currentMaxTimestamp*)  }  })  **val** data: KeyedStream[SalePrice, String] = timestamps\_data.keyBy(line => line.productName)  **val** window\_data: WindowedStream[SalePrice, String, TimeWindow] = data.timeWindow(Time.*seconds*(3))  **val** apply: DataStream[SalePrice] = window\_data.apply(**new** MyWindowFunc)  apply.print()  env.execute()   } } **case class** SalePrice(time: Long, boosName: String, productName: String, price: Double) **class** MyWindowFunc **extends** WindowFunction[SalePrice , SalePrice , String, TimeWindow]{  **override def** apply(key: String, window: TimeWindow, input: Iterable[SalePrice], out: Collector[SalePrice]): Unit = {  **val** seq = input.toArray  **val** take: Array[SalePrice] = seq.sortBy(line => line.price).reverse.take(1)  **for**(info <- take){  out.collect(info)  }  } } |

上列图讲解：



##### ProcessingTime

对于processTime而言，是flink处理数据的时间，所以就不关心发过来的数据是不是有延迟操作，只关心数据具体的处理时间，所以不需要水印处理，操作相对来说简单了很多

|  |
| --- |
| **object** ProcessingTimeExample {  **def** main(args: Array[String]) {  *//创建执行环境，并设置为使用EventTime* **val** env = StreamExecutionEnvironment.*getExecutionEnvironment* env.setParallelism(2)*//注意控制并发数  //置为使用ProcessingTime* env.setStreamTimeCharacteristic(TimeCharacteristic.*ProcessingTime*)   **val** source = env.socketTextStream(**"localhost"**, 9999)  **case class** SalePrice(time: Long, boosName: String, productName: String, price: Double)   **val** dst1: DataStream[SalePrice] = source.map(value => {  **val** columns = value.split(**","**)  *SalePrice*(columns(0).toLong, columns(1), columns(2), columns(3).toDouble)  })  *//processTime不需要提取消息中的时间 // val timestamps\_data: DataStream[SalePrice] = dst1.assignAscendingTimestamps(line => line.time)* **val** keyby\_data: KeyedStream[SalePrice, String] = dst1.keyBy(line => line.productName)  *//****TODO 窗口事件是:TumblingProcessingTimeWindows* val** window\_data: WindowedStream[SalePrice, String, TimeWindow] = keyby\_data.window(TumblingProcessingTimeWindows.*of*(Time.*seconds*(5)))**val** max\_price: DataStream[SalePrice] = window\_data.max(**"price"**)  max\_price.print()  env.execute()  } } |

#### Window apply

和window的操作类似，只不过操作更加灵活，具体的操作需要在匿名内部类的方法中实现；当有比较复杂的需求时候，可以使用；

|  |
| --- |
| **object** WindowApply {  **def** main(args: Array[String]): Unit = {  **val** env = StreamExecutionEnvironment.*getExecutionEnvironment* **val** textStream: DataStream[String] = env.socketTextStream(**"localhost"** , 9999)  **val** flatmapdata: DataStream[String] = textStream.flatMap(x => x.split(**","**))  **val** mapdata: DataStream[(String, Int)] = flatmapdata.map(line => (line,1))  **val** keybyStream: KeyedStream[(String, Int), String] = mapdata.keyBy(line => line.\_1)  **val** window: WindowedStream[(String, Int), String, TimeWindow] = keybyStream.timeWindow(Time.*of*(1 , TimeUnit.*SECONDS*) ,Time.*of*(100,TimeUnit.*MILLISECONDS*))  **val** data = window.apply(**new** WindowFunction[(String, Int) , (String, Int) , String , TimeWindow] {  **override def** apply(key: String,  window: TimeWindow,  input: Iterable[(String, Int)],  out: Collector[(String, Int)]): Unit = {  **var** output = **""  var** index = 0  **for**(in <- input){  output += **"key :"** + in.\_1 + **" value:"**+in.\_2  index = index + 1  out.collect(output , index)  }  }  })  data.print()  env.execute()   } } |

注意，例子中使用的是window，所以对应的匿名内部类是：WindowFunction

如果使用的是windowAll，则需要使用的内部类是：AllWindowFunction

#### connect

用来将两个dataStream组装成一个ConnectedStreams

而且这个connectedStream的组成结构就是保留原有的dataStream的结构体；这样我们就可以把不同的数据组装成同一个结构

|  |
| --- |
| **val** env = StreamExecutionEnvironment.*getExecutionEnvironment* env.setParallelism(1) **val** src: DataStream[Int] = env.fromElements(1, 3, 5) **val** stringMap: DataStream[String] = src.map(line => **"x "**+line) **val** result = stringMap.connect(src).map(**new** CoMapFunction[String , Int , String] {  **override def** map2(value: Int): String = {  **"x "**+ (value + 1)  }   **override def** map1(value: String): String = {  value  } }) result.print() env.execute() |

#### Split和select



Split就是将一个DataStream分成两个或者多个DataStream

Select就是获取分流后对应的数据

|  |
| --- |
| **val** env = StreamExecutionEnvironment.*getExecutionEnvironment* env.setParallelism(1) **val** elements: DataStream[Int] = env.fromElements(1,2,3,4,5,6) *//数据分流* **val** split\_data = elements.split(  (num: Int) => (num % 2) **match** {  **case** 0 => *List*(**"even"**)  **case** 1 => *List*(**"odd"**)  } ) *//获取分流后的数据* **val** select: DataStream[Int] = split\_data.select(**"even"**) select.print() env.execute() |

### Flink在流处理上常见的Source和sink操作

flink在流处理上的source和在批处理上的source基本一致。大致有4大类

1.基于本地集合的source（Collection-based-source）

2.基于文件的source（File-based-source）

3.基于网络套接字的source（Socket-based-source）

4.自定义的source（Custom-source）

#### 1：基于集合的source

|  |
| --- |
| import org.apache.flink.streaming.api.scala.{StreamExecutionEnvironment, \_}  import scala.collection.immutable.{Queue, Stack}  import scala.collection.mutable  import scala.collection.mutable.{ArrayBuffer, ListBuffer}  object DataSource001 {  def main(args: Array[String]): Unit = {  val senv = StreamExecutionEnvironment.getExecutionEnvironment  //0.用element创建DataStream(fromElements)  val ds0: DataStream[String] = senv.fromElements("spark", "flink")  ds0.print()  //1.用Tuple创建DataStream(fromElements)  val ds1: DataStream[(Int, String)] = senv.fromElements((1, "spark"), (2, "flink"))  ds1.print()  //2.用Array创建DataStream  val ds2: DataStream[String] = senv.fromCollection(Array("spark", "flink"))  ds2.print()  //3.用ArrayBuffer创建DataStream  val ds3: DataStream[String] = senv.fromCollection(ArrayBuffer("spark", "flink"))  ds3.print()  //4.用List创建DataStream  val ds4: DataStream[String] = senv.fromCollection(List("spark", "flink"))  ds4.print()  //5.用List创建DataStream  val ds5: DataStream[String] = senv.fromCollection(ListBuffer("spark", "flink"))  ds5.print()  //6.用Vector创建DataStream  val ds6: DataStream[String] = senv.fromCollection(Vector("spark", "flink"))  ds6.print()  //7.用Queue创建DataStream  val ds7: DataStream[String] = senv.fromCollection(Queue("spark", "flink"))  ds7.print()  //8.用Stack创建DataStream  val ds8: DataStream[String] = senv.fromCollection(Stack("spark", "flink"))  ds8.print()  //9.用Stream创建DataStream（Stream相当于lazy List，避免在中间过程中生成不必要的集合）  val ds9: DataStream[String] = senv.fromCollection(Stream("spark", "flink"))  ds9.print()  //10.用Seq创建DataStream  val ds10: DataStream[String] = senv.fromCollection(Seq("spark", "flink"))  ds10.print()  //11.用Set创建DataStream(不支持)  //val ds11: DataStream[String] = senv.fromCollection(Set("spark", "flink"))  //ds11.print()  //12.用Iterable创建DataStream(不支持)  //val ds12: DataStream[String] = senv.fromCollection(Iterable("spark", "flink"))  //ds12.print()  //13.用ArraySeq创建DataStream  val ds13: DataStream[String] = senv.fromCollection(mutable.ArraySeq("spark", "flink"))  ds13.print()  //14.用ArrayStack创建DataStream  val ds14: DataStream[String] = senv.fromCollection(mutable.ArrayStack("spark", "flink"))  ds14.print()  //15.用Map创建DataStream(不支持)  //val ds15: DataStream[(Int, String)] = senv.fromCollection(Map(1 -> "spark", 2 -> "flink"))  //ds15.print()  //16.用Range创建DataStream  val ds16: DataStream[Int] = senv.fromCollection(Range(1, 9))  ds16.print()  //17.用fromElements创建DataStream  val ds17: DataStream[Long] = senv.generateSequence(1, 9)  ds17.print()    senv.execute(this.getClass.getName)  }  } |

#### 2.基于文件的source（File-based-source）

|  |
| --- |
| *//****TODO 2.基于文件的source（File-based-source）*** *//0.创建运行环境* **val** env = StreamExecutionEnvironment.*getExecutionEnvironment //****TODO 1.读取本地文件* val** text1 = env.readTextFile(**"data2.csv"**) text1.print() *//****TODO 2.读取hdfs文件* val** text2 = env.readTextFile(**"hdfs://hadoop01:9000/input/flink/README.txt"**) text2.print() env.execute() |

#### 3.基于网络套接字的source（Socket-based-source）

|  |
| --- |
| **val** source = env.socketTextStream(**"IP"**, PORT) |

#### 4.自定义的source（Custom-source,以kafka为例）

Kafka基本命令：

|  |
| --- |
| ● 查看当前服务器中的所有topic  bin/kafka-topics.sh --list --zookeeper hadoop01:2181  ● 创建topic  bin/kafka-topics.sh --create --zookeeper hadoop01:2181 --replication-factor 1 --partitions 1 --topic test  ● 删除topic  sh bin/kafka-topics.sh --delete --zookeeper zk01:2181 --topic test  需要server.properties中设置delete.topic.enable=true否则只是标记删除或者直接重启。  ● 通过shell命令发送消息  sh bin/kafka-console-producer.sh --broker-list hadoop01:9092 --topic test  ● 通过shell消费消息  bin/kafka-console-consumer.sh --zookeeper hadoop01:2181 --from-beginning --topic test1  ● 查看消费位置  bin/kafka-run-cla.ss.sh kafka.tools.ConsumerOffsetChecker --zookeeper zk01:2181 --group testGroup  ● 查看某个Topic的详情  bin/kafka-topics.sh --topic test --describe --zookeeper zk01:2181  ● 对分区数进行修改  kafka-topics.sh --zookeeper zk01 --alter --partitions 15 --topic utopic |

使用flink消费kafka的消息

|  |
| --- |
| **import** java.util.Properties  **import** org.apache.flink.streaming.api.scala.{DataStream, StreamExecutionEnvironment} **import** org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer09 **import** org.apache.flink.streaming.util.serialization.SimpleStringSchema **import** org.apache.flink.api.scala.\_ */\*\*  \* Created by angel；  \*/* **object** DataSource\_kafka {  **def** main(args: Array[String]): Unit = {  *//1指定kafka数据流的相关信息* **val** zkCluster = **"hadoop01,hadoop02,hadoop03:2181"  val** kafkaCluster = **"hadoop01:9092,hadoop02:9092,hadoop03:9092"  val** kafkaTopicName = **"test"** *//2.创建流处理环境* **val** env = StreamExecutionEnvironment.*getExecutionEnvironment   //3.创建kafka数据流* **val** properties = **new** Properties()  properties.setProperty(**"bootstrap.servers"**, kafkaCluster)  properties.setProperty(**"zookeeper.connect"**, zkCluster)  properties.setProperty(**"group.id"**, kafkaTopicName)   **val** kafka09 = **new** FlinkKafkaConsumer09[String](kafkaTopicName,  **new** SimpleStringSchema(), properties)  *//4.添加数据源addSource(kafka09)* **val** text = env.addSource(kafka09).setParallelism(4)   */\*\*  \* test#CS#request http://b2c.csair.com/B2C40/query/jaxb/direct/query.ao?t=S&c1=HLN&c2=CTU&d1=2018-07-12&at=2&ct=2&inf=1#CS#POST#CS#application/x-www-form-urlencoded#CS#t=S&json={'adultnum':'1','arrcity':'NAY','childnum':'0','depcity':'KHH','flightdate':'2018-07-12','infantnum':'2'}#CS#http://b2c.csair.com/B2C40/modules/bookingnew/main/flightSelectDirect.html?t=R&c1=LZJ&c2=MZG&d1=2018-07-12&at=1&ct=2&inf=2#CS#123.235.193.25#CS#Mozilla/5.0 (Windows NT 5.1) AppleWebKit/537.1 (KHTML, like Gecko) Chrome/21.0.1180.89 Safari/537.1#CS#2018-01-19T10:45:13:578+08:00#CS#106.86.65.18#CS#cookie  \* \*/* **val** values: DataStream[ProcessedData] = text.map{  line =>  **var** encrypted = line  **val** values = encrypted.split(**"#CS#"**)  **val** valuesLength = values.length  **var** regionalRequest = **if**(valuesLength > 1) values(1) **else ""  val** requestMethod = **if** (valuesLength > 2) values(2) **else ""  val** contentType = **if** (valuesLength > 3) values(3) **else ""** *//Post提交的数据体* **val** requestBody = **if** (valuesLength > 4) values(4) **else ""** *//http\_referrer* **val** httpReferrer = **if** (valuesLength > 5) values(5) **else ""** *//客户端IP* **val** remoteAddr = **if** (valuesLength > 6) values(6) **else ""** *//客户端UA* **val** httpUserAgent = **if** (valuesLength > 7) values(7) **else ""** *//服务器时间的ISO8610格式* **val** timeIso8601 = **if** (valuesLength > 8) values(8) **else ""** *//服务器地址* **val** serverAddr = **if** (valuesLength > 9) values(9) **else ""** *//获取原始信息中的cookie字符串* **val** cookiesStr = **if** (valuesLength > 10) values(10) **else ""** *ProcessedData*(regionalRequest,  requestMethod,  contentType,  requestBody,  httpReferrer,  remoteAddr,  httpUserAgent,  timeIso8601,  serverAddr,  cookiesStr)  }  values.print()  **val** remoteAddr: DataStream[String] = values.map(line => line.remoteAddr)  remoteAddr.print()  *//5.触发运算* env.execute(**"flink-kafka-wordcunt"**)  } }  *//保存结构化数据* **case class** ProcessedData(regionalRequest: String,  requestMethod: String,  contentType: String,  requestBody: String,  httpReferrer: String,  remoteAddr: String,  httpUserAgent: String,  timeIso8601: String,  serverAddr: String,  cookiesStr: String  ) |

### Flink在流处理上常见的sink

#### 1：将数据sink到本地文件(参考批处理)

#### 2：Sink到本地集合(参考批处理)

#### 3：Sink到HDFS(参考批处理)

#### 4：sink到kafka

|  |  |
| --- | --- |
| **package** com.flink.DataStream  **import** java.util.Properties  **import** org.apache.flink.api.common.serialization.SimpleStringSchema **import** org.apache.flink.streaming.api.scala.{DataStream, StreamExecutionEnvironment} **import** org.apache.flink.streaming.connectors.kafka.{FlinkKafkaConsumer09, FlinkKafkaProducer09} **import** org.apache.flink.streaming.api.functions.source.SourceFunction **import** org.apache.flink.streaming.api.functions.source.SourceFunction.SourceContext **import** org.apache.flink.api.scala.\_ **import** org.apache.kafka.common.serialization.ByteArraySerializer */\*\*  \* Created by angel；  \*/* **object** DataSource\_kafka {  **def** main(args: Array[String]): Unit = {  *//1指定kafka数据流的相关信息* **val** zkCluster = **"hadoop01,hadoop02,hadoop03:2181"  val** kafkaCluster = **"hadoop01:9092,hadoop02:9092,hadoop03:9092"  val** kafkaTopicName = **"test"  val** sinkKafka = **"test2"** *//2.创建流处理环境* **val** env = StreamExecutionEnvironment.*getExecutionEnvironment   //3.创建kafka数据流* **val** properties = **new** Properties()  properties.setProperty(**"bootstrap.servers"**, kafkaCluster)  properties.setProperty(**"zookeeper.connect"**, zkCluster)  properties.setProperty(**"group.id"**, kafkaTopicName)   **val** kafka09 = **new** FlinkKafkaConsumer09[String](kafkaTopicName, **new** SimpleStringSchema(), properties)  *//4.添加数据源addSource(kafka09)* **val** text = env.addSource(kafka09).setParallelism(4)   */\*\*  \* test#CS#request http://b2c.csair.com/B2C40/query/jaxb/direct/query.ao?t=S&c1=HLN&c2=CTU&d1=2018-07-12&at=2&ct=2&inf=1#CS#POST#CS#application/x-www-form-urlencoded#CS#t=S&json={'adultnum':'1','arrcity':'NAY','childnum':'0','depcity':'KHH','flightdate':'2018-07-12','infantnum':'2'}#CS#http://b2c.csair.com/B2C40/modules/bookingnew/main/flightSelectDirect.html?t=R&c1=LZJ&c2=MZG&d1=2018-07-12&at=1&ct=2&inf=2#CS#123.235.193.25#CS#Mozilla/5.0 (Windows NT 5.1) AppleWebKit/537.1 (KHTML, like Gecko) Chrome/21.0.1180.89 Safari/537.1#CS#2018-01-19T10:45:13:578+08:00#CS#106.86.65.18#CS#cookie  \* \*/* **val** values: DataStream[ProcessedData] = text.map{  line =>  **var** encrypted = line  **val** values = encrypted.split(**"#CS#"**)  **val** valuesLength = values.length  **var** regionalRequest = **if**(valuesLength > 1) values(1) **else ""  val** requestMethod = **if** (valuesLength > 2) values(2) **else ""  val** contentType = **if** (valuesLength > 3) values(3) **else ""** *//Post提交的数据体* **val** requestBody = **if** (valuesLength > 4) values(4) **else ""** *//http\_referrer* **val** httpReferrer = **if** (valuesLength > 5) values(5) **else ""** *//客户端IP* **val** remoteAddr = **if** (valuesLength > 6) values(6) **else ""** *//客户端UA* **val** httpUserAgent = **if** (valuesLength > 7) values(7) **else ""** *//服务器时间的ISO8610格式* **val** timeIso8601 = **if** (valuesLength > 8) values(8) **else ""** *//服务器地址* **val** serverAddr = **if** (valuesLength > 9) values(9) **else ""** *//获取原始信息中的cookie字符串* **val** cookiesStr = **if** (valuesLength > 10) values(10) **else ""** *ProcessedData*(regionalRequest,  requestMethod,  contentType,  requestBody,  httpReferrer,  remoteAddr,  httpUserAgent,  timeIso8601,  serverAddr,  cookiesStr)   }  values.print()  **val** remoteAddr: DataStream[String] = values.map(line => line.remoteAddr)  remoteAddr.print()  *//****TODO sink到kafka* val** p: Properties = **new** Properties  p.setProperty(**"bootstrap.servers"**, **"hadoop01:9092,hadoop02:9092,hadoop03:9092"**)  p.setProperty(**"key.serializer"**, *classOf*[ByteArraySerializer].getName)  p.setProperty(**"value.serializer"**, *classOf*[ByteArraySerializer].getName)  **val** sink = **new** FlinkKafkaProducer09[String](sinkKafka, **new** SimpleStringSchema(), properties)  remoteAddr.addSink(sink)  *//5.触发运算* env.execute(**"flink-kafka-wordcunt"**)  } } *//保存结构化数据* **case class** ProcessedData(regionalRequest: String,  requestMethod: String,  contentType: String,  requestBody: String,  httpReferrer: String,  remoteAddr: String,  httpUserAgent: String,  timeIso8601: String,  serverAddr: String,  cookiesStr: String  ) |  |

### 基于mysql的sink和source

#### 1：基于mysql的source操作：

|  |
| --- |
| **object** MysqlSource {  **def** main(args: Array[String]): Unit = {  **val** env = StreamExecutionEnvironment.*getExecutionEnvironment* **val** source: DataStream[Student] = env.addSource(**new** SQL\_source)  source.print()  env.execute()  } } **class** SQL\_source **extends** RichSourceFunction[Student]{  **private var** *connection*: Connection = **null  private var** *ps*: PreparedStatement = **null   override def** open(parameters: Configuration): Unit = {  **val** driver = **"com.mysql.jdbc.Driver"  val** url = **"jdbc:mysql://hadoop01:3306/test"  val** username = **"root"  val** password = **"root"** Class.*forName*(driver)  *connection* = DriverManager.*getConnection*(url, username, password)  **val** sql = **"select stuid , stuname , stuaddr , stusex from Student"** *ps* = *connection*.prepareStatement(sql)  }   **override def** close(): Unit = {  **if**(*connection* != **null**){  *connection*.close()  }  **if**(*ps* != **null**){  *ps*.close()  }  }    **override def** run(sourceContext: SourceContext[Student]): Unit = {  **val** queryRequest = *ps*.executeQuery()  **while** (queryRequest.next()){  **val** stuid = queryRequest.getInt(**"stuid"**)  **val** stuname = queryRequest.getString(**"stuname"**)  **val** stuaddr = queryRequest.getString(**"stuaddr"**)  **val** stusex = queryRequest.getString(**"stusex"**)  **val** stu = **new** Student(stuid , stuname , stuaddr , stusex)  sourceContext.collect(stu)  }  }  **override def** cancel(): Unit = {} }   **case class** Student(stuid:Int , stuname:String , stuaddr:String , stusex:String){**override def** toString: String = {  **"stuid:"**+stuid+**" stuname:"**+stuname+**" stuaddr:"**+stuaddr+**" stusex:"**+stusex  } } |

#### 2：基于mysql的sink操作

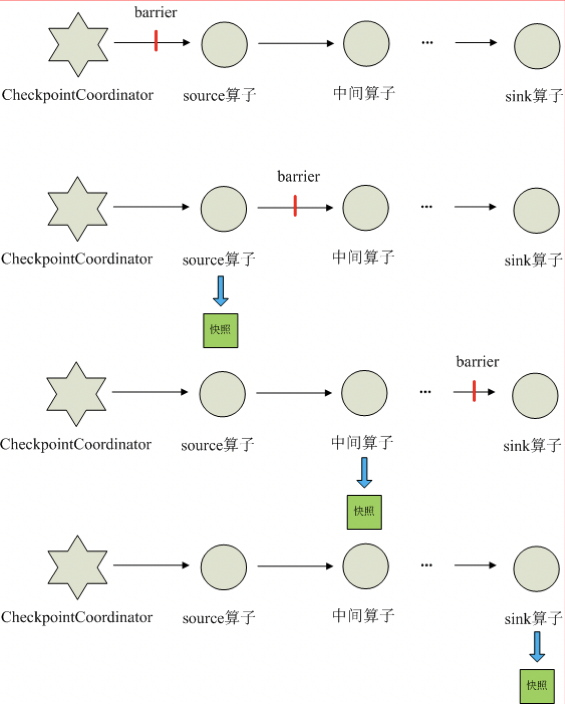
|  |
| --- |
| **object** MysqlSink {  **def** main(args: Array[String]): Unit = {  *//1.创建流执行环境* **val** env = StreamExecutionEnvironment.*getExecutionEnvironment  //2.准备数据* **val** dataStream:DataStream[Student] = env.fromElements(  *Student*(8, **"xiaoming"**, **"beijing biejing"**, **"female"**) *// Student(6, "daming", "tainjing tianjin", "male "), // Student(7, "daqiang ", "shanghai shanghai", "female")* )   *//3.将数据写入到自定义的sink中（这里是mysql）* dataStream.addSink(**new** StudentSinkToMysql)  *//4.触发流执行* env.execute()  } }  **class** StudentSinkToMysql **extends** RichSinkFunction[Student]{  **private var** *connection*:Connection = **null  private var** *ps*:PreparedStatement = **null   override def** open(parameters: Configuration): Unit = {  **val** driver = **"com.mysql.jdbc.Driver"  val** url = **"jdbc:mysql://hadoop01:3306/test"  val** username = **"root"  val** password = **"root"** *//1:加载驱动* Class.*forName*(driver)  *//2：创建连接  connection* = DriverManager.*getConnection*(url , username , password)  **val** sql = **"insert into Student(stuid , stuname , stuaddr , stusex) values(?,?,?,?);"** *//3:获得执行语句  ps* = *connection*.prepareStatement(sql)  }   *//关闭连接操作* **override def** close(): Unit = {  **if**(*connection* != **null**){  *connection*.close()  }  **if**(*ps* != **null**){  *ps*.close()  }  }  *//每个元素的插入，都要触发一次invoke，这里主要进行invoke插入* **override def** invoke(stu: Student): Unit = {  **try**{  *//4.组装数据，执行插入操作  ps*.setInt(1, stu.stuid)  *ps*.setString(2, stu.stuname)  *ps*.setString(3, stu.stuaddr)  *ps*.setString(4, stu.stusex)  *ps*.executeUpdate()  }**catch** {  **case** e:Exception => *println*(e.getMessage)  }  } } |

## Flink的容错

#### Checkpoint介绍

checkpoint机制是Flink可靠性的基石，可以保证Flink集群在某个算子因为某些原因(如 异常退出)出现故障时，能够将整个应用流图的状态恢复到故障之前的某一状态，保 证应用流图状态的一致性。Flink的checkpoint机制原理来自“Chandy-Lamport algorithm”算法。

每个需要checkpoint的应用在启动时，Flink的JobManager为其创建一个 CheckpointCoordinator，CheckpointCoordinator全权负责本应用的快照制作。



1. CheckpointCoordinator周期性的向该流应用的所有source算子发送barrier。

2.当某个source算子收到一个barrier时，便暂停数据处理过程，然后将自己的当前状 态制作成快照，并保存到指定的持久化存储中，最后向CheckpointCoordinator报告 自己快照制作情况，同时向自身所有下游算子广播该barrier，恢复数据处理

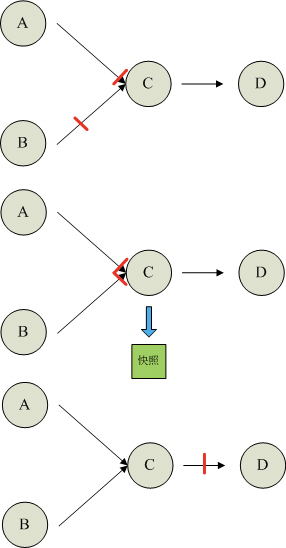
3.下游算子收到barrier之后，会暂停自己的数据处理过程，然后将自身的相关状态制作成快照，并保存到指定的持久化存储中，最后向CheckpointCoordinator报告自身 快照情况，同时向自身所有下游算子广播该barrier，恢复数据处理。

4. 每个算子按照步骤3不断制作快照并向下游广播，直到最后barrier传递到sink算子，快照制作完成。

5. 当CheckpointCoordinator收到所有算子的报告之后，认为该周期的快照制作成功; 否则，如果在规定的时间内没有收到所有算子的报告，则认为本周期快照制作失败

如果一个算子有两个输入源，则暂时阻塞先收到barrier的输入源，等到第二个输入源相 同编号的barrier到来时，再制作自身快照并向下游广播该barrier。具体如下图所示

两个输入源 checkpoint 过程



1. 假设算子C有A和B两个输入源

2. 在第i个快照周期中，由于某些原因(如处理时延、网络时延等)输入源A发出的 barrier先到来，这时算子C暂时将输入源A的输入通道阻塞，仅收输入源B的数据。

3. 当输入源B发出的barrier到来时，算子C制作自身快照并向CheckpointCoordinator报 告自身的快照制作情况，然后将两个barrier合并为一个，向下游所有的算子广播。

当由于某些原因出现故障时，CheckpointCoordinator通知流图上所有算子统一恢复到某 个周期的checkpoint状态，然后恢复数据流处理。分布式checkpoint机制保证了数据仅被 处理一次(Exactly Once)。

#### 持久化存储

目前，Checkpoint持久化存储可以使用如下三种:

##### MemStateBackend

该持久化存储主要将快照数据保存到JobManager的内存中，仅适合作为测试以及

快照的数据量非常小时使用，并不推荐用作大规模商业部署。

##### FsStateBackend

该持久化存储主要将快照数据保存到文件系统中，目前支持的文件系统主要是 HDFS和本地文件。如果使用HDFS，则初始化FsStateBackend时，需要传入以 “hdfs://”开头的路径(即: new FsStateBackend("hdfs:///hacluster/checkpoint"))， 如果使用本地文件，则需要传入以“file://”开头的路径(即:new FsStateBackend("file:///Data"))。在分布式情况下，不推荐使用本地文件。如果某 个算子在节点A上失败，在节点B上恢复，使用本地文件时，在B上无法读取节点 A上的数据，导致状态恢复失败。

##### RocksDBStateBackend

RocksDBStatBackend介于本地文件和HDFS之间，平时使用RocksDB的功能，将数 据持久化到本地文件中，当制作快照时，将本地数据制作成快照，并持久化到 FsStateBackend中(FsStateBackend不必用户特别指明，只需在初始化时传入HDFS 或本地路径即可，如new RocksDBStateBackend("hdfs:///hacluster/checkpoint")或new RocksDBStateBackend("file:///Data"))。

如果用户使用自定义窗口(window)，不推荐用户使用RocksDBStateBackend。在自 定义窗口中，状态以ListState的形式保存在StatBackend中，如果一个key值中有多 个value值，则RocksDB读取该种ListState非常缓慢，影响性能。用户可以根据应用 的具体情况选择FsStateBackend+HDFS或RocksStateBackend+HDFS。

##### 语法

|  |
| --- |
| **val** env **=** **StreamExecutionEnvironment.**getExecutionEnvironment**()**  *// start a checkpoint every 1000 ms*  env**.**enableCheckpointing**(**1000**)**  *// advanced options:*  *// 设置checkpoint的执行模式，最多执行一次或者至少执行一次*  env**.**getCheckpointConfig**.**setCheckpointingMode**(CheckpointingMode.EXACTLY\_ONCE)**  *// 设置checkpoint的超时时间*  env**.**getCheckpointConfig**.**setCheckpointTimeout**(**60000**)**  *// 如果在只做快照过程中出现错误，是否让整体任务失败：true是 false不是*  env**.**getCheckpointConfig**.**setFailTasksOnCheckpointingErrors**(false)**  *//设置同一时间有多少 个checkpoint可以同时执行*  env**.**getCheckpointConfig**.**setMaxConcurrentCheckpoints**(**1**)** |

##### 例子

###### 需求

假定用户需要每隔1秒钟需要统计4秒中窗口中数据的量，然后对统计的结果值进行checkpoint处理

###### 数据规划

1. 使用自定义算子每秒钟产生大约10000条数据。
2. 产生的数据为一个四元组(Long，String，String，Integer)—------(id,name,info,count)。
3. 数据经统计后，统计结果打印到终端输出。
4. 打印输出的结果为Long类型的数据。

###### 开发思路

1. source算子每隔1秒钟发送10000条数据，并注入到Window算子中。

2. window算子每隔1秒钟统计一次最近4秒钟内数据数量。

3. 每隔1秒钟将统计结果打印到终端

4. 每隔6秒钟触发一次checkpoint，然后将checkpoint的结果保存到HDFS中。

|  |  |
| --- | --- |
| *//发送数据形式* **case class** SEvent(id: Long, name: String, info: String, count: Int)  **class** SEventSourceWithChk **extends** RichSourceFunction[SEvent]{  **private var** *count* = 0L  **private var** *isRunning* = **true  private val** *alphabet* = **"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWZYX0987654321"** *// 任务取消时调用* **override def** cancel(): Unit = {  *isRunning* = **false** }  *//// source算子的逻辑，即:每秒钟向流图中注入10000个元组* **override def** run(sourceContext: SourceContext[SEvent]): Unit = {  **while**(*isRunning*) {  **for** (i <- 0 until 10000) {  sourceContext.collect(*SEvent*(1, **"hello-"**+*count*, *alphabet*,1))  *count* += 1L  }  Thread.*sleep*(1000)  }  } }  */\*\* 该段代码是流图定义代码，具体实现业务流程，另外，代码中窗口的触发时间使 用了event time。  \*/* **object** FlinkEventTimeAPIChkMain {  **def** main(args: Array[String]): Unit ={  **val** env = StreamExecutionEnvironment.*getExecutionEnvironment* env.setStateBackend(**new** FsStateBackend(**"hdfs://hadoop01:9000/flink-checkpoint/checkpoint/"**))  env.getCheckpointConfig.setCheckpointingMode(CheckpointingMode.*EXACTLY\_ONCE*)  env.getCheckpointConfig.setCheckpointInterval(6000)  env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)  *// 应用逻辑* **val** source: DataStream[SEvent] = env.addSource(**new** SEventSourceWithChk)  source.assignTimestampsAndWatermarks(**new** AssignerWithPeriodicWatermarks[SEvent] {  *// 设置watermark* **override def** getCurrentWatermark: Watermark = {  **new** Watermark(System.*currentTimeMillis*())  }  *// 给每个元组打上时间戳* **override def** extractTimestamp(t: SEvent, l: Long): Long = {  System.*currentTimeMillis*()  }  })  .keyBy(0)  .window(SlidingEventTimeWindows.*of*(Time.*seconds*(4), Time.*seconds*(1)))  .apply(**new** WindowStatisticWithChk)  .print()  env.execute()  } }    *//该数据在算子制作快照时用于保存到目前为止算子记录的数据条数。 // 用户自定义状态* **class** UDFState **extends** Serializable{  **private var** *count* = 0L  *// 设置用户自定义状态* **def** setState(s: Long) = *count* = s  *// 获取用户自定状态* **def** getState = *count* }   *//该段代码是window算子的代码，每当触发计算时统计窗口中元组数量。* **class** WindowStatisticWithChk **extends** WindowFunction[SEvent, Long, Tuple, TimeWindow] **with** ListCheckpointed[UDFState]{  **private var** *total* = 0L   *// window算子的实现逻辑，即:统计window中元组的数量* **override def** apply(key: Tuple, window: TimeWindow, input: Iterable[SEvent], out: Collector[Long]): Unit = {  **var** count = 0L  **for** (event <- input) {  count += 1L  }  *total* += count  out.collect(count)  }  *// 从自定义快照中恢复状态* **override def** restoreState(state: util.List[UDFState]): Unit = {  **val** udfState = state.get(0)  *total* = udfState.getState  }   *// 制作自定义状态快照* **override def** snapshotState(checkpointId: Long, timestamp: Long): util.List[UDFState] = {  **val** udfList: util.ArrayList[UDFState] = **new** util.ArrayList[UDFState]  **val** udfState = **new** UDFState  udfState.setState(*total*)  udfList.add(udfState)  udfList  } } |  |

## flink-SQL

Table API和SQL捆绑在flink-table Maven工件中。必须将以下依赖项添加到你的项目才能使用Table API和SQL：

<dependency>

<groupId>org.apache.flink</groupId>

<artifactId>flink-table\_2.11</artifactId>

<version>1.5.0</version>

</dependency>

另外，你需要为Flink的Scala批处理或流式API添加依赖项。对于批量查询，您需要添加：

<dependency>

<groupId>org.apache.flink</groupId>

<artifactId>flink-scala\_2.11</artifactId>

<version>1.5.0</version>

</dependency>

### Table API和SQL程序的结构

Flink的批处理和流处理的Table API和SQL程序遵循相同的模式；

所以我们只需要使用一种来演示即可

要想执行flink的SQL语句，首先需要获取SQL的执行环境：

两种方式（batch和streaming）：

|  |
| --- |
| *// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\**  *// STREAMING QUERY*  *// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\**  **val** sEnv **=** **StreamExecutionEnvironment.**getExecutionEnvironment  *// create a TableEnvironment for streaming queries*  **val** sTableEnv **=** **TableEnvironment.**getTableEnvironment**(**sEnv**)**  *// \*\*\*\*\*\*\*\*\*\*\**  *// BATCH QUERY*  *// \*\*\*\*\*\*\*\*\*\*\**  **val** bEnv **=** **ExecutionEnvironment.**getExecutionEnvironment  *// create a TableEnvironment for batch queries*  **val** bTableEnv **=** **TableEnvironment.**getTableEnvironment**(**bEnv**)** |

通过getTableEnvironment可以获取TableEnviromment；这个TableEnviromment是Table API和SQL集成的核心概念。它负责：

* 在内部目录中注册一个表
* 注册外部目录
* 执行SQL查询
* 注册用户定义的（标量，表格或聚合）函数
* 转换DataStream或DataSet成Table
* 持有一个ExecutionEnvironment或一个参考StreamExecutionEnvironment

#### 1、在内部目录中注册一个表

TableEnvironment维护一个按名称注册的表的目录。有两种类型的表格，*输入表格*和*输出表格*。

输入表可以在Table API和SQL查询中引用并提供输入数据。输出表可用于将表API或SQL查询的结果发送到外部系统

输入表可以从各种来源注册：

* 现有Table对象，通常是表API或SQL查询的结果。
* TableSource，它访问外部数据，例如文件，数据库或消息传递系统。
* DataStream或DataSet来自DataStream或DataSet程序。

输出表可以使用注册TableSink。

##### 1、注册一个表

|  |
| --- |
| *// get a TableEnvironment*  **val** tableEnv **=** **TableEnvironment.**getTableEnvironment**(**env**)**  *// register the Table projTable as table "projectedX"*  tableEnv**.**registerTable**(**"projectedTable"**,** projTable**)**  *// Table is the result of a simple projection query*  **val** projTable**:** **Table** **=** tableEnv**.**scan**(**"projectedTable "**).**select**(...)** |

##### 2、注册一个tableSource

TableSource提供对存储在诸如数据库（MySQL，HBase等），具有特定编码（CSV，Apache [Parquet，Avro，ORC]，...）的文件的存储系统中的外部数据的访问或者消息传送系统（Apache Kafka，RabbitMQ，...）

|  |
| --- |
| *// get a TableEnvironment*  **val** tableEnv **=** **TableEnvironment.**getTableEnvironment**(**env**)**  *// create a TableSource*  **val** csvSource**:** **TableSource** **=** **new** **CsvTableSource(**"/path/to/file"**,** **...)**  *// register the TableSource as table "CsvTable"* tableEnv**.**registerTableSource**(**"CsvTable"**,** csvSource**)** |

##### 3、注册一个tableSink

注册TableSink可用于将[表API或SQL查询的结果](https://ci.apache.org/projects/flink/flink-docs-release-1.5/dev/table/common.html" \l "emit-a-table)发送到外部存储系统，如数据库，键值存储，消息队列或文件系统（使用不同的编码，例如CSV，Apache [Parquet ，Avro，ORC]，...）

|  |
| --- |
| *// get a TableEnvironment*  **val** tableEnv **=** **TableEnvironment.**getTableEnvironment**(**env**)**  *// create a TableSink*  **val** csvSink**:** **TableSink** **=** **new** **CsvTableSink(**"/path/to/file"**,** **...)**  *// define the field names and types*  **val** fieldNames**:** **Array[String]** **=** **Array(**"a"**,** "b"**,** "c"**)**  **val** fieldTypes**:** **Array[TypeInformation[\_]]** **=** **Array(Types.INT,** **Types.STRING,** **Types.LONG)**  *// register the TableSink as table "CsvSinkTable"*  tableEnv**.**registerTableSink**(**"CsvSinkTable"**,** fieldNames**,** fieldTypes**,** csvSink**)** |

例子：

|  |
| --- |
| *//创建batch执行环境* **val** env = ExecutionEnvironment.*getExecutionEnvironment  //创建table环境用于batch查询* **val** tableEnvironment = TableEnvironment.*getTableEnvironment*(env)  *//加载外部数据* **val** csvTableSource = CsvTableSource.*builder*()  .path(**"data1.csv"**)*//文件路径* .field(**"id"** , Types.*INT*)*//第一列数据* .field(**"name"** , Types.*STRING*)*//第二列数据* .field(**"age"** , Types.*INT*)*//第三列数据* .fieldDelimiter(**","**)*//列分隔符，默认是"，"* .lineDelimiter(**"\n"**)*//换行符* .ignoreFirstLine()*//忽略第一行* .ignoreParseErrors()*//忽略解析错误* .build()  *//将外部数据构建成表* tableEnvironment.registerTableSource(**"tableA"** , csvTableSource)  *//****TODO 1：使用table方式查询数据* val** table = tableEnvironment.scan(**"tableA"**).select(**"id , name , age"**).filter(**"name == 'lisi'"**)  *//将数据写出去* table.writeToSink(**new** CsvTableSink(**"bbb"** , **","** , 1 , FileSystem.WriteMode.*OVERWRITE*))  *//****TODO 2：使用sql方式*** *// val sqlResult = tableEnvironment.sqlQuery("select id,name,age from tableA where id > 0 order by id limit 2") //// //将数据写出去 // sqlResult.writeToSink(new CsvTableSink("aaaaaa.csv", ",", 1, FileSystem.WriteMode.OVERWRITE))* env.execute() |

#### 2、Table和DataStream和DataSet的集成

##### 1：将DataStream或DataSet转换为表格

在上面的例子讲解中，直接使用的是：registerTableSource注册表

对于flink来说，还有更灵活的方式：比如直接注册DataStream或者DataSet转换为一张表。

然后DataStream或者DataSet就相当于表，这样可以继续使用SQL来操作流或者批次的数据

语法：

*// get TableEnvironment*

*// registration of a DataSet is equivalent*

*Env:DataStream*

**val** tableEnv **=** **TableEnvironment.**getTableEnvironment**(**env**)**

**val** stream**:** **DataStream[(Long**, **String)]** **=** **...**

*// register the DataStream as Table "myTable" with fields "f0", "f1"*

tableEnv**.**registerDataStream**(**"myTable"**,** stream**)**

例子：

|  |
| --- |
| **object** SQLToDataSetAndStreamSet {  **def** main(args: Array[String]): Unit = {   *// set up execution environment* **val** env = StreamExecutionEnvironment.*getExecutionEnvironment* **val** tEnv = TableEnvironment.*getTableEnvironment*(env)  *//构造数据* **val** orderA: DataStream[Order] = env.fromCollection(*Seq*(  *Order*(1L, **"beer"**, 3),  *Order*(1L, **"diaper"**, 4),  *Order*(3L, **"rubber"**, 2)))  **val** orderB: DataStream[Order] = env.fromCollection(*Seq*(  *Order*(2L, **"pen"**, 3),  *Order*(2L, **"rubber"**, 3),  *Order*(4L, **"beer"**, 1)))  *// 根据数据注册表* tEnv.registerDataStream(**"OrderA"**, orderA)  tEnv.registerDataStream(**"OrderB"**, orderB)  *// union the two tables* **val** result = tEnv.sqlQuery(  **"SELECT \* FROM OrderA WHERE amount > 2 UNION ALL "** +  **"SELECT \* FROM OrderB WHERE amount < 2"**)  result.writeToSink(**new** CsvTableSink(**"ccc"** , **","** , 1 , FileSystem.WriteMode.*OVERWRITE*))  env.execute()  } } **case class** Order(user: Long, product: String, amount: Int) |

#### 3、将表转换为DataStream或DataSet

A Table可以转换成a DataStream或DataSet。通过这种方式，可以在Table API或SQL查询的结果上运行自定义的DataStream或DataSet程序

###### 1：将表转换为DataStream

有两种模式可以将 Table转换为DataStream：

1：**Append Mode**

**将一个表附加到流上**

**2：Retract Mode**

**将表转换为流**

**语法格式：**

|  |
| --- |
| **// get TableEnvironment.**  **// registration of a DataSet is equivalent**  **// ge val tableEnv = TableEnvironment.getTableEnvironment(env)**  **// Table with two fields (String name, Integer age)**  **val table: Table = ...**  **// convert the Table into an append DataStream of Row**  **val dsRow: DataStream[Row] = tableEnv.toAppendStream[Row](table)**  **// convert the Table into an append DataStream of Tuple2[String, Int]**  **val dsTuple: DataStream[(String, Int)] dsTuple =**  **tableEnv.toAppendStream[(String, Int)](table)**  **// convert the Table into a retract DataStream of Row.**  **// A retract stream of type X is a DataStream[(Boolean, X)].**  **// The boolean field indicates the type of the change.**  **// True is INSERT, false is DELETE.**  **val retractStream: DataStream[(Boolean, Row)] = tableEnv.toRetractStream[Row](table)** |

例子:

|  |
| --- |
| **object** TableTODataSet\_DataStream {  **def** main(args: Array[String]): Unit = {  *//构造数据，转换为table* **val** data = *List*(  *Peoject*(1L, 1, **"Hello"**),  *Peoject*(2L, 2, **"Hello"**),  *Peoject*(3L, 3, **"Hello"**),  *Peoject*(4L, 4, **"Hello"**),  *Peoject*(5L, 5, **"Hello"**),  *Peoject*(6L, 6, **"Hello"**),  *Peoject*(7L, 7, **"Hello World"**),  *Peoject*(8L, 8, **"Hello World"**),  *Peoject*(8L, 8, **"Hello World"**),  *Peoject*(20L, 20, **"Hello World"**))   **val** env = StreamExecutionEnvironment.*getExecutionEnvironment* env.setParallelism(1)  **val** tEnv = TableEnvironment.*getTableEnvironment*(env)  **val** stream = env.fromCollection(data)  **val** table: Table = tEnv.fromDataStream(stream)  *//****TODO 将table转换为DataStream----将一个表附加到流上Append Mode* val** appendStream: DataStream[Peoject] = tEnv.toAppendStream[Peoject](table)  *//****TODO 将表转换为流Retract Mode true代表添加消息，false代表撤销消息* val** retractStream: DataStream[(Boolean, Peoject)] = tEnv.toRetractStream[Peoject](table)  retractStream.print()  env.execute()   } }  **case class** Peoject(user: Long, index: Int, content: String) |

##### 2：将表转换为DataSet

语法格式

|  |
| --- |
| *// get TableEnvironment*  *// registration of a DataSet is equivalent*  **val** tableEnv **=** **TableEnvironment.**getTableEnvironment**(**env**)**  *// Table with two fields (String name, Integer age)*  **val** table**:** **Table** **=** **...**  *// convert the Table into a DataSet of Row*  **val** dsRow**:** **DataSet[Row]** **=** tableEnv**.**toDataSet**[Row](**table**)**  *// convert the Table into a DataSet of Tuple2[String, Int]*  **val** dsTuple**:** **DataSet[(String**, **Int)]** **=** tableEnv**.**toDataSet**[(String**, **Int)](**table**)** |

例子：

|  |
| --- |
| **case class** Peoject(user: Long, index: Int, content: String)  **object** TableTODataSet{  **def** main(args: Array[String]): Unit = {   *//构造数据，转换为table* **val** data = *List*(  *Peoject*(1L, 1, **"Hello"**),  *Peoject*(2L, 2, **"Hello"**),  *Peoject*(3L, 3, **"Hello"**),  *Peoject*(4L, 4, **"Hello"**),  *Peoject*(5L, 5, **"Hello"**),  *Peoject*(6L, 6, **"Hello"**),  *Peoject*(7L, 7, **"Hello World"**),  *Peoject*(8L, 8, **"Hello World"**),  *Peoject*(8L, 8, **"Hello World"**),  *Peoject*(20L, 20, **"Hello World"**))  *//初始化环境，加载table数据* **val** env = ExecutionEnvironment.*getExecutionEnvironment* env.setParallelism(1)  **val** tableEnvironment = TableEnvironment.*getTableEnvironment*(env)  **val** collection: DataSet[Peoject] = env.fromCollection(data)  **val** table: Table = tableEnvironment.fromDataSet(collection)  *//****TODO 将table转换为dataSet* val** toDataSet: DataSet[Peoject] = tableEnvironment.toDataSet[Peoject](table)  toDataSet.print() *// env.execute()* } } |

# 案例：数据同步

企业运维的数据库最常见的是mysql；但是mysql有个缺陷：当数据量达到千万条的时候，mysql的相关操作会变的非常迟缓；

如果这个时候有需求需要实时展示数据；对于mysql来说是一种灾难；而且对于mysql来说，同一时间还要给多个开发人员和用户操作；

所以经过调研，将mysql数据实时同步到hbase中；

最开始使用的架构方案：

Mysql---logstash—kafka---sparkStreaming---hbase---web

Mysql—sqoop---hbase---web

但是无论使用logsatsh还是使用kafka，都避免不了一个尴尬的问题：

他们在导数据过程中需要去mysql中做查询操作：

比如logstash：

|  |
| --- |
|  |

比如sqoop：

|  |
| --- |
|  |

不可避免的，都需要去sql中查询出相关数据，然后才能进行同步；这样对于mysql来说本身就是增加负荷操作；

所以我们真正需要考虑的问题是：有没有什么方法，能将mysql数据实时同步到hbase；但是不增加mysql的负担；

答案是有的：可以使用canal或者maxwell来解析mysql的binlog日志

那么之前的架构就需要改动了：

Mysql---canal—kafka—flink—hbase—web

## 开发

### 第一步：开启mysql的binlog日志

Mysql的binlog日志作用是用来记录mysql内部增删等对mysql数据库有更新的内容的记录（对数据库的改动），对数据库的查询select或show等不会被binlog日志记录;主要用于数据库的主从复制以及增量恢复。

mysql的binlog日志必须打开log-bin功能才能生存binlog日志

-rw-rw---- 1 mysql mysql   669 8月  10 21:29 mysql-bin.000001

-rw-rw---- 1 mysql mysql   126 8月  10 22:06 mysql-bin.000002

-rw-rw---- 1 mysql mysql 11799 8月  15 18:17 mysql-bin.000003

（1）：修改/etc/my.cnf,在里面添加如下内容

|  |
| --- |
| [mysqld]  log-bin=/var/lib/mysql/mysql-bin 【binlog日志存放路径】  binlog-format=ROW 【日志中会记录成每一行数据被修改的形式】  server\_id=1 【指定当前机器的服务ID（如果是集群，不能重复）】 |

（2）：配置完毕之后，登录mysql，输入如下命令：

show variables like ‘%log\_bin%’



出现如下形式，代表binlog开启；

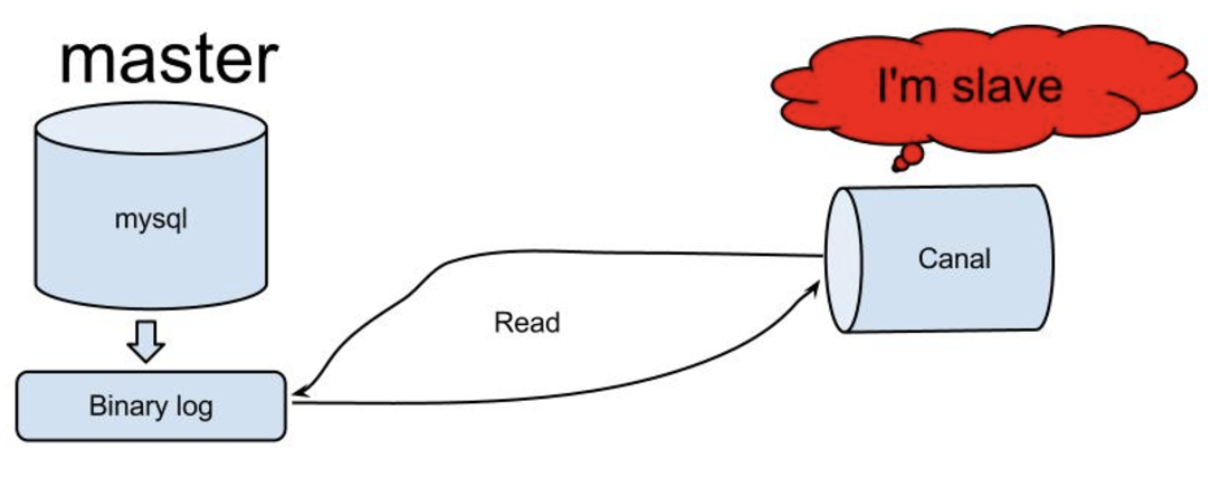
### 第二步：安装canal

#### Canal介绍

canal是阿里巴巴旗下的一款开源项目，纯Java开发。基于数据库增量日志解析，提供增量数据订阅&消费，目前主要支持了MySQL（也支持mariaDB）。

起源：早期，阿里巴巴B2B公司因为存在杭州和美国双机房部署，存在跨机房同步的业务需求。不过早期的数据库同步业务，主要是基于trigger的方式获取增量变更，不过从2010年开始，阿里系公司开始逐步的尝试基于数据库的日志解析，获取增量变更进行同步，由此衍生出了增量订阅&消费的业务，从此开启了一段新纪元。

工作原理



原理相对比较简单：

1、canal模拟mysql slave的交互协议，伪装自己为mysql slave，向mysql master发送dump协议

2、mysql master收到dump请求，开始推送binary log给slave(也就是canal)

3、canal解析binary log对象(原始为byte流)

#### 使用canal解析binlog，数据落地到kafka

##### (1):解压安装包：canal.deployer-1.0.23.tar.gz

tar -zxvf canal.deployer-1.0.23.tar.gz -C /export/servers/canal

修改配置文件：

vim /export/servers/canal/conf/example/instance.properties



##### (2):编写canal代码

仅仅安装了canal是不够的；canal从架构的意义上来说相当于mysql的“从库”，此时还并不能将binlog解析出来实时转发到kafka上，因此需要进一步开发canal代码；

Canal已经帮我们提供了示例代码，只需要根据需求稍微更改即可；

Canal提供的代码：

|  |
| --- |
| https://github.com/alibaba/canal/wiki/ClientExample |

上面的代码中可以解析出binlog日志，但是没有将数据落地到kafka的代码逻辑，所以我们还需要添加将数据落地kafka的代码；

Maven导入依赖：

|  |
| --- |
| <**dependency**>  <**groupId**>com.alibaba.otter</**groupId**>  <**artifactId**>canal.client</**artifactId**>  <**version**>1.0.25</**version**> </**dependency**>  *<!-- https://mvnrepository.com/artifact/org.apache.kafka/kafka -->* <**dependency**>  <**groupId**>org.apache.kafka</**groupId**>  <**artifactId**>kafka\_2.11</**artifactId**>  <**version**>0.9.0.1</**version**> </**dependency**> |

代码：

|  |
| --- |
| **import** com.alibaba.otter.canal.client.CanalConnector; **import** com.alibaba.otter.canal.client.CanalConnectors; **import** com.alibaba.otter.canal.protocol.CanalEntry; **import** com.alibaba.otter.canal.protocol.Message; **import** com.google.protobuf.InvalidProtocolBufferException;  **import** java.net.InetSocketAddress; **import** java.util.ArrayList; **import** java.util.List; **import** java.util.UUID;  */\*\*  \* Created by angel；  \*/* **public class** Canal\_Client {  **public static void** main(String[] args) {  *//****TODO 1:连接cnnal*** CanalConnector connector = CanalConnectors.*newSingleConnector*(**new** InetSocketAddress(**"hadoop01"** , 11111) , **"example"** , **"root"** , **"root"**);  **int** batchSize = 100 ;  **int** emptyCount = 1 ;  **try**{  connector.connect();  connector.subscribe(**".\*\\..\*"**);  connector.rollback();  **int** totalEmptyCount = 120 ;  **while** (totalEmptyCount > emptyCount){  **final** Message message = connector.getWithoutAck(batchSize);  **final long** batchid = message.getId();  **final int** size = message.getEntries().size();  **if**(batchid == -1 || size ==0){  System.***out***.println(**"暂时没有数据 :"**+emptyCount);  **try** {  Thread.*sleep*(1000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }**else**{  *//****TODO 解析binlog*** *Analysis*(message.getEntries() , emptyCount);  emptyCount++;  }  }  }**catch**(Exception e){  e.printStackTrace();  }**finally** {  connector.disconnect();  }   }  *//****TODO 3：将解析出来的数据区分好：增 删 改 发送到kafka* public static void** Analysis(List<CanalEntry.Entry> entries , **int** emptyCount){  **for**(CanalEntry.Entry entry : entries){  **if**(entry.getEntryType() == CanalEntry.EntryType.***TRANSACTIONBEGIN*** || entry.getEntryType() == CanalEntry.EntryType.***TRANSACTIONEND***){  **continue**;  }  **try** {  **final** CanalEntry.RowChange rowChange = CanalEntry.RowChange.*parseFrom*(entry.getStoreValue());  **final** CanalEntry.EventType eventType = rowChange.getEventType();  **final** String logfileName = entry.getHeader().getLogfileName();  **final long** logfileOffset = entry.getHeader().getLogfileOffset();  **final** String dbname = entry.getHeader().getSchemaName();  **final** String tableName = entry.getHeader().getTableName();  **for**(CanalEntry.RowData rowData : rowChange.getRowDatasList()){  *//区分增删改操作，然后发送给kafka* **if**(eventType == CanalEntry.EventType.***DELETE***){  *//删除操作* System.***out***.println(**"=======删除操作======="**);  *dataDetails*(rowData.getAfterColumnsList() , logfileName , logfileOffset , dbname , tableName , emptyCount);  }**else if** (eventType == CanalEntry.EventType.***INSERT***){  *//插入操作* System.***out***.println(**"=======插入操作======="**);  *dataDetails*(rowData.getAfterColumnsList() , logfileName , logfileOffset , dbname , tableName , emptyCount);  }**else** {  *//更改操作* System.***out***.println(**"=======更改操作======="**);  *dataDetails*(rowData.getAfterColumnsList() , logfileName , logfileOffset , dbname , tableName , emptyCount);   }  }  } **catch** (InvalidProtocolBufferException e) {  e.printStackTrace();  }  }  }  **public static void** dataDetails(List<CanalEntry.Column> columns , String fileName , Long offset , String DBname , String tableName , **int** emptyCount){  List<Object> list = **new** ArrayList<Object>();  String sendkey = UUID.*randomUUID*().toString();  **for**(CanalEntry.Column column:columns){  List<Object> ll = **new** ArrayList<Object>();  *//获取原值* ll.add(column.getName());*//字段名* ll.add(column.getValue());*//字段值* ll.add(column.getUpdated());*//是否更改 true代表更改，false代表不更改* list.add(ll);  }  System.***out***.println(fileName + **"#CS#"** + offset +**"#CS#"**+DBname+**"#CS#"**+tableName+**"#CS#"**+list+**"#CS#"**+ emptyCount);  *//将解析后的数据发送到kafka上* KafkaProducer.sendMsg(**"canal"** , sendkey , fileName + **"#CS#"** + offset +**"#CS#"**+DBname+**"#CS#"**+tableName+**"#CS#"**+list+**"#CS#"**+ emptyCount);  } } |

Kafka代码：

|  |
| --- |
| **import** kafka.javaapi.producer.Producer; **import** kafka.producer.KeyedMessage; **import** kafka.producer.ProducerConfig; **import** kafka.serializer.StringEncoder; **import** java.util.Properties; */\*\*  \* Created by angel；  \*/* **public class** KafkaProducer {  **private** String **topic**;  **public** KafkaProducer(String topic){  **super**();  **this**.**topic** = topic ;  }   **public static void** sendMsg(String topic , String sendKey , String data){  Producer producer = *createProducer*();  producer.send(**new** KeyedMessage<String, String>(topic,sendKey,data));  }   **public static** Producer<Integer,String> createProducer(){  Properties properties = **new** Properties();  properties.put(**"zookeeper.connect"** , **"hadoop01:2181"**);  properties.put(**"metadata.broker.list"** , **"hadoop01:9092"**);  properties.put(**"serializer.class"** , StringEncoder.**class**.getName());  **final** ProducerConfig producerConfig = **new** ProducerConfig(properties);  **return new** Producer<Integer,String>(producerConfig);  } } |

### 测试canal代码

1. 启动kafka并创建topic

/export/servers/kafka/bin/kafka-server-start.sh /export/servers/kafka/config/server.properties >/dev/null 2>&1 &

/export/servers/kafka/bin/kafka-topics.sh --create --zookeeper hadoop01:2181 --replication-factor 1 --partitions 1 --topic canal

1. 启动mysql的消费者客户端，观察canal是否解析binlog

/export/servers/kafka/bin/kafka-console-consumer.sh --zookeeper hadoop01:2181 --from-beginning --topic canal

2、启动canal：canal/bin/startup.sh

3、启动mysql：service mysqld start

4、进入mysql：mysql -u 用户 -p 密码；然后进行增删改

### 使用flink将kafka中的数据解析成Hbase的DML操作，然后实时存储到hbase中

|  |
| --- |
| **import** java.util **import** java.util.Properties  **import** org.apache.commons.lang3.StringUtils **import** org.apache.flink.streaming.api.scala.StreamExecutionEnvironment **import** org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer09 **import** org.apache.flink.streaming.util.serialization.SimpleStringSchema **import** org.apache.flink.api.scala.\_ **import** org.apache.flink.runtime.state.filesystem.FsStateBackend **import** org.apache.flink.streaming.api.{CheckpointingMode, TimeCharacteristic} **import** org.apache.hadoop.hbase.{HBaseConfiguration, HColumnDescriptor, HTableDescriptor, TableName} **import** org.apache.hadoop.hbase.client.{ConnectionFactory, Delete, Put} **import** org.apache.hadoop.hbase.util.Bytes   */\*\*  \* Created by angel；  \*/* **object** DataExtraction {  *//1指定相关信息* **val** *zkCluster* = **"hadoop01,hadoop02,hadoop03"  val** *kafkaCluster* = **"hadoop01:9092,hadoop02:9092,hadoop03:9092"  val** *kafkaTopicName* = **"canal"  val** *hbasePort* = **"2181"  val** *tableName*:TableName = TableName.*valueOf*(**"canal"**)  **val** *columnFamily* = **"info"    def** main(args: Array[String]): Unit = {  *//2.创建流处理环境* **val** env = StreamExecutionEnvironment.*getExecutionEnvironment* env.setStateBackend(**new** FsStateBackend(**"hdfs://hadoop01:9000/flink-checkpoint/checkpoint/"**))  env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)  env.getConfig.setAutoWatermarkInterval(2000)*//定期发送* env.getCheckpointConfig.setCheckpointingMode(CheckpointingMode.*EXACTLY\_ONCE*)  env.getCheckpointConfig.setCheckpointInterval(6000)  System.*setProperty*(**"hadoop.home.dir"**, **"/"**);  *//3.创建kafka数据流* **val** properties = **new** Properties()  properties.setProperty(**"bootstrap.servers"**, *kafkaCluster*)  properties.setProperty(**"zookeeper.connect"**, *zkCluster*)  properties.setProperty(**"group.id"**, *kafkaTopicName*)   **val** kafka09 = **new** FlinkKafkaConsumer09[String](*kafkaTopicName*, **new** SimpleStringSchema(), properties)  *//4.添加数据源addSource(kafka09)* **val** text = env.addSource(kafka09).setParallelism(1)  *//5、解析kafka数据流，封装成canal对象* **val** values = text.map{  line =>  **val** values = line.split(**"#CS#"**)  **val** valuesLength = values.length  *//* **val** fileName = **if**(valuesLength > 0) values(0) **else ""  val** fileOffset = **if**(valuesLength > 1) values(1) **else ""  val** dbName = **if**(valuesLength > 2) values(2) **else ""  val** tableName = **if**(valuesLength > 3) values(3) **else ""  val** eventType = **if**(valuesLength > 4) values(4) **else ""  val** columns = **if**(valuesLength > 5) values(5) **else ""  val** rowNum = **if**(valuesLength > 6) values(6) **else ""** *//(mysql-bin.000001,7470,test,users,[uid, 18, true, uname, spark, true, upassword, 1111, true],null,1)  Canal*(fileName , fileOffset , dbName , tableName ,eventType, columns , rowNum)  }    *//6、将数据落地到Hbase* **val** list\_columns\_ = values.map{  line =>  *//处理columns字符串* **val** strColumns = line.columns  *//[[uid, 22, true], [uname, spark, true], [upassword, 1111, true]]* **val** array\_columns = *packaging\_str\_list*(strColumns)  *//获取主键* **val** primaryKey = *getPrimaryKey*(array\_columns)  *//拼接rowkey DB+tableName+primaryKey* **val** rowkey = line.dbName+**"\_"**+line.tableName+**"\_"**+primaryKey  *//获取操作类型INSERT UPDATE DELETE* **val** eventType = line.eventType  *//获取触发的列:inser update* **val** triggerFileds: util.ArrayList[UpdateFields] = *getTriggerColumns*(array\_columns , eventType)  *//因为不同表直接有关联，肯定是有重合的列，所以hbase表=line.dbName + line.tableName* **val** hbase\_table = line.dbName + line.tableName  *//根据rowkey删除数据* **if**(eventType.equals(**"DELETE"**)){  *operatorDeleteHbase*(rowkey , eventType)  }**else**{  **if**(triggerFileds.size() > 0){  *operatorHbase*(rowkey , eventType , triggerFileds)  }   }  }  env.execute()   }     *//封装字符串列表* **def** packaging\_str\_list(str\_list:String):String ={  **val** substring = str\_list.substring(1 , str\_list.length-1)  substring  }    *//获取每个表的主键* **def** getPrimaryKey(columns :String):String = {  *// [uid, 1, false], [uname, abc, false], [upassword, uabc, false]* **val** arrays: Array[String] = StringUtils.*substringsBetween*(columns , **"["** , **"]"**)  **val** primaryStr: String = arrays(0)*//uid, 13, true* primaryStr.split(**","**)(1).trim  }   *//获取触发更改的列* **def** getTriggerColumns(columns :String , eventType:String): util.ArrayList[UpdateFields] ={  **val** arrays: Array[String] = StringUtils.*substringsBetween*(columns , **"["** , **"]"**)  **val** list = **new** util.ArrayList[UpdateFields]()  eventType **match** {  **case "UPDATE"** =>  **for**(index <- 1 to arrays.length-1){  **val** split: Array[String] = arrays(index).split(**","**)  **if**(split(2).trim.toBoolean == **true**){  list.add(*UpdateFields*(split(0) , split(1)))  }  }  list  **case "INSERT"** =>  **for**(index <- 1 to arrays.length-1){  **val** split: Array[String] = arrays(index).split(**","**)  list.add(*UpdateFields*(split(0) , split(1)))  }  list  **case** \_ =>  list   }  }  *//增改操作* **def** operatorHbase(rowkey:String , eventType:String , triggerFileds:util.ArrayList[UpdateFields]): Unit ={  **val** config = HBaseConfiguration.*create*();  config.set(**"hbase.zookeeper.quorum"**, *zkCluster*);  config.set(**"hbase.master"**, **"hadoop01:60000"**);  config.set(**"hbase.zookeeper.property.clientPort"**, *hbasePort*);  config.setInt(**"hbase.rpc.timeout"**, 20000);  config.setInt(**"hbase.client.operation.timeout"**, 30000);  config.setInt(**"hbase.client.scanner.timeout.period"**, 200000);  **val** connect = ConnectionFactory.*createConnection*(config);  **val** admin = connect.getAdmin  *//构造表描述器* **val** hTableDescriptor = **new** HTableDescriptor(*tableName*)  *//构造列族描述器* **val** hColumnDescriptor = **new** HColumnDescriptor(*columnFamily*)  hTableDescriptor.addFamily(hColumnDescriptor)  **if**(!admin.tableExists(*tableName*)){  admin.createTable(hTableDescriptor);  }  *//如果表存在，则开始插入数据* **val** table = connect.getTable(*tableName*)  **val** put = **new** Put(Bytes.*toBytes*(rowkey))  *//获取对应的列[UpdateFields(uname, spark), UpdateFields(upassword, 1111)]* **for**(index <- 0 to triggerFileds.size()-1){  **val** fields = triggerFileds.get(index)  **val** key = fields.key  **val** value = fields.value  put.addColumn(Bytes.*toBytes*(*columnFamily*) , Bytes.*toBytes*(key) , Bytes.*toBytes*(value))  }  table.put(put)  }  *//删除操作* **def** operatorDeleteHbase(rowkey:String , eventType:String): Unit ={  **val** config = HBaseConfiguration.*create*();  config.set(**"hbase.zookeeper.quorum"**, *zkCluster*);  config.set(**"hbase.zookeeper.property.clientPort"**, *hbasePort*);  config.setInt(**"hbase.rpc.timeout"**, 20000);  config.setInt(**"hbase.client.operation.timeout"**, 30000);  config.setInt(**"hbase.client.scanner.timeout.period"**, 200000);  **val** connect = ConnectionFactory.*createConnection*(config);  **val** admin = connect.getAdmin  *//构造表描述器* **val** hTableDescriptor = **new** HTableDescriptor(*tableName*)  *//构造列族描述器* **val** hColumnDescriptor = **new** HColumnDescriptor(*columnFamily*)  hTableDescriptor.addFamily(hColumnDescriptor)  **if**(admin.tableExists(*tableName*)){  **val** table = connect.getTable(*tableName*)  **val** delete = **new** Delete(Bytes.*toBytes*(rowkey))  table.delete(delete)  }  }   } *//[uname, spark, true], [upassword, 11122221, true]* **case class** UpdateFields(key:String , value:String)   *//(fileName , fileOffset , dbName , tableName ,eventType, columns , rowNum)* **case class** Canal(fileName:String ,  fileOffset:String,  dbName:String ,  tableName:String ,  eventType:String ,  columns:String ,  rowNum:String  ) |

测试flink-hbase的程序：

1、启动hbase

2、登录hbase shell

3、启动程序

4、观察数据是否实时落地到hbase

### 打包上线

添加maven打包依赖：

1：打包java程序

|  |
| --- |
| <**build**>  <**sourceDirectory**>**src/main/java**</**sourceDirectory**>  <**testSourceDirectory**>src/test/scala</**testSourceDirectory**>  <**plugins**>   <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-compiler-plugin</**artifactId**>  <**version**>2.5.1</**version**>  <**configuration**>  <**source**>1.7</**source**>  <**target**>1.7</**target**>  *<!--<encoding>${project.build.sourceEncoding}</encoding>-->* </**configuration**>  </**plugin**>   <**plugin**>  <**groupId**>net.alchim31.maven</**groupId**>  <**artifactId**>scala-maven-plugin</**artifactId**>  <**version**>3.2.0</**version**>  <**executions**>  <**execution**>  <**goals**>  <**goal**>compile</**goal**>  <**goal**>testCompile</**goal**>  </**goals**>  <**configuration**>  <**args**>  *<!--<arg>-make:transitive</arg>-->* <**arg**>-dependencyfile</**arg**>  <**arg**>${project.build.directory}/.scala\_dependencies</**arg**>  </**args**>   </**configuration**>  </**execution**>  </**executions**>  </**plugin**>  <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-surefire-plugin</**artifactId**>  <**version**>2.18.1</**version**>  <**configuration**>  <**useFile**>false</**useFile**>  <**disableXmlReport**>true</**disableXmlReport**>  <**includes**>  <**include**>\*\*/\*Test.\*</**include**>  <**include**>\*\*/\*Suite.\*</**include**>  </**includes**>  </**configuration**>  </**plugin**>   <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-shade-plugin</**artifactId**>  <**version**>2.3</**version**>  <**executions**>  <**execution**>  <**phase**>package</**phase**>  <**goals**>  <**goal**>shade</**goal**>  </**goals**>  <**configuration**>  <**filters**>  <**filter**>  <**artifact**>\*:\*</**artifact**>  <**excludes**>  *<!--  zip -d learn\_spark.jar META-INF/\*.RSA META-INF/\*.DSA META-INF/\*.SF  -->* <**exclude**>META-INF/\*.SF</**exclude**>  <**exclude**>META-INF/\*.DSA</**exclude**>  <**exclude**>META-INF/\*.RSA</**exclude**>  </**excludes**>  </**filter**>  </**filters**>  <**transformers**>  <**transformer implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer"**>  <**mainClass**>**canal.CanalClient**</**mainClass**>  </**transformer**>  </**transformers**>  </**configuration**>  </**execution**>  </**executions**>  </**plugin**>  </**plugins**> </**build**> |

2：打包scala程序

将上述的maven依赖红色标记处修改成：

<**sourceDirectory**>**src/main/scala**</**sourceDirectory**>

<**mainClass**>scala的驱动类</**mainClass**>

maven打包步骤：

